RUNTIME VERIFICATION OF PARTIALLY SYNCHRONOUS DISTRIBUTED CYBER-PHYSICAL SYSTEMS

By

Anik Momtaz

A DISSERTATION

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

Computer Science—Doctor of Philosophy

2023

ABSTRACT

This dissertation addresses the problem of runtime verification of *distributed cyber-physical systems* (CPS) with respect to a given *formal specification*. Cyber-physical systems are computer systems with integrated software and physical (hardware) components that, in an ideal environment, seamlessly interact with the real world, as well as each other. Since exhaustively validating *correctness* of a distributed CPS is usually infeasible (if not impossible), many modern validation methods involve *runtime verification* of distributed CPS based on safety properties. Our work focuses on developing time efficient and resource efficient verification techniques that can run in parallel with the execution of these systems to ensure reliability.

In this dissertation, we propose different methodologies to reason about the correctness of distributed CPS in *real-time*, depending on the system settings and architecture. We also provide case studies relevant to each approach in order to demonstrate real-world applications. In all our proposed techniques, we assume a *partially synchronous* setting, where a clock synchronization algorithm guarantees a bound on clock drifts among all signals.

To this end, we first introduce two monitoring methods for distributed systems with *discrete events*, where the specification in the *linear temporal logic* (LTL) [12] is evaluated on a system using (1) a deterministic finite *automaton*-based technique, and (2) a *progression*-based formula rewriting technique.

We then extend this work to detecting violations of *predicates* over distributed *continuous-time* and *continuous-valued* signals in CPS. We introduce a novel *retiming* technique that allows reasoning about the correctness of predicates among continuous-time signals that do not share a global view of time. In addition, we show that leveraging simple knowledge of *physical dynamics* allows for further reduction in run time.

Leveraging the previous two methods, we then introduce a monitoring technique for solving the problem of runtime verification for distributed CPS using the *signal temporal logic* (STL) [36]. We employ a formula progression technique utilizing a signal retiming

method, that enables reasoning about the correctness of formulas among continuous-time and continuous-valued signals in CPS, even when only a partial signal is available.

We also extend our previous work on detecting violations of predicates over distributed signals in CPS from a *centralized* monitoring setting to a *decentralized* monitoring setting. We employ a technique that allows us to indentify *all* possible violations, not just one. Which in turn allows for identification and elimination of bugs from distributed systems regardless of the actual clock drift.

Finally, we introduce the notion of monitoring *reliability* on a network of *monitors* in decentralized monitoring setting. To this end, we present a generalized model of a class of CPS, where each monitor is represented by an *Internet of Things* (IoT) device (or *node*) in a *layered network of producers and consumers*. Our model monitors the events in nodes where resource usage occurs, and captures the tradeoffs between the reliability of the system and resource usage. We present an efficient algorithm to determine the optimal selection of *processing quality* for each node in this *producer-consumer network*, such that target system reliability is achieved while respecting the given resource bounds, and resource usage is minimized. In addition, we present a lightweight *machine learning* based solution to improve our model in terms of run time.

To you, Nuban Mama.
I will forever endeavor to illuminate the void created by your absence
with the light you have inspired.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere and heartfelt gratitude to my PhD advisor, Dr. Borzoo Bonakdarpour, for his unwavering guidance and invaluable mentorship throughout the journey to complete my degree. His deep expertise, relentless dedication, and insightful feedback have been instrumental in shaping the quality and depth of my research. It is impossible for me to overstate the pivotal role his support has played in my academic growth. I am truly fortunate to have had the privilege of working under his tutelage. I would also like to express my gratitude to the remaining members of my PhD guidance committee, Dr. Betty Cheng, Dr. Bahare Kiumarsi, and Dr. Sandeep Kulkarni, for their continuous support and indispensable feedback on my research.

Throughout my academic endeavors, I have had the pleasure of closely working with Dr. Houssam Abbas, from Oregon State University. His immeasurable contributions to my work in runtime verification of signals have made it possible to produce multiple high-quality papers, including one on predicate detection for signals that received the Best Paper Award at The 21^{st} International Conference on Runtime Verification. I have also co-authored several papers with my exceptional colleague and dear friend, Ritam Ganguly. In every paper we co-authored, his contributions were instrumental, and second to none. My sincere gratitude is also extended to the rest of my brilliant colleagues, Oyendrila Dobe, Tzu-Han Hsu, and Eshita Zaman, who generously devoted many hours from their busy schedules to review and provide valuable suggestions on numerous aspects of my research.

Words cannot fully express the depth of my gratitude to my family for giving me unconditional love, and continuous encouragement from halfway across the globe. I want to specifically extend my heartfelt thanks to my parents, Asfia Sabina (*Ammu*) and Motazid Momtaz (*Abba*), my little sister, Monisha Momtaz (*Monomono*), my aunt, Simin Seury (*Khamma*), and my grandmother, Banu Tarafdar (*Didda*). Additionally, I am immensely grateful to my wonderful wife, Tiana Momtaz, for her love, support, and boundless sacrifices throughout this journey, and for always believing in me when, at times, even I could not.

I am also thankful to Sadika Amreen, Reazul Hoque, Balabhadra Khatiwada, Meena Khatiwada, and Emily Mui, for being extraordinary friends, and possessing the remarkable ability to make good times better, and not-so-good times bearable.

Last, but most certainly not least, I must express my gratitude to Michigan State University, specifically the Department of Computer Science and Engineering, for affording me the opportunity to pursue my dream of obtaining a PhD in a field I am passionate about. I am truly and unequivocally proud to call myself a Spartan.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AATC Automated Air Traffic Control

ANN Artificial Neural Network

AP Atomic Proposition

CLA Cold Leg Accumulator

CPS Cyber-Physical Systems

CPU Central Processing Unit

ECCS Emergency Core Cooling System

FAA Federal Aviation Administration

HLC Hybrid Logical Clock

IoT Internet of Things

LTL Linear Temporal Logic

MPC Multi-Party Computation

MTL Metric Temporal Logic

NTP Network Time Protocol

PTP Precision Time Protocol

PVC Physical Vector Clock

RAM Random Access Memory

RWST Refueling Water Storage Tank

SMT Satisfiability Modulo Theory

STL Signal Temporal Logic

UAV Unmanned Aerial Vehicle

CHAPTER 1

INTRODUCTION

Distributed monitoring is the process of analyzing the execution of distributed systems with a *centralized* or *decentralized* monitor in relation to a given formal specification. While attempting to complete a collaborative job, distributed systems often consist of numerous systems that do not share a *global clock* and memory. In a distributed database, for example, data is kept in several physical locations, usually spread across a network of interlinked computers. A monitor may want to guarantee that queries to the distributed database fulfill some form of consistency requirements. The class of systems containing both software and physical (hardware) components that interact with the actual world as well as each other is a prominent class of distributed systems. These systems are referred to as *cyber-physical systems* (CPS) [122].

Our reliance on CPS has grown rapidly over the past decade, as these systems are more and more frequently deployed over networks of agents due to the emergence of the *Internet of Things* (IoT) and edge applications [27]. Therefore, validating the accuracy of these systems, especially for the class of CPS that is safety-critical, is now of paramount importance. Software applications deployed among networked nodes, referred to as agents, are with a critical class of CPS. Examples include autonomous car fleets, sensor networks in infrastructure, health-monitoring wearables, and medical device networks. Because CPS are often safety-sensitive, obtaining assurance regarding their accuracy is vital. CPS are distinguished by three defining characteristics:

- **First**, because the signals are *analog*, they include an infinite number of events, rendering traditional reasoning approaches designed for discrete systems ineffective, if not inapplicable in most circumstances. The applications we target, such as those mentioned above, require continuous-time behavior. It is not enough, for example, to assert that a voltage does not spike at sample times. As a result, increasing the signal sample rate does nothing to alleviate the necessity for analog signal reasoning.

- **Second**, each agent in these CPS has a *local clock* that drifts from the clocks of other agents. Hence, the concept of time, which is taken for granted in centralized systems, must be changed, as it is unclear whether events are consecutive and concurrent. Furthermore, it is unclear how continuous events in various processes respect the *happened before* relation [73], and how one may reason about the sequence of occurrence of continuous events.
- **Third**, CPS signals obey physical laws and dynamics. An understanding of these dynamics may be used to reason about distributed signals and predict their behavior, as well as improve efficiency of reasoning.

The characteristics listed above define the concept of distributed signals, and reasoning about them necessitates the establishment of some notion of *ordering*. Building such ordering for an infinite number of events from different signals while clock drifts occur at runtime is a difficult undertaking.

1.1 Motivating Examples

We demonstrate the crucial need of monitoring distributed CPS through a critical application in automated air traffic control (AATC). The market for unmanned aerial vehicles (UAVs) is expanding rapidly [61]. In the United States, the Federal Aviation Administration (FAA) envisions a federated framework in which UAVs that contribute in monitoring global air safety parameters are rewarded with faster free-flight pathways to their destinations [39, 43].

To support this federated structure, AATC tower software must monitor analog inputs such as UAV location and velocity to determine if they violate global instantaneous safety characteristics, also known as predicates. These predicates are Boolean expressions defined over the concurrent states of the several CPS agents, such as mutual separation, conditional speed limitations, and minimal energy storage. These predicates must be evaluated on the global state, which is the combined state of all UAVs at the same time. However, in the absence of a perfect shared clock across all UAVs, UAV_1's clock may report $\text{t} = 5$ and UAV_2's

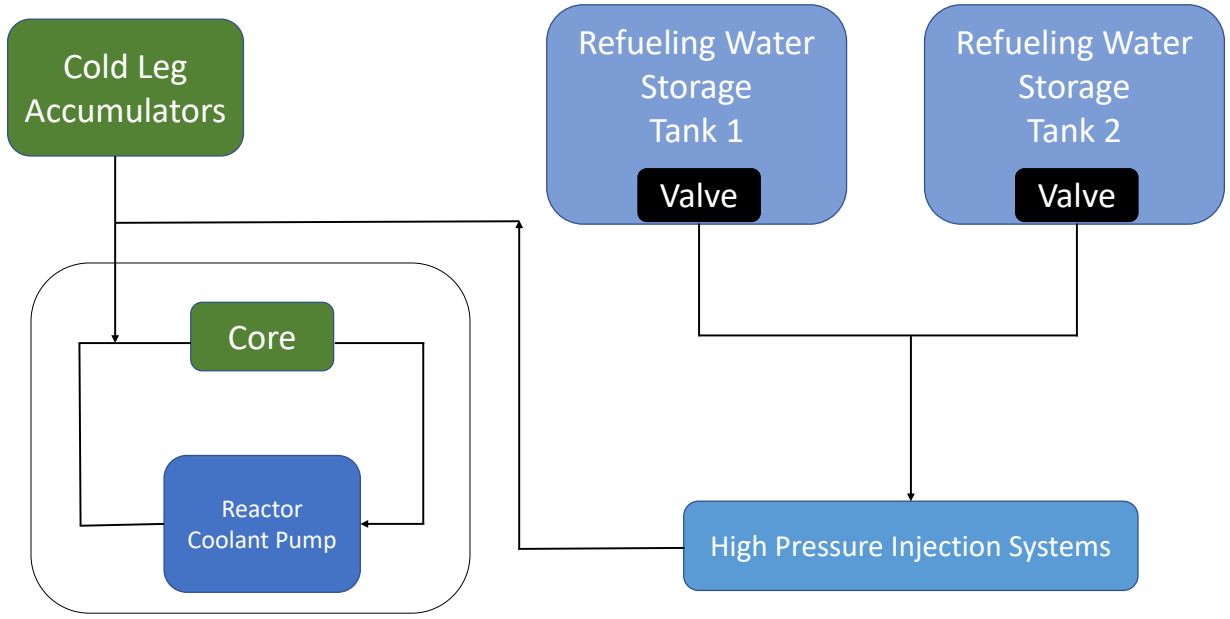


Figure 1.1 Hybrid dynamic cooling system with water tanks.

clock may report t = 5.2 at the same physical 'real' time. Equivalently, the same value on two clocks may represent distinct physical moments. If the central AATC monitor relies on these two states to determine if the predicate has been violated, then it may result in false negatives (i.e., missing violations) or false positives (i.e., declaring a violation when none exists).

The UAV example has two characteristics that are shared by many different distributed CPS: First, while perfect continuous-time synchrony is often impossible to achieve, clock synchronization algorithms such as Network Time Protocol (NTP) [88] ensure that drift among local clocks remains within some bounds. Second, the central monitor frequently recognizes certain restrictions on the UAV dynamics, such as velocity limits. In this case, the AATC tower would be aware of the UAVs' speed limitations. In developing our solution, we make use of these two characteristics.

As another example, consider the water distribution system shown in Figure 1.1, where several tanks deliver water to an offsite location via a common pipe. Water tank outflow rate and pressure are monitored locally using drifting local clocks. If the compounded pressure

or flow rate on the pipe is a concern and has to be monitored, correctly measuring these values becomes difficult since the continuous signals indicating the pressure and/or flow rate of the tanks are not synchronized. If the flow rate and pressure must always remain below a given threshold, clock drift among the local clocks may cause values for which the threshold is breached to be missed.

1.2 Challenges

While there are approaches for monitoring temporal logic for distributed discrete-event systems (e.g., [49, 58, 96, 99]), we still lack a good understanding of distributed CPS. Although the literature on distributed computing is decades old, and many important problems have been solved in the context of discrete-event systems, the main challenge with distributed monitoring is that it is not always possible for the monitor to establish the right order of occurrence of events across different agents in the absence of a global clock. Given the non-deterministic nature of distributed programs, it is expected of a runtime monitor to provide multiple results for the same distributed computation. This leads to a combinatorial explosion of possibilities that the monitor must examine at runtime, making the task computationally costly.

Monitoring and detecting violations of formal specifications is a common and effective technique to reasoning about the health of CPS. Broadly speaking, the state of the art in runtime monitoring focuses on either (1) centralized monitoring for stand-alone applications or multi-agent systems that share a global clock while being blind to system dynamics [1, 5, 34, 35, 33, 83], or (2) decentralized monitoring in pure discrete-time for ordering discrete events [10, 16, 26, 28, 31, 46, 47, 49, 55, 64, 96, 99, 58], which is appropriate for pure software, but not CPS. As a result, solutions for monitoring CPS where analog signals are created by distributed agents that do not share a global clock are currently lacking (see the related work in Chapter 8). Lack of synchronization, in particular, poses substantial issues since the monitor must reason about signal levels at distinct agents' local times, which may result in conflicting monitoring verdicts. This problem is exacerbated by the fact that agents often

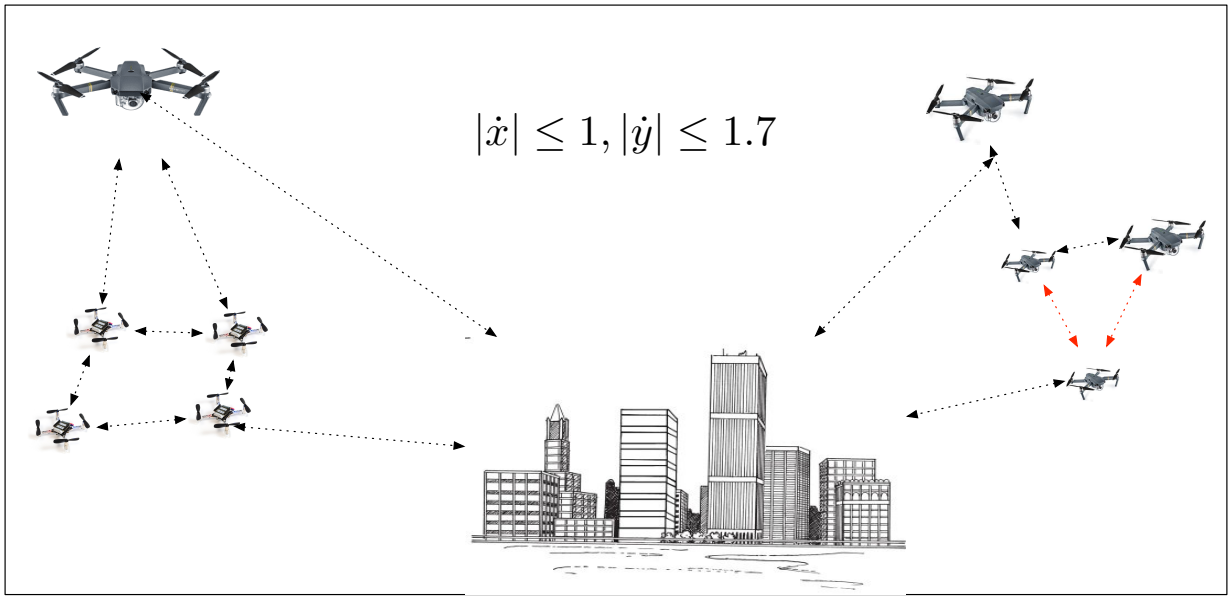


Figure 1.2 A distributed CPS composed of autonomous aerial vehicles with drifting clocks. The violation property to be monitored is, for any two aerial vehicles the distance along x axis is within 1 and the distance along y axis is within 1.7. Asynchronous signals produced by the vehicles must be monitored for predicate violations, while leveraging some knowledge of system dynamics.

communicate with one another, imposing extra limits on event ordering. Furthermore, in a distributed system, a central monitor that receives all signals is subject to a single point of failure. That is, if the monitor fails, predicate detection fails altogether.

In decentralized monitoring, the concept of reliability of a network of monitors adds another layer to the list of challenges. To handle trade-offs, most systems use manual controls. Some network applications, for example, enable administrators to alter the quality of malicious activity detection systems based on predicted traffic [92, 134]. This strategy frequently focuses on a subset of resources and lacks the flexibility required by huge dynamic systems. Another strategy is to aggressively over-provision the processing infrastructure in terms of machine capabilities (e.g., CPU, memory, etc.), network bandwidth, and assigned power budget to ensure that no limitations are reached [15]. This is an extremely expensive approach that is frequently not feasible and is not future-proof.

The major problem in resource management and optimization is that monitors in a network often receive data, process it, and then transmit it to succeeding monitors. This results

in a quality vs. cost trade-off across distinct monitors, where resources are determined not simply by pairs of consecutively interacting monitors, but by the interaction of all monitors in the network. In other words, lowering the processing quality of a monitors might have an impact on subsequent monitors in the network that receive lower quality data. This means that quality versus resource utilization must be optimized across the entire network, not simply for pairs of monitors communicating with each other. On top of that, it is easy to see that quality and resource utilization are frequently at odds; that is, greater quality and dependability need higher resource usage, making optimization more challenging.

1.3 Thesis Statement

Now that we have provided challenges and motivation for this dissertation, in this section, we define the statement of thesis as follows:

Thesis Statement

It is possible to develop trustworthy verification methodologies under both centralized and decentralized monitoring settings in order to reason about the correctness of safety-critical partially synchronous distributed cyber-physical systems in real-time.

1.4 Contributions

In this dissertation, we take steps toward rigorous, automated reasoning about distributed CPS, the accuracy and integrity of which is critical to ensuring the safety of the environment in which they function. Based on the proposed verification approaches, our contributions are grouped into five primary segments. These techniques differ in terms of (1) system architecture (i.e., discrete events vs. continuous time), (2) monitor architecture (i.e., centralized vs. decentralized), and (3) specification language (i.e., LTL vs. STL).

1.4.1 Monitoring Discrete-Event Systems using LTL

First, we present two sound and complete solutions to the problem of distributed runtime verification (RV) with regard to LTL formulas. Both approaches employ a fault-proof

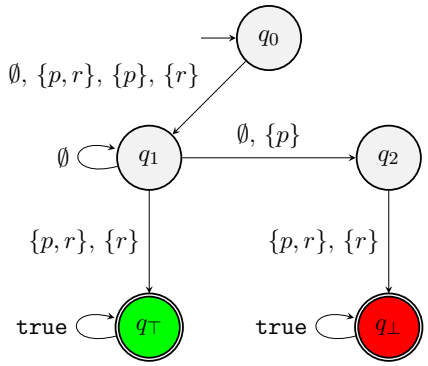


Figure 1.3 Monitoring automaton for formula φ.

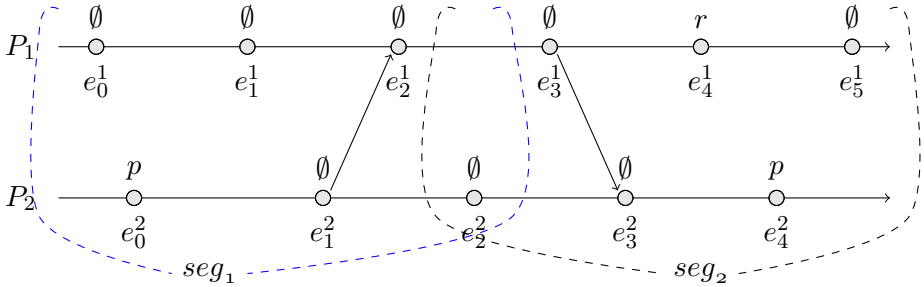


Figure 1.4 A distributed computation.

central monitor, and to address the explosion of various interleavings, we propose a practical assumption, namely, a bounded skew ε between local clocks of each pair of processes, which is guaranteed by a fault-proof clock synchronization mechanism (e.g., NTP [88]). This implies that time instants from multiple local clocks within ε are deemed concurrent, i.e., their order of occurrence cannot be determined. This is a partial synchrony setting that does not presume a global clock but restricts the impact of asynchrony within clock drifts.

Our first approach is based on constructing the LTL_3 [12] monitor automaton of an LTL formula and constructing multiple Satisfiability Modulo Theory (SMT)[6] queries to determine which states of the monitor automaton are reachable for a given distributed computation. For example, Figure 1.3 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 Figure 1.4. We transform our monitoring decision problem into an SMT solving problem. The SMT instance includes constraints that encode (1) our monitoring algorithm based on the 3-valued semantics of LTL, (2) behavior of

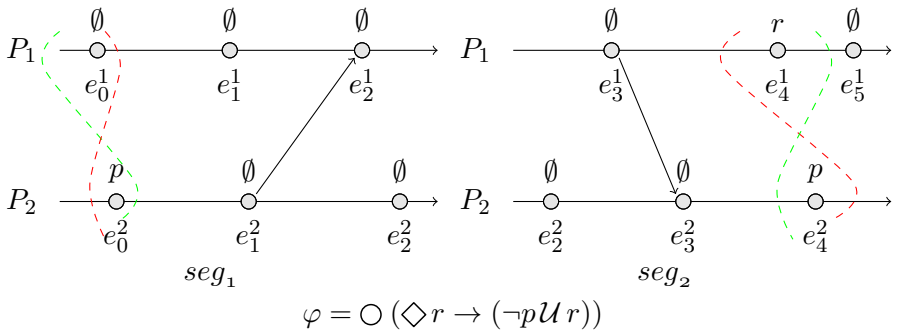


Figure 1.5 Progression and segmentation.

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. Afterwards, it attempts to concretize an uninterpreted function whose evaluation provides the possible verdicts of the monitor with respect to the given computation. We divide a computation into multiple segments to make the verification problem tractable, significantly reducing the search space of each SMT query. Thus, the result of monitoring each segment (the possible LTL_3 states) should be carried to the next segment. Furthermore, because distributed applications are now operated on large cloud services, we extend our method to a parallel monitoring algorithm to take use of the available computational resources and gain greater 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 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 monitoring 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.

We create a formula progression approach to convey the results of monitoring from one segment to the next. 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 that yields `true`, `false`, or an LTL formula based on what has been seen thus far. We emphasize fundamental distinction between our approach and the standard rewriting technique [59] is that, function Praccepts a finite trace as input, whereas the algorithm in [59] rewrites the input LTL formula in a state-by-state manner. This suggests that rewriting based on the fixed point representation of temporal operators is not possible in our context. Our motivation stems from the fact that when a given distributed computation is divided into a number of segments, a state-by-state rewriting approach will generate too many SMT queries, rendering it unscalable. For example, in Figure 1.5 (which is the computation in Figure 1.4 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 $\Box(\neg p)$) as compared to [49]. 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 in [49].

We make a detailed comparison between the proposed approaches 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 [85]. Our experiments show that the progression-based approach has 35% reduced overhead as compared to the automata-based approach.

1.4.2 Monitoring Predicates on CPS

We provide a sound and complete solution to the problem of predicate monitoring for distributed systems when extended to CPS. Our system, which employs a central monitor to receive distributed signals, may be characterized as follows: We assume a clock synchronization mechanism guarantees limited skew ε between all local clocks. That is, time instants from separate clocks within ε are regarded concurrent, i.e., their sequence of occurrence cannot be determined below an ε of resolution. The limited skew assumption is used to supplement the classic happened-before relation [73]. We introduce a retiming technique that

leverages the concept of retiming functions from stochastic processes to make the monitor align the locally timed agent signals. A retiming function aligns the supports of two signals while taking into account the order, ε-skew, and arbitrary message exchanges between agents. Our monitoring decision problem is transformed into a *Satisfiability Modulo Theory* (SMT) problem that seeks a retiming function that observes a predicate violation. We show how to simplify the general SMT problem of searching for arbitrary retiming functions to the considerably simpler problem of looking for piece-wise linear retimings. Furthermore, knowledge about agent dynamics constraints may be used to decrease monitoring overhead. The following are our contributions:

1. An SMT-based algorithm for centralized monitoring of distributed analog signals for predicate violations, supplemented with a clock synchronization algorithm that ensures finite skew between all local clocks, employing the classic happened-before relation. [73];

2. A signal retiming approach based on the concept of retiming functions as used in stochastic processes to address the challenges presented by time asynchrony;

3. A lightweight approach for adding system dynamics constraints in order to decrease monitoring overhead;

4. An analysis of the relationship between monitoring overhead's sensitivity to the skew bound and the quantity of communication between agents, and

5. A method for parallelizing the monitoring algorithm in order to improve scalability.

We have fully implemented our methodologies and provide the results of experiments on monitoring a network of autonomous ground vehicles (in the real world), aerial vehicles (in simulation), and a water distribution system (in simulation). It should be noted that systems with a central monitor are inherently vulnerable to a single point of failure. Our work is concerned with establishing the suggested theory and does not take into consideration fault tolerance. The following are our observations. First, while our solution is based on SMT

solving, it may be used for online monitoring if the monitor is run at an acceptable frequency (i.e., the monitoring overhead does not exceed the system's regular operating time). Second, adding knowledge of system dynamics is hugely beneficial in decreasing monitoring overhead. In some cases, the speedup (as compared to when the information is not used) can be an order of magnitude. Third, when practical clock synchronization protocols (e.g., NTP and PTP) are used, monitoring overhead is independent of clock skews. Finally, we notice that communication between agents does not always reduce monitoring overhead in the continuous-time context; this contradicts popular perception in the discrete-time situation, where communication event orderings are thought to make automated reasoning more efficient.

1.4.3 Monitoring CPS using STL

We expand our approach from monitoring just Boolean predicates across distributed signals to whole signal temporal logic (STL) [36]. To this end, we start with a partially synchronous scenario, in which a clock synchronization mechanism ensures a maximum bound ε on clock drifts across all signals. This can be ensured by off-the-shelf algorithms such as NTP [88]. We use the signal retiming approach presented in [95] to align continuous-time signals that do not share a global sense of time. Assuming the bound ε, the decision problem is to find a retiming function that violates an input STL formula. If no such function exists, then it indicates that the distributed signals have not yet broken the formula (it may or may not in the future).

To minimize the size of a distributed signal to more manageable smaller problems, we break the original signal into smaller signals known as segments. The problem here is that the outcome of monitoring one segment should be carried over to the next. For example, consider STL formula $\varphi = \Box_{[0,5]}\, p$ (which means proposition p should hold at all times in time interval $[0, 5]$) and the current segment of signals that end at time 3. This means if p holds in the interval $[0, 3]$, then the formula has to be rewritten to $\varphi' = \Box_{[0,2]}\, p$ for the second segment. Of course, such rewriting can become challenging when the formulas have multiple

nested temporal operators with relative time intervals. To this end, we propose a *formula progression* technique that takes as inputs an STL formula and a finite-time distributed signal σ and returns an STL formula φ' such that for any extension σ', we have $\sigma\sigma' \models \varphi$ if and only if $\sigma' \models \varphi'$. We encode the resulting problem as a (SMT) problem that searches for a retiming function given the constraints of the current segment and STL formula. We provide approaches for solving the SMT encoding efficiently. We should highlight that we are not concerned in this dissertation with problems such as monitoring fault-tolerance (i.e., we assume a flawless centralized monitor with no noise or communication failures).

We have fully implemented our approach on two distributed CPS applications: monitoring of a (1) network of aerial vehicles for a set of properties such as mutual separation and formation, and (2) a water distribution system for the property in which the outflow pressure exceeds the threshold pressure. The results indicate that in some circumstances, a distributed CPS can be monitored fast enough for online deployment.

1.4.4 Decentralized Monitoring Predicates on CPS

In order to address the issue of single point of failure in a distributed system, we also expand our approach of centralized predicate detection for distributed CPS with drifting clocks under partial synchrony to a decentralized monitoring approach. To this end, our contributions are as follows:

1. A fully decentralized monitoring approach, where each agent only has access to its own signal, and exchanges a limited amount of information with other agents;

2. A detection technique that identifies all violating predicates, not just one;

3. An online algorithm applying a class of global properties that are conjunctions of local propositions, that can be executed in parallel to tasks carried out by agents;

4. A novel *physical vector clock* that orders continuous-time events in a distributed computation without a shared clock, and

5. A method to deploy our algorithm to existing infrastructure. Specifically, our algorithm includes a modified version of the classical detector described in [26] that can be deployed on top of existing infrastructure.

Our methodologies are fully implemented, and we provide the results of experiments on two synthetically generated signal datasets.

1.4.5 Monitoring Reliability in a Multi-Layered CPS

Finally, we introduce the notion of monitoring reliability on a network of monitors in decentralized monitoring setting. To this end, we present a generalized model of a class of CPS, where each monitor is represented by an (IoT) device or a node in a layered network of producers and consumers. Assuming a layered producer-consumer network with stream processing, each node in the network faces a trade-off between processing quality and resource utilization. An abstract model of stream processing applications is presented. The processing nodes, in particular, are modeled as a network of producers-consumers, which is a directed acyclic graph in which a node can be a producer, a consumer, or both based on its incoming/outgoing edges. Each node in the network consumes data that flows through its incoming edges and produces data that flows through its outgoing edges. The processing of data consumed/produced by a node can be done at various quality levels. The quantity of resources utilized by the node is determined by the processing quality level. Power, energy, RAM, disk, or network bandwidth are all examples of resources. In addition to these resources, we represent reliability as a nonrenewable resource that flows across the network and is partially depleted based on the quality levels of the nodes through which it flows. Individual and collective resource limits and bounds apply to nodes. Lower quality leads to more error, which propagates across the network and has the potential to affect the quality of subsequent nodes as well as overall reliability. Our goal is to provide an efficient framework for modeling a system in such a way that resource bounds are respected and a designer-specified goal is optimized. This goal is supplemented with optimization objectives such as optimizing reliability and minimizing energy or other resource usage in the system.

To answer the above-mentioned multi-objective optimization problem, we reduce it to the satisfiability problem for the satisfiability modulo theory (SMT). SMT-solving technology has advanced dramatically over the last two decades [77], and we use its improvements to solve our problem. To that end, we represent (1) the elements of the producers-consumers graph, as well as the concepts of data rates, quality, reliability, and resource consumption, as SMT entities (e.g., variables, functions, constants, and so on), (2) the resource constraints and bounds as a set of SMT constraints, (3) the pillars of our original optimization problem as additional SMT constraints that will be checked and searched using a binary search algorithm to find the optimal solution, and (4) a machine learning based model that aims to even further optimize the problem in terms execution time at the cost of minimal loss in accuracy.

The SMT aspects of our technique is implemented using the SMT-solver Z3 [32] and the machine learning aspects of our technique is implemented using the machine learning toolkit Scikit-learn [101] and Keras [65] artificial neural network interface. Our model aims to optimize reliability and resource consumption trade-offs. We explore these trade-offs through detailed synthetic experiments. We also apply our techniques on a real-world case study, where we optimize a network of embedded streaming devices, so that the network (1) delivers the best possible performance using the available resources, or it (2) uses the minimal amount of a certain resource while meeting a given performance goal.

1.5 Organization

This chapter (Chapter 1) provided an overview of the motivation, challenges and contributions of this dissertation. The remainder is organized as follows. Chapter 2 discusses the background for our work. Chapter 3 provides details on our runtime verification of distributed systems using automata-based and progression-based techniques. Chapter 4 extends this work to CPS and Boolean predicate detection. Chapter 5 further extends this work from Boolean predicates to STL, whereas, Chapter 6 extends this from a centralized monitoring setting to a decentralized monitoring setting. Chapter 7 introduces the notion of

reliability and provides a resource optimization technique. Chapter 8 elaborates on related work, and finally Chapter 9 summarizes the findings, discusses ongoing work and suggests avenues for further research.

CHAPTER 2

PRELIMINARIES

In this Chapter, we present the background concepts of our work. We start with the formal specification languages we use in our approaches, and then introduce other crucial background components of our work.

2.1 Linear Temporal Logics (LTL)

Let AP be a set of *atomic propositions* and $\Sigma = 2^{\mathsf{AP}}$ be the set of all possible *states*. A *trace* is a sequence $s_0 s_1 \ldots$, where $s_i \in \Sigma$ for every $i \geq 0$. We denote by Σ^* (resp., Σ^ω) the set of all finite (resp., infinite) traces. For a finite trace $\alpha = s_0 s_1 \ldots s_k$, $|\alpha|$ denotes its *length*, $k+1$. Also, for $\alpha = s_0 s_1 \ldots s_k$, by α^i, we mean trace $s_i s_{i+1} \ldots s_k$ of α.

2.1.1 Infinite-trace Semantics of LTL

The syntax and semantics of the *linear temporal logic* (LTL) [104] are defined for infinite traces. The syntax is defined by the following grammar:

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

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

$(\sigma, i) \models p$	iff	$p \in s_i$
$(\sigma, i) \models \neg\varphi$	iff	$(\sigma, i) \not\models \varphi$
$(\sigma, i) \models \varphi \vee \psi$	iff	$(\sigma, i) \models \varphi$ or $(\sigma, i) \models \psi$
$(\sigma, i) \models \bigcirc\varphi$	iff	$(\sigma, i+1) \models \varphi$
$(\sigma, i) \models \varphi \,\mathcal{U}\, \psi$	iff	$\exists k \geq i : (\sigma, k) \models \psi$ and $\forall j \in [i, k) : (\sigma, j) \models \varphi$

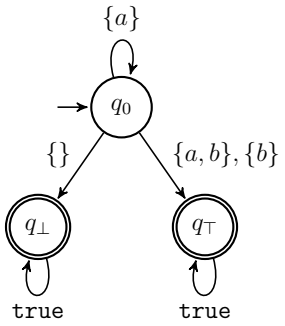


Figure 2.1 LTL_3 monitor for $\varphi = a \,\mathcal{U}\, b$.

2.1.2 Finite-trace Semantics of LTL

In the context of RV, the 3-valued LTL (LTL_3 for short) [12] evaluates LTL formulas for *finite* traces, but with an eye on possible future extensions, whereas the finite LTL, or FLTL [80] solely considers the present trace with no regard for the future. In LTL_3, the set of truth values is $\mathbb{B}_3 = \{\top, \bot, ?\}$, where $\top$ (resp., $\bot$) denotes that the formula is *permanently* satisfied (resp., violated), regardless of how far 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 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 1. The 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 $\alpha \in \Sigma^*$. ■

As an example, Figure 2.1, shows the monitor automaton for formula $\varphi = a \,\mathcal{U}\, b$. FLTL has the same syntax as LTL, and its 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:

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

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

Consider the formula $\varphi = \Box p$, and a finite trace $\alpha = s_0 s_1 \cdots s_n$ to further illustrate the difference between LTL and FLTL and LTL_3. If $p \notin s_i$ 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 $p \notin s_i$ for all $i \in [0,n]$, then $[\alpha \models_3 \varphi] =?$. This is because there exist infinite extensions 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.

2.2 Distributed Computation

We assume a loosely coupled asynchronous message passing system, consisting of n reliable processes (that do not fail), denoted by $A = \{A_1, A_2, \dots, A_n\}$, without any shared memory or global clock. Channels are assumed to be First In, First Out (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 that acts as a '*real*' timekeeper. It is to be understood,

however, that this global clock is a theoretical object used in definitions, and is *not* available to the processes.

We make a practical assumption, known as *partial synchrony* [40]. The *local clock* (or time) of a process A_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 $c_i(\chi)$ is the value of the local clock at global time χ. Therefore, for any two processes A_i and A_j, we have:

$$\forall \chi \in \mathbb{R}_{\geq 0}. |c_i(\chi) - c_j(\chi)| < \varepsilon$$

with $\varepsilon > 0$ being the maximum *clock skew*. The value ε is assumed to be fixed and known by the monitor in the rest of this dissertation. 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 [88], to ensure bounded clock skew among all processes.

An *event* in process A_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 χ, that is, $\tau = c_i(\chi)$. We assume that for every two events $e^i_{\tau,\sigma}$ and $e^i_{\tau',\sigma'}$, we have $(\tau < \tau') \Leftrightarrow (\sigma < \sigma')$.

Definition 2. 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 [73], subject to the partial synchrony assumption:

- In every process A_i, $1 \leq i \leq N$, all events are totally ordered, that is,

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

- 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 A_i and A_j, and any two events $e^i_{\tau,\sigma}, e^j_{\tau',\sigma'} \in \mathcal{E}$, if $\tau + \varepsilon < \tau'$, 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 3. Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$, a subset of events $C \subseteq \mathcal{E}$ is said to form a *consistent cut* iff when C contains an event e, then it contains all events that happened-before e. Formally, $\forall e \in \mathcal{E}.(e \in C) \wedge (f \rightsquigarrow e) \rightarrow f \in C$. ■

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

2.3 Hybrid Logical Clocks

A *hybrid logical clock* (HLC) [71] 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 A_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 A_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, $\sigma - \tau \leq \varepsilon$).
  - For any two processes A_i and A_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.
- $\omega : \mathcal{E} \rightarrow \mathbb{Z}_{\geq 0}$ is a function that maps each event in $\mathcal{E}$ to the causality updates, where:
  - For any process A_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.

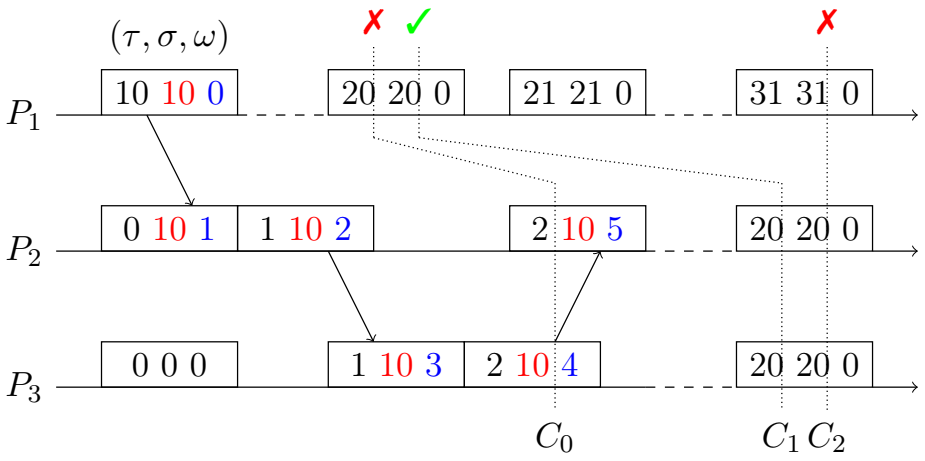


Figure 2.2 HLC example.

- For any two processes A_i and A_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 on $\max\{\sigma, \sigma', \tau\}$.
- For any two processes A_i and A_j, and any two events $e^i_{\tau,\sigma,\omega}, e^j_{\tau',\sigma',\omega'} \in \mathcal{E}$, $(\tau = \tau') \wedge (\omega < \omega') \rightarrow e^i_{\tau,\sigma,\omega} \rightsquigarrow e^j_{\tau',\sigma',\omega'}$.

We presume that HLC is fault-proof in our implementation. Figure 2.2 depicts an HLC with partially synchronous concurrent timelines of three processes with $\varepsilon = 10$. Note that the local times of all events in $\mathsf{front}(C_1)$ are bounded by ε. As a result, C_1 is a consistent cut, but C_0 and C_2 are not.

2.3.1 Physical Vector Clocks

We first define Physical Vector Clocks (PVCs), which generalize vector clocks [81] from countable to uncountable sets of events. They are used by the abstractor process (next section) to track the happened-before relation. A PVC captures one agent's knowledge, at appropriate local times, of events at other agents.

Definition 4. Given a distributed signal $(E, \rightsquigarrow)$ on N agents, a *Physical Vector Clock*, or PVC, is a set of N-dimensional timestamp vectors $\mathbf{v}^t_n \in \mathbb{R}^N_+$, where vector $\mathbf{v}^t_n$ is defined by the following:

1. Initialization: $\mathbf{v}^0_n[i] = 0, \quad \forall i \in \{1, \ldots, N\}$

2. Timestamps store the local time of their agent: $\mathbf{v}_n^t[n] = t$ for all $t > 0$.

3. Timestamps keep a consistent view of time: Let V_n^t be the set of all timestamps $\mathbf{v}_m^s$ s.t. e_m^s happened-before e_n^t in E. Then:

$$\mathbf{v}_n^t[i] = \max_{\mathbf{v}_m^s \in V_n^t} (\mathbf{v}_m^s[i]), \quad \forall i \in [N] \setminus \{n\}, t > 0$$

PVCs are partially ordered: $\mathbf{v}_n^t < \mathbf{v}_m^{t'}$ iff $\mathbf{v}_n^t \neq \mathbf{v}_m^{t'}$ and $\mathbf{v}_n^t[i] \leq \mathbf{v}_m^{t'}[i]$ $\forall i \in [N]$. ■

We say $\mathbf{v}_n^t$ is *assigned* to e_n^t. The detection algorithm can now know the happened-before relation by comparing PVCs.

Lemma 1. Let $n \neq m$ and $t, t' \neq 0$. Then $(e_n^t \rightsquigarrow e_m^{t'})$ iff $(\mathbf{v}_m^{t'}[n] \geq t)$.

Proof. We split the bidirectional implication into its two directions:

1. $(e_n^t \rightsquigarrow e_m^{t'}) \implies (\mathbf{v}_m^{t'}[n] \geq t)$

   Since $\mathbf{v}_n^t[n] = t$ by Definition 4 2 and $e_n^t \rightsquigarrow e_m^{t'}$, then by Definition 4 3, $\mathbf{v}_m^{t'}[n] \geq t$.

2. $(e_n^t \rightsquigarrow e_m^{t'}) \impliedby (\mathbf{v}_m^{t'}[n] \geq t)$

   a) Case $(\mathbf{v}_m^{t'}[n] = t) \implies (e_n^t \rightsquigarrow e_m^{t'})$:

   Besides initialization, the only case in Definition 4 where a value is assigned which did not come from another timestamp is Definition 4 2. Consider an event e_n^t. The timestamp of this event at index n is t, by Definition 4 2. At the point in time when this event is created (local time t on agent A_n), no other timestamp has the value t at index n. All other $\mathbf{v}_m^{t'}$ which have the value t at index n must be assigned by Definition 4 3. This means that they have the relation $e_n^t \rightsquigarrow e_m^{t'}$, due to the transitive property of the happened-before relation.

   b) Case $(\mathbf{v}_m^{t'}[n] > t) \implies (e_n^t \rightsquigarrow e_m^{t'})$:

   Consider a t'' where $\mathbf{v}_m^{t'}[n] = t''$ and $t'' > t$. Then by the previous case, $e_n^{t''} \rightsquigarrow e_m^{t'}$. Since by the happened-before relation all events on an agent are totally ordered (Definition 7 2), $e_n^t \rightsquigarrow e_n^{t''}$. By the transitive property of the happened-before relation (Definition 7 2), $e_n^t \rightsquigarrow e_m^{t'}$.

■

Theorem 1. Given a distributed signal $(E, \rightsquigarrow)$, let V be the corresponding set of PVC timestamps. Then $(V, <)$ and $(E, \rightsquigarrow)$ are order isomorphic, i.e., there is a bijective mapping between V and E s.t. $e_n^t \rightsquigarrow e_m^{t'}$ iff $\mathbf{v}_n^t < \mathbf{v}_m^{t'}$.

Proof. Since each PVC timestamp corresponds to exactly one event and all events have a timestamp, there is clearly a bijective mapping. To show it preserves order, we need to confirm that $(e_n^t \rightsquigarrow e_m^{t'}) \iff (\mathbf{v}_n^t < \mathbf{v}_m^{t'})$.

1. $e_n^t \rightsquigarrow e_m^{t'} \implies \mathbf{v}_n^t < \mathbf{v}_m^{t'}$

   By Definition 4 3, each element of $\mathbf{v}_n^t$ must be less than or equal to the corresponding element of $\mathbf{v}_m^{t'}$. So then we need to show that $\mathbf{v}_n^t \neq \mathbf{v}_m^{t'}$. Definition 4 2 indicates that $\mathbf{v}_m^{t'}[m] = t'$. By Theorem 1 if $\mathbf{v}_n^t[m] = t'$ then $e_m^{t'} \rightsquigarrow e_n^t$; but there cannot be cycles in the happened-before order relation, then $\mathbf{v}_n^t[m] < t'$. This implies that $\mathbf{v}_n^t < \mathbf{v}_m^{t'}$.

2. $(e_n^t \rightsquigarrow e_m^{t'}) \impliedby (\mathbf{v}_n^t < \mathbf{v}_m^{t'})$

   $\mathbf{v}_n^t < \mathbf{v}_m^{t'}$ means that $\mathbf{v}_n^t[i] \leq \mathbf{v}_m^{t'}[i]$, $\forall i \in [N]$. Consider index n, where $\mathbf{v}_n^t[n] \leq \mathbf{v}_m^{t'}[n]$. By Definition 4 2, $\mathbf{v}_n^t[n] = t$, so $\mathbf{v}_m^{t'}[n] \geq t$. Then Theorem 1 states that this implies $e_n^t \rightsquigarrow e_m^{t'}$.

■

Definition 4 is not quite a constructive definition. We need a way to actually compute PVCs. This is enabled by the next theorem.

Theorem 2. The assignment

$$\mathbf{v}_n^t = \begin{cases} [0, \ldots, 0, t, 0, \ldots, 0], & t < \epsilon \\ [t - \epsilon, \ldots, t - \epsilon, t, t - \epsilon, \ldots, t - \epsilon], & t \geq \epsilon \end{cases}$$

where the t is in the n^{th} position in both cases, satisfies the conditions of PVC in Definition 4.

Proof. Consider Definition 7 2. This indicates that all events $e_i^{t-\epsilon}$ happened-before e_n^t, $\forall i \in [N] \setminus \{n\}$. Therefore, if these events directly happened-before e_n^t (there is no $e_m^{t'}$ where $e_i^{t-\epsilon} \rightsquigarrow e_m^{t'}$ and $e_m^{t'} \rightsquigarrow e_n^t$), then this vector is a correct assignment.

By looking at each point in Definition 7, we can see that the only case where one event happened-before another on a different process is when there is at least ϵ difference, Definition 2. While an event may have happened-before e_n^t by indirectly following Definition 2 by way of 2 and 2, we do not need to consider this event because there is not a direct happened-before relation with e_n^t (no event in between). Therefore, the assignment $[t-\epsilon, \ldots, t-\epsilon, t, t-\epsilon, \ldots, t-\epsilon]$ is suitable for timestamp $\mathbf{v}_n^t$. ■

2.4 Signal Model

In this section, we introduce our signal model, i.e., our model of the output signal of an agent. To this end, first, we set some notations. The set of reals is $\mathbb{R}$, the set of non-negative reals is $\mathbb{R}_+$, and the set of positive reals is $\mathbb{R}_+^*$. The set of integers $\{1, \ldots, N\}$ is abbreviated as $[N]$. *Global* time values, kept track of by a *hypothetical* global clock are denoted by χ, χ', etc., while the letters t, t', t_1, t_2, s, s', s_1, s_2, etc. denote corresponding local clock values particular to individual signals/agents, which are always clear from the context.

Definition 5. An *output signal* (of some agent A) is a function $x : [a, b] \to \mathbb{R}^d$, which is right-continuous, left-limited, and is not Zeno. Here, $[a, b]$ is an interval in $\mathbb{R}_+$, and will be referred to as the *timeline* of the signal. ■

Definition 6. A *root* is an event e_n^t where $x_n(t) = 0$ or a discontinuity at which the signal changes sign: $\text{sgn}(x_n(t)) \neq \text{sgn}(\lim_{s \to t^-} x_n(s))$. A *left root* e_n^t is a root preceded by negative values: there exists a positive real δ s.t. $x_n(t-\alpha) < 0$ for all $0 < \alpha \leq \delta$. A *right root* e_n^t is a root followed by negative values: $x_n(t+\alpha) < 0$ for all $0 < \alpha \leq \delta$. ■

We assume that x is one-dimensional, i.e., $d = 1$. Therefore, *Right-continuity* implies that for each t in its support, $\lim_{s \to t_+} x(s) = x(t)$. The function is *Left-limitedness* if it has a finite left-limit at every t in its support: $\lim_{s \to t_-} x(s) < \infty$. *Not being Zeno* means that x

has a finite number of discontinuities in any bounded interval in its support. This prevents the signal from jumping indefinitely many times in a finite length of time. A *discontinuity* in a signal $x(\cdot)$ can be caused by a discrete event within agent A (such as a variable updated by software), or to a message transmitted to or received from another agent A'.

We assume a loosely linked system with N reliable *agents* that never fail, denoted by $\{A_1, \ldots, A_N\}$, without any shared memory or global clock. The output signal of agent A_n is denoted by x_n, for $1 \leq n \leq N$. We refer to some global clock which acts as a '*real*' time-keeper. However, this global clock is a hypothetical object used in definitions and theorems, and is *not* available to the agents. We make two assumptions:

- *(A1) Partial synchrony.* The *local clock* (or time) of an agent A_n can be represented as an increasing function $c_n : \mathbb{R}_+ \to \mathbb{R}_+$, where $c_n(\chi)$ is the value of the local clock at global time χ. Then, for any two agents A_n and A_m, where $m, n \in [N]$, we have:

$$\forall \chi \in \mathbb{R}_+ . |c_n(\chi) - c_m(\chi)| < \varepsilon$$

  where the maximum clock skew presumed fixed and known by the monitor is $\varepsilon > 0$. When we refer to 'local' or 'global' time in the sequel, we make it clear.
- *(A2) Deadlock-freedom.* The agents being analyzed do not enter a deadlock state.

Assumption (A1) is met by using a clock synchronization algorithm, like NTP [88], to ensure bounded clock skew across all agents.

An event in the discrete-time setting is a change in value of an agent's variables. We now update this definition for the continuous-time setting of this work. Specifically, in an agent A_n, an *event* is either a (i) a pair $(t, x_n(t))$, where t is the local time (i.e., returned by function c_n); (ii) a message transmission, or (iii) a message reception. The communications that the agents transmit to each other are free of assumptions. Messages that are sent to the monitor are timestamped by their respective local clocks. Since the agents evolve in continuous time and their output signals are defined for all local times t, a message transmission or reception always coincides with a signal value; i.e., if A_n receives a message at local time t, its signal

has value $x_n(t)$ at that time. Thus, without loss of generality, every event will be represented as a (local time, value) pair $(t, x_n(t))$, often abbreviated as e_t^n (n and t will be omitted when irrelevant).

A *distributed signal* is modeled as a set of signals, where events in each signal are partially ordered by a variation of the *happened-before* ($\rightsquigarrow$) relation [73], extended by our assumption (A1) on bounded clock skew among all agents. The following defines a continuous-time/value *distributed signal* under partial synchrony.

Definition 7. A *distributed signal* on N agents is a pair $(E, \rightsquigarrow)$, where $E = (x_1, \ldots, x_N)$ is a vector of signals, the set I_n is a bounded nonempty interval, and the relation $\rightsquigarrow$ is a relation between events in signals such that:

1. In every signal x_n, all events are totally ordered, that is, for all $n \in [N]$, for any $t, t' \in I_n$, if $t < t'$, then $(t, x_n(t)) \rightsquigarrow (t', x_n(t'))$. That is,

$$\forall n \in [N].\ \forall t, t' \in I_n. \Big(t < t'\Big) \Rightarrow \Big((t, x_n(t)) \rightsquigarrow (t', x_n(t'))\Big),$$

   where the set I_n is a bounded nonempty interval.

2. If the time between any two events is more than the maximum clock skew ε, then the events are totally ordered, that is, for all $m, n \in [N]$, for any $t, t' \in I_n$, if $t + \varepsilon < t'$, then $(t, x_n(t)) \rightsquigarrow (t', x_n(t'))$. That is,

$$\forall m, n \in [N].\ \forall t, t' \in I_n. \Big(t + \varepsilon < t'\Big) \Rightarrow \Big((t, x_m(t)) \rightsquigarrow (t', x_n(t'))\Big).$$

3. If e is a message send event in an agent and f is the corresponding receive event by another agent, then we have $e \rightsquigarrow f$.

4. For any three events e, f, and g, if $e \rightsquigarrow f$ and $f \rightsquigarrow g$, then $e \rightsquigarrow g$. ■

Setting $\varepsilon = \infty$ yields the classic instance of total asynchrony. The constraints on I_n (bounded and non-empty) are required in the continuous-time context and will be discussed more in the next section. Because the agents are synchronized within ε, it is not possible to

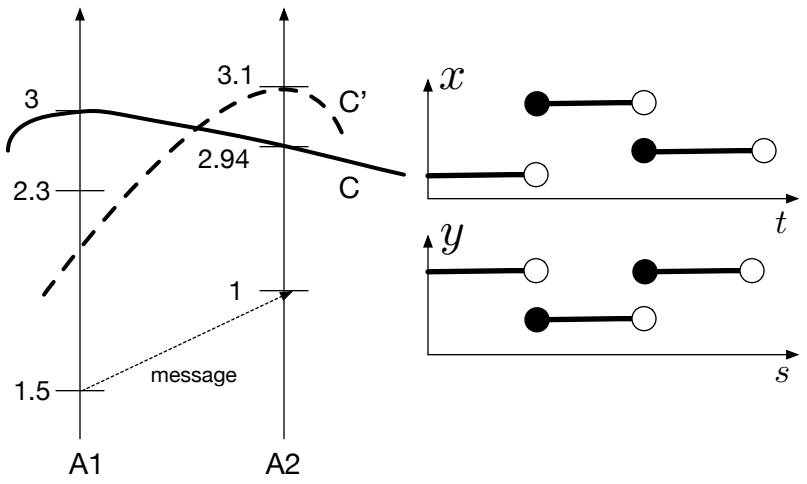


Figure 2.3 Two partially synchronous continuous concurrent timelines with $\varepsilon = 0.5$, and corresponding signals x and y. (Solid dot indicates signal value at discontinuity). C is a consistent cut but C' is not.

analyze all signals in global time simultaneously. The following definition of consistent cut captures plausible global states, that is, states that might be legitimate global states. Figure 2.3 shows two partially synchronous concurrent timelines generated by two agents. Every moment in each timeline corresponds to an event $(t, x_n(t))$, $n \in [2]$. Thus, the following hold: $(1, x_1(1)) \rightsquigarrow (2.3, x_1(2.3))$, $(2.3, x_1(2.3)) \rightsquigarrow (2.94, x_2(2.94))$, $(1, x_2(1)) \rightsquigarrow (2.94, x_2(2.94))$, and $(2.94, x_2(2.94)) \not\rightsquigarrow (3, x_1(3))$.

Definition 8. Let $(E, \rightsquigarrow)$ be a distributed signal over N agents and S be the set of all events defined as follows:

$$S = \Big\{(t, x_n(t)) \mid x_n \in E \wedge t \in I_n \wedge I_n \subseteq \mathbb{R}_+\Big\}.$$

A *consistent cut* C is a subset of S if and only if when C contains an event e, then it contains all events that happened before e. Formally,

$$\forall e, f \in S \;.\; (e \in C) \wedge (f \rightsquigarrow e) \Rightarrow (f \in C). \;\blacksquare$$

From this definition and Definition 7 it follows that if $(t', x_n(t'))$ is in C, then C also contains every event $(t, x_m(t))$ s.t. $t + \varepsilon < t'$. Note that due to time asynchrony, there exists an infinite number of consistent cuts represented by $\mathcal{C}(\chi)$ at any global time $\chi \in \mathbb{R}_+$. This

is due to the fact that there are an infinite number of time instances between any two local time instances t_1 and t_2 on some signal x. As a result, an infinite number of consistent cuts can be created.

A consistent cut C can be represented by its *frontier*

$$\mathsf{front}(C) = \Big\{(t_1, x_1(t_1)), \ldots, (t_N, x_n(t_N))\Big\},$$

in which each $(t_n, x_n(t_n))$, where $1 \leq n \leq N$, is the last event of agent A_n appearing in C. Formally:

$$\forall n \in [N] \;.\; (t_n, x_n(t_n)) \in C \text{ and } t_n = \max\Big\{t \in I_n \mid \exists (t, x_n(t)) \in C\Big\}.$$

Example Assuming $\varepsilon = 0.1$ in Figure 2.3, it comes that all events below (thus, before) the solid arc form a consistent cut C with frontier $\mathsf{front}(C) = \{(3, x_1(3)), (2.94, x_2(2.94))\}$. On the other hand, all events below the dashed arc do *not* form a consistent cut since $(2.3, x_1(2.3)) \rightsquigarrow (3.1, x_2(3.1))$ and $(3.1, x_2(3.1))$ is in the set C', but $(2.3, x_1(2.3))$ is not in C'.

2.5 Signal Temporal Logic (STL)

Let AP be a set of *atomic propositions*. The syntax for *signal temporal logic* (STL) [79] is defined for infinite traces using the following grammar:

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \,\mathcal{U}_{[a,b]}\, \varphi$$

where $p \in \mathsf{AP}$ and $\mathcal{U}$ is the 'until' temporal operator. We view other propositional and temporal operators as abbreviations, that is, $\top = p \vee \neg p$ (`true`), $\bot = \neg\top$ (`false`), $\Diamond_{[a,b]}\, \varphi = \top\, \mathcal{U}_{\,[a,b]} \varphi$ (*eventually or F*), $\Box_{[a,b]}\, \varphi = \neg \Diamond_{[a,b]}\, \neg\varphi$ (*always or G*). We denote the set of all STL formulas by Φ_{STL}.

Let a *trace* $\sigma = (x_1, \ldots, x_N)$ be a vector of N continuous-time and continuous-valued signals. In the context of STL, we express p as $f(x_1[t], \ldots, x_n[t]) > 0$, where $(x_1[t], \ldots, x_n[t]) \in \mathbb{R}^n$ is a vector of signal values at time t, and $f : \mathbb{R}^n \to \mathbb{R}$ is a function that evaluates a vector of signal values.

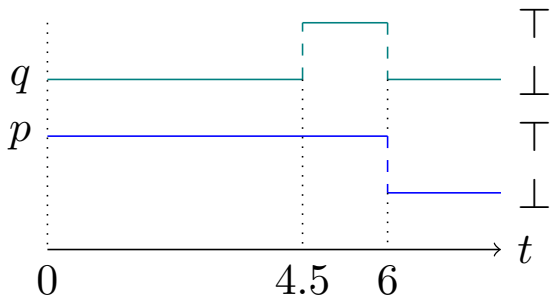


Figure 2.4 A trace σ generated by a system.

The infinite-trace semantics of STL is defined as follows. Let $\models$ be the *satisfaction* relation, and the satisfaction of formula φ by a trace σ at time t be:

$$
\begin{aligned}
(\sigma, t) &\models p && \text{iff} && f(x_1[t], \ldots, x_n[t]) > 0 \\
(\sigma, t) &\models \varphi \wedge \psi && \text{iff} && (\sigma, t) \models \varphi \text{ and } (\sigma, t) \models \psi \\
(\sigma, t) &\models \neg\varphi && \text{iff} && \neg((\sigma, t) \models \varphi) \\
(\sigma, t) &\models \varphi \mathcal{U}_{[a,b]} \psi && \text{iff} && \exists t' \in [t+a, t+b] : (\sigma, t') \models \psi \text{ and } \forall t'' \in [t, t'] : (\sigma, t) \models \varphi
\end{aligned}
$$

For the sake of simplification, from this point and onward, we write $\sigma \models \varphi$ if and only if $(\sigma, 0) \models \varphi$ holds. As an example of STL, given the trace σ shown in Figure 2.4, the STL formula $\varphi = p \mathcal{U}_{[4,6.5]} q$ holds at time 0, that is, $\sigma \models \varphi$. However, φ does not hold after time 2, as in that case, q must hold after time $2 + 4$ and before $2 + 6.5$, which does not happen.

The STL semantics are over infinite signals, however a distributed signal E is defined to have a fixed duration (I_n is bounded), which is suited for online monitoring, but the STL semantics are over infinite signals. Given a (completely synchronous) finite duration signal x, we say it satisfies/violates φ iff every extension $(x.y)$, where y is an infinite signal, satisfies/violates φ. Otherwise, Unknown is returned by the monitor. The dot '.' here represents time concatenation.

2.6 Producer-Consumer Network

A *producer-consumer network* is a directed acyclic graph (DAG) $G = (\mathbb{V}, \mathbb{E})$, in which each vertex $v \in \mathbb{V}$ is a *node*), that may be either a producer, a consumer, or both, based on its incoming/outgoing edges. A *producer* node only has outgoing edges, a *consumer* node

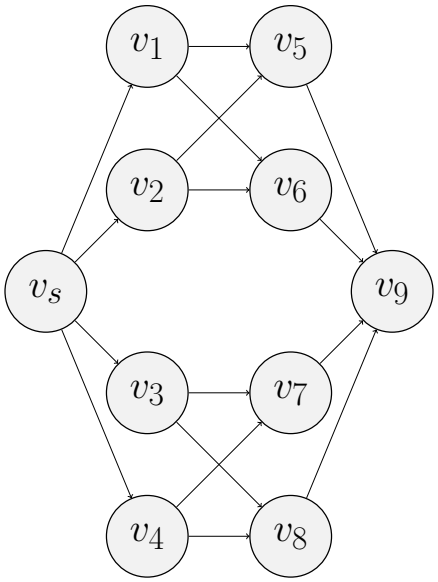


Figure 2.5 A producer-consumer network of 10 nodes.

only has incoming edges, and a producer/consumer node has both incoming and outgoing edges. Let $\mathsf{Pred}(v)$ denote the finite set of predecessor nodes from which v receives data, and $\mathsf{Succ}(v)$ denote the finite set of successor nodes which receive data from v. The set $\mathbb{E}$ of edges represented as ordered pairs of vertices such that:

$$\mathbb{E} = \Big\{(u, v) \mid v \in \mathsf{Succ}(u)\Big\}.$$

An edge from u to v represents a stream of items flowing from u to v, in which case u is a producer (potentially also a consumer) and v is a consumer. A node $v \in \mathbb{V}$, where $\mathsf{Pred}(v) = \emptyset$ is called a *source* and a node $u \in \mathbb{V}$, where $\mathsf{Succ}(u) = \emptyset$ is called a *sink*.

Figure 2.5 depicts a producer-consumer network. The network represents a hierarchical monitoring system, in which $v_{[1,4]}$ are producers of events that are consumed and manipulated by nodes $v_{[5,8]}$. Nodes $v_{[5,8]}$ then transmit the manipulated events into v_9.

A producer (respectively, consumer) node $v \in \mathbb{V}$ may receive (respectively, emit) data at a set of *possible input rates* denoted by *possible output rates* $\mathsf{IRate}(v)$ (respectively, ORate_v). Let $\mathsf{Out}(u, v)$ denote the outgoing data rate from node u into node v. For example, in Figure 2.5, the incoming data for v_1 is received from v_s, and the outgoing data is sent to v_5

and v_6. For every node $v \in \mathbb{V}$, we define $\mathsf{In}(v)$ such that,

$$\mathsf{In}(v) = \sum_{u \in \mathsf{Pred}(v)} \mathsf{Out}(u, v).$$

CHAPTER 3

RUNTIME VERIFICATION OF PARTIALLY SYNCHRONOUS DISTRIBUTED DISCRETE-EVENT SYSTEMS

In this chapter, we present two sound and complete solutions to the distributed runtime verification (RV) problem in relation to LTL formulas. In order to address the explosion of different interleaving, we adopt a practical assumption, namely, a finite skew between local clocks of each pair of processes, which is ensured by a fault-proof clock synchronization system, such as NTP [88]. Both approaches utlize a fault-proof central monitor.

To this end, we consider *discrete-event* systems [20], where the discrete states in the said systems are transitioned via events. The events can be message send events, message receive events or local processing events. As stated in Chapter 1, the agents in these systems do not share a global clock and memory, while attempting to perform a joint task. However, a clock synchronization algorithm (see Subsection 2.3) guarantees a *maximum clock skew* among the agents; thus, allowing partial synchrony. In other words, we make the following assumptions:

- The systems under observation are discrete-event systems. That is, for every agent, within any time period, there is a finite number of event executions. These events could be internal to agents (e.g. variable updates), a message send event, or a message receive event.
- A *bounded skew* ε between local clocks of every pair of processes, guaranteed by a fault-proof clock synchronization algorithm (e.g., NTP). This means time instants from different local clocks within ε are considered concurrent, i.e., it is not possible to determine their order of occurrence. This setting constitutes *partial synchrony*, which does not assume a global clock but limits the impact of asynchrony within clock drifts.

In the following sections, we elaborate on our runtime verification approach for partially synchronous distributed systems using an automata-based technique and a progression-based formula rewriting technique.

3.1 Problem Statement

Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$, as defined in Definition 2, and an LTL formula φ, we say $(\mathcal{E}, \rightsquigarrow)$ satisfies φ iff there exists a trace, α, defined by a sequence of frontiers in $(\mathcal{E}, \rightsquigarrow)$, that satisfies φ. Formally, the evaluation of the LTL formula φ with respect to $(\mathcal{E}, \rightsquigarrow)$ in the finite semantics is the following:

Problem Statement

Monitoring of Distributed Systems. 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 $i \geq 0$, we have (1) $C_i \subset C_{i+1}$, and (2) $|C_i| + 1 = |C_{i+1}|$. 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:

$$\big\{\mathsf{front}(C_0)\mathsf{front}(C_1)\cdots \mid C_0C_1C_2\cdots \in \mathcal{C}\big\}.$$

The evaluation of the LTL formula φ with respect to $(\mathcal{E}, \rightsquigarrow)$ in the finite semantics is the following:

$$[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] = \Big\{\alpha \models_3 \varphi \mid \alpha \in \big\{\mathsf{front}(C_0)\mathsf{front}(C_1)\cdots \mid C_0C_1C_2\cdots \in \mathcal{C}\big\}\Big\}$$

and,

$$[(\mathcal{E}, \rightsquigarrow) \models_F \varphi] = \Big\{\alpha \models_F \varphi \mid \alpha \in \big\{\mathsf{front}(C_0)\mathsf{front}(C_1)\cdots \mid C_0C_1C_2\cdots \in \mathcal{C}\big\}\Big\}$$

This means that evaluating a distributed computation against a formula yields a *set* of verdicts, because a computation may contain multiple traces. It should be noted that throughout this chapter, $(\mathcal{E}, \rightsquigarrow)$ is used to denote distributed computation.

3.2 Formula Progression for LTL

Because of the existence of a total ordering of events in a synchronous system, verification on a computation may be accomplished in a state by state method [10]. However, in a

partially synchronous system, such event ordering is not possible. A distributed computation $(\mathcal{E}, \rightsquigarrow)$ may have different event orderings governed by different event interleavings. As a result, multiple verdicts might be obtained from the same distributed computation $(\mathcal{E}, \rightsquigarrow)$. To explore these verdicts, we present a formula progression-based monitoring approach that, if possible, partially evaluates a formula on the current computation and, depending on the verdict, provides a rewritten formula 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*.

Definition 9. A *progression function* $\mathsf{Pr} : \Sigma^* \times \Phi_{\mathsf{LTL}} \rightarrow \Phi_{\mathsf{LTL}}$ is one that for all finite traces $\alpha \in \Sigma^*$, infinite traces $\sigma \in \Sigma^\omega$, and formulas $\varphi \in \Phi_{\mathsf{LTL}}$, we have: $\alpha\sigma \models \varphi$ iff and only if $\sigma \models \mathsf{Pr}(\alpha, \varphi)$. ■

Our method and the traditional rewriting method [59] vary primarily in that our function Pr accepts finite traces as input, whereas the algorithm in [59] rewrites the input LTL formula in a state-by-state manner. As a result, it is not feasible to rewrite using the fixed point representation of temporal operators. The fact that a given distributed computation is divided into a number of segments so an SMT query is used to verify each segment serves as the motivation for our method. A state-by-state approach would generate excessive amounts of SMT queries, rendering the approach inefficient and 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 $\varphi' \in \Phi_{\mathsf{LTL}}$, 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`:

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

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:

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

Next operator. Let $\varphi = \bigcirc\phi$. The verdicts `true`, `false` and ϕ' can only be reached if $\alpha^1 \neq \varepsilon$. Otherwise, or 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:

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

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

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

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

Note that the semantics of FLTL is not frequently used, due to LTL_3 being generally more expressive, as shown in [11]. However, LTL_3 cannot be used to construct the progression rules. To be more precise, the '?' (*unknown*) verdict in 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. In fact, if progression results in a formula, it represents the '?' verdict in LTL_3. Therefore, we use FLTL for specifying the progression rules without any loss of generality as shown later in the proof of Lemma 2.

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 independently, and the verdicts of each are used to establish 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 reach a verdict without taking the order of occurrence of these sub-formulas into account. That is, on a given finite trace α, if φ_2 holds in $\alpha(i)$ (denoted $\Diamond_i \varphi_2$) and φ_1 holds throughout in all states from $\alpha(0)$ to $\alpha(i-1)$ (denoted $\Box_{i-1} \varphi_1$), then the progression of φ becomes `true`. If this is not the case, and $\Box \varphi_1$ 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. The progression of φ remains unchanged if φ_1 holds throughout α, but φ_2 does not hold anywhere:

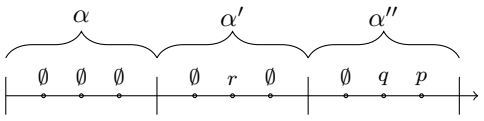


Figure 3.1 Progression example.

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

Example. Consider the formula $\varphi = \Diamond r \rightarrow (\neg p \,\mathcal{U}\, q)$, which can be broken into sub-formulas $\varphi_s = \{\Diamond r,\ q,\ \Diamond q,\ \Box p\}$, according to our progression rules. Consider the trace in Figure 3.1 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 $\Diamond r$ 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., $\mathsf{Pr}(\alpha'', \neg p \,\mathcal{U}\, q) = \texttt{true}$. Put it another way, $\mathsf{Pr}(\alpha\alpha'\alpha'', \varphi) = \texttt{true}$.

Lemma 2. Given an LTL formula φ, and a finite and infinite trace $\alpha \in \Sigma^*, \sigma \in \Sigma^\omega$ respectively, 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, $\varphi = p$.

($\Rightarrow$) 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:

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

Case 5: Now, we consider the case where the formula is $\varphi = \varphi_1 \mathcal{U} \varphi_2$. We prove this by induction:

Base Case: $\mid \alpha \mid = 0$.

$$\begin{aligned}
[\alpha\sigma \models_F \varphi] &\iff [\sigma \models_F \mathrm{Pr}(\alpha, \varphi)] \\
&\iff [\sigma \models_F \varphi]
\end{aligned}$$

Hypothesis Step: $|\ \alpha\ | = k$.

$$
\begin{aligned}
&[\alpha\sigma \models_F \varphi_1 \mathcal{U} \varphi_2] \\
&\iff [\alpha\sigma \models_F \Big(\varphi_2 \vee \big(\varphi_1 \wedge \bigcirc(\varphi_1 \mathcal{U} \varphi_2)\big)\Big)] \\
&\iff [\alpha\sigma \models_F \varphi_2] \vee [\alpha\sigma \models_F \Big(\varphi_1 \wedge \bigcirc(\varphi_1 \mathcal{U} \varphi_2)\Big)] \\
&\iff [\alpha\sigma \models_F \varphi_2] \vee \Big([\alpha\sigma \models_F \varphi_1] \wedge [\alpha^1\sigma \models_F \varphi_1 \mathcal{U} \varphi_2\Big)] \\
&\iff [\alpha\sigma \models_F \varphi_2] \vee \Bigg([\alpha\sigma \models_F \varphi_1] \wedge [\alpha^1\sigma \models_F \Big(\varphi_2 \vee \big(\varphi_1 \wedge \bigcirc(\varphi_1 \mathcal{U} \varphi_2)\big)\Big)]\Bigg) \\
&\iff [\alpha\sigma \models_F \varphi_2] \vee \big([\alpha\sigma \models_F \varphi_1] \wedge [\alpha^1\sigma \models_F \varphi_2]\big) \vee \ldots \vee \Big([\alpha\sigma \models_F \varphi_1] \wedge \\
&\quad [\alpha^1\sigma \models_F \varphi_1] \wedge \ldots \wedge [\alpha^{k-2}\sigma \models_F \varphi_1] \wedge [(\alpha^{k-1}\sigma \models_F \varphi_2]\Big) \vee \\
&\quad \Big([\alpha\sigma \models_F \varphi_1] \wedge \ldots \wedge [\alpha^{k-1}\sigma \models_F \varphi_1] \wedge [\alpha^k\sigma \models_F \varphi_1 \mathcal{U} \varphi_2]\Big) \\
&\iff [\alpha\sigma \models_F \varphi_2] \vee \Big([\alpha\sigma \models_F \varphi_1] \wedge [\alpha^1\sigma \models_F \varphi_2]\Big) \vee \ldots \vee \Big([\alpha\sigma \models_F \varphi_1] \wedge \\
&\quad \ldots \wedge [\alpha^{k-1}\sigma \models_F \varphi_1] \wedge [\sigma \models_F \varphi_1 \mathcal{U} \varphi_2]\Big)
\end{aligned}
$$

Inductive Step: $|\ \alpha\ | = k + 1$ Trivially expanded from the above expansion.

$$
\begin{aligned}
[\alpha\sigma \models_F \varphi_1 \mathcal{U} \varphi_2] \iff [\alpha\sigma \models_F \varphi_2] \vee \Big([\alpha\sigma \models_F \varphi_1] \wedge [\alpha^1\sigma \models_F \varphi_2]\Big) \vee \ldots \vee \\
\Big([\alpha\sigma \models_F \varphi_1] \wedge \ldots \wedge [\alpha^{k-1}\sigma \models_F \varphi_1] \wedge [\alpha^k\sigma \models_F \varphi_1] \wedge [\sigma \models_F \varphi_1 \mathcal{U} \varphi_2]\Big)
\end{aligned}
$$

Now, in order for $[\alpha\sigma \models_F \varphi_1 \mathcal{U} \varphi_2]$ to yield `true`, there must be a $k \geq 1$ such that $[\alpha\sigma \models_F \varphi_1 \wedge \ldots \wedge \alpha^{k-1}\sigma \models_F \varphi_1 \wedge \alpha^k\sigma \models_F \varphi_2]$, that is,

$$
\begin{aligned}
[\alpha\sigma \models_F \varphi_1 \mathcal{U} \varphi_2] &\iff [\exists k \geq 1 \,.\, \alpha^0\sigma \models_F \varphi_1 \wedge \ldots \wedge \alpha^{k-1}\sigma \models_F \varphi_1 \wedge \\
&\qquad \alpha^k\sigma \models_F \varphi_2] \\
&\iff [\exists k \geq 1 \,.\, \alpha\sigma \models_F \Diamond_k \varphi_2 \wedge \alpha\sigma \models_F \Box_{k-1} \varphi_1]
\end{aligned}
$$

Note that the above recursive definition of Until allows us to evaluate any until formula, and by extension, any always ($\Box\varphi = \varphi \mathcal{U} \bot$) and eventually ($\Diamond\varphi = \top \mathcal{U} \varphi$) formula. Therefore, we can evaluate any sub-formula using this fixed point representation of until.

■

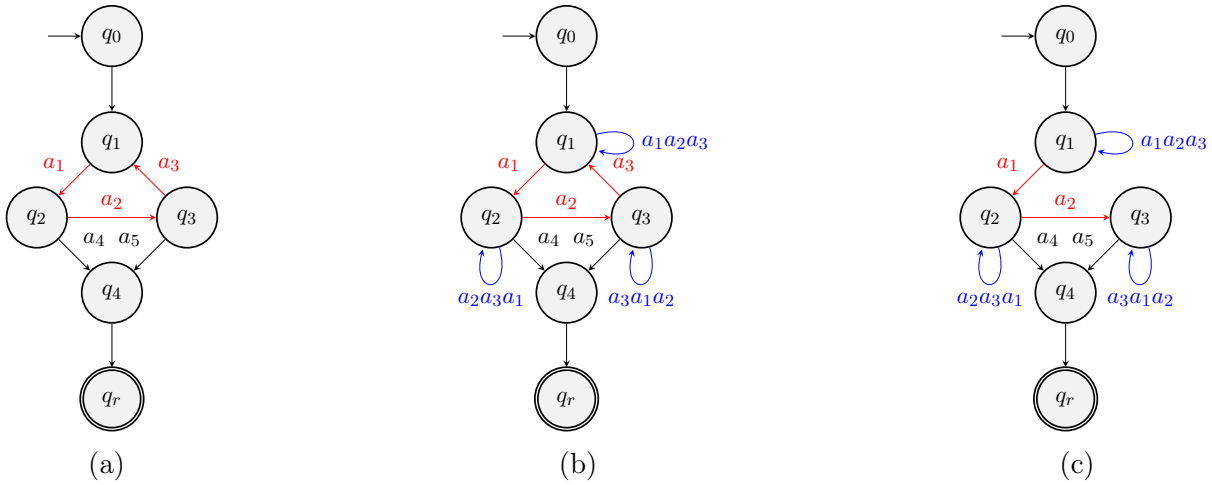


Figure 3.2 Removing non-loop cycles in an LTL_3 Monitor.

3.3 SMT-based Solution

In this section, we go into further detail about our approach to distributed monitoring utilizing the two previously discussed monitoring techniques: (1) automata-based approach, and (2) progression-based approach.

3.3.1 Overall Idea

Automata-based approach. Recall from Figure 1.5 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 1). 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 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 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 Figure 2.1, we consider two paths $q_0^* q_\perp$ 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 LTL_3 monitors may contain non-self-loop cycles. In order to simplify the SMT instance creation process (for each possible path in the LTL_3 monitor), we collapse each

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

Algorithm 3.1 Non-Self Loop Cycle Removal Algorithm

non-self-loop cycle into one state with a self-loop labeled by the sequence of events in the cycle using Algorithm 3.1. As an example, in Figure 3.2, Algorithm 3.1 first takes an LTL_3 monitor (Figure 3.2a) and adds the necessary self-loops (Figure 3.2b). Then it eliminates all non-self-loop cycles by removing transitions from states with higher identifiers to states with lower identifiers in cycles (Figure 3.2c). The non-deterministic nature of the final automata ensure that all the transitions and the accepting language of the automata are preserved.

Lemma 3. 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 3.1 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: 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 $\alpha = a_1 a_2 \cdots a_n$, where $\forall i \in [1, n].a_i \in \Sigma$. Algorithm 3.1 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: 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 $\alpha = a_1 a_1 \cdots a_n$, where $\forall i \in [1, n].a_i \in \Sigma$. A self-loop in $\mathcal{M}'_\varphi$ can be represented by $\exists i \in [1, n], \exists k \in [1, n - i] \ . \ \delta'(q, a_i a_{i+1} \cdots a_{i+k}) = q$. 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 $i \in [1, m]$.

■

Progression-based approach. Due to the existence of a total ordering of events in a synchronous system, verification on a computation may be carried out using a state-by-state methodology [10]. A partially synchronous system, however, makes such an ordering of events impossible. Varying interleavings of events can lead to different orderings of events in a distributed computation $(\mathcal{E}, \rightsquigarrow)$. Therefore, it is possible to obtain multiple verdicts on the same distributed computation $(\mathcal{E}, \rightsquigarrow)$. To explore these verdicts, we provide a formula progression monitoring approach that, if feasible, partially evaluates a formula on the current computation and, in response to the verdict, offers 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 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 $\texttt{hbSet}[i][j] = 1$ if $\mathcal{E}[i] \rightsquigarrow \mathcal{E}[j]$, and $\texttt{hbSet}[i][j] = 0$ otherwise. We introduce a function $\mu : \mathcal{E} \times \mathsf{AP} \rightarrow \{\texttt{true}, \texttt{false}\}$ 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 Subsection 3.3.3.

3.3.3 SMT Constraints

We next go on to the SMT constraints after defining the requisite SMT entities. The SMT constraints for consistent cuts that are enforced on both the automata-based and the

progression-based approaches are first defined. 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.

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

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

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

We determine if a series of consistent cuts conforms to the specification after it has been created. This is done using (1) progression-based approach, where the LTL formula is represented by a SMT constrain and (2) 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 LTL_3 automata respectively as discussed below.

Let C represent for the conjunction of the aforementioned constraints. Recall that there is only one valid path that is relevant to this conjunction C. Since there may be multiple paths in the monitor, 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 above satisfiable, then a valid path exists.

Constraints for LTL progression over ρ. Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$, the aforementioned constraints may provide a valid series of consistent cuts that may result in multiple verdicts depending on how the concurrent events are ordered. Therefore, while evaluating an LTL formula on $(\mathcal{E}, \rightsquigarrow)$, all potential outcomes are investigated in order to prevent false positives. To achieve this, we examine the sequence of consistent cuts $C_0C_1C_2 \cdots C_m$ interpreted by the uninterpreted function $\rho(m)$, looking for both satisfaction and violation. Note that applying our progression rules to monitor any LTL formula will cause it to eventually monitor sub-formulas that only include atomic propositions, globally, and eventually temporal operators:

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

Situations to the contrary will lead to a rewritten formula that will go on to the following segment. In general, the verdict for any LTL formula will be derived using our progression rules in Section 3.2.

3.4 Optimization

We employ several optimization techniques in our implementation to speed up and improve the monitoring process. In this section, we discuss two crucial optimization techniques, as well as their impact on run time.

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 [53]. 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:

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

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 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 automata-based approach. That is, Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$ of length l divided into $\frac{l}{g}$ segments of length g, the evaluation of the LTL formula φ on, by the automata-based approach is equal, i.e.,

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

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, $\{\alpha \models_3 \varphi \mid \alpha \in \mathrm{Tr}(\mathcal{E}, \rightsquigarrow)\} \neq \{\alpha \models_3 \varphi \mid \alpha \in \mathrm{Tr}(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow)\}$

($\Rightarrow$) Let C_k be a consistent cut such that C_k is in $\mathrm{Tr}(\mathcal{E}, \rightsquigarrow)$, but not in $\mathrm{Tr}(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow)$ 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 $\{\alpha \models_3 \varphi \mid \alpha \in \mathrm{Tr}(\mathcal{E}, \rightsquigarrow)\} \subseteq \{\alpha \models_3 \varphi \mid \alpha \in \mathrm{Tr}(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow)\}$. 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 $\mathrm{Tr}(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow)$, but not in $\mathrm{Tr}(\mathcal{E}, \rightsquigarrow)$ 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 $\{\alpha \models_3 \varphi \mid \alpha \in \text{Tr}(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow)\} \subseteq \{\alpha \models_3 \varphi \mid \alpha \in \text{Tr}(\mathcal{E}, \rightsquigarrow)\}$. 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 5. 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,

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

3.4.2 Parallelized Monitoring

Clusters of computers with several processing cores and processors are used by many cloud services. They can now create high-performance parallel/distributed applications and handle huge data rates as a result. Utilizing the extensive infrastructure should also be possible for monitoring such applications. In light of this, we will now talk about parallelizing our SMT-based monitoring technique.

Let G be a sequence of g segments $G = seg_1 seg_2 \cdots seg_g$. For each computer core that is available, a task queue will be established.The segments will then be distributed evenly among all of the queues so that each core may independently monitor its queue. However, merely dividing up all the segments across cores will not guarantee a reliable outcome. 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. The monitor operating on Cr_2 must be aware of the outcome of the monitor operating on Cr_1 in order to render the proper verdict. 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 $\neg a$. 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 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 Figure 3.3 for the monitor shown in Figure 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 $RM[i][j][\top] = \texttt{true}$, then $RM[i'][\top][\top] = \texttt{true}$ for all $i' \in [i, |Q| - 1]$. Likewise, if $RM[i][j][\bot] = \texttt{true}$, then $RM[i'][\bot][\bot] = \texttt{true}$ 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 $RM[i][k][j] = \texttt{true}$, 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, Figure3.4 shows the verdict reachability tree generated from the matrix in Figure 3.3.

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 6. A distributed computation $(\mathcal{E}, \rightsquigarrow)$ of length l satisfies an LTL formula φ if and

	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 3.3 Reachability Matrix for $a \mathcal{U} b$.

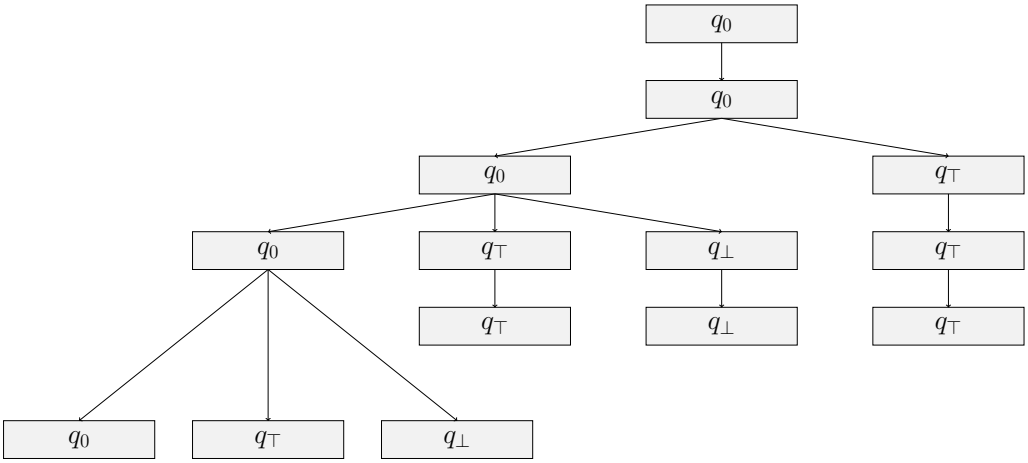


Figure 3.4 Reachability Tree for $a \mathcal{U} b$.

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 focus on our SMT-based solution without digressing into other aspects like instrumentation, data gathering, 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) [19, 72], and the RACE dataset from NASA [85].

3.5.1 Implementation and Experimental Setup

Three steps may be identified for each experiment: (1) data generation, (2) data collection, and (3) data verification. For the purpose of generating data, we create a synthetic program that at random generates a distributed computation (i.e., the behaviors of a set of programs in terms of their inter-process communication and local calculations). 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 3.5b.

For data verification, we consider the following parameters: (1) number of processes (N), (2) computation duration (l secs), (3) segment length (g), (4) event rate (r events/process/sec), (5) maximum clock skew (ϵ), and (6) number of nested temporal operators ($|\phi|$) for the LTL formula under monitoring. The primary *metric* is to calculate the SMT solving runtime for each parameter configuration. In all of the charts shown in this section, the time axis is displayed in log scale. By keeping the values of all the other parameters at sensible fixed values, we can study the impact of changing one parameter. 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 [97] 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 [97].

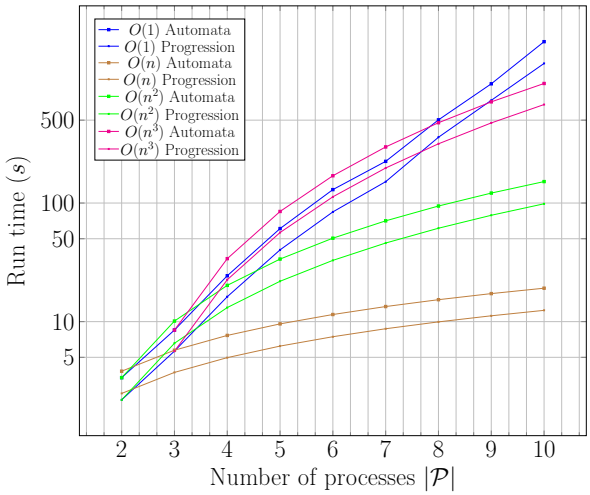
3.5.2 Analysis of Results – Synthetic Experiments

In this series of experiments, we examine every parameter that is available and record how it impacts SMT solution. To investigate how each parameter affects runtime, we test each one separately. Since the created synthetic data is independent of any outside influences, we include a delay to both reduce the amount of events occurring at each time unit and to ensure that events are distributed equally across the execution of each process. We assign a value to each local computation event in each process using a uniform distribution $(0, |\Sigma|)$. The findings of the following experiments only make use of one CPU core.

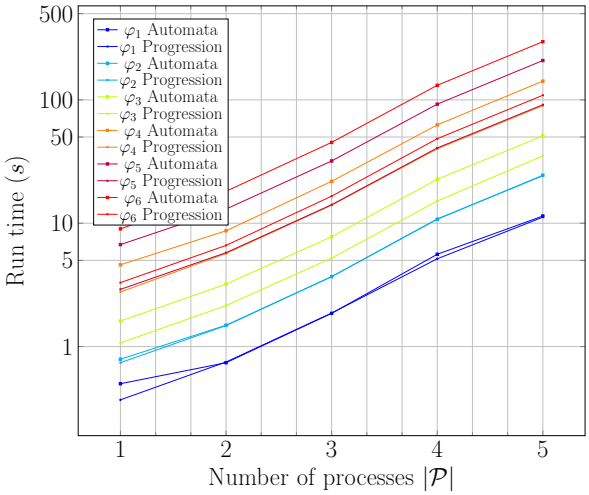
Overall, we notice an improvement of around 35% when the progression based technique is compared to the other automata based approach. This improvement in performance owes to two main reasons: (1) compared to the automata-based approach, the LTL constrains in our progression-based approach is less demanding 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 3.5a), 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.

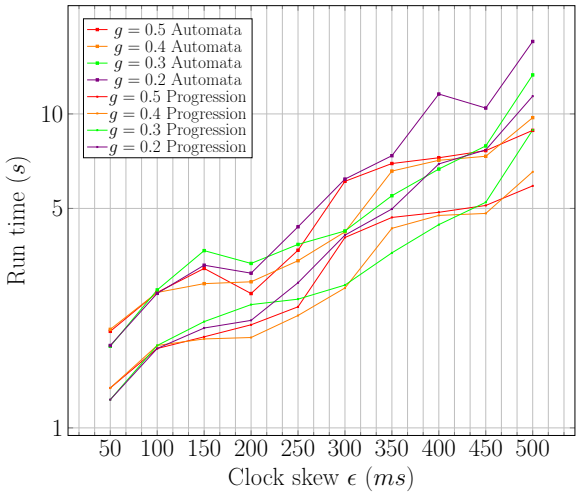
Having said that, during our experiments when comparing the runtime of our monitoring approach for increasing number of sub-formulas, we observe a slight decrease in the overall efficiency in runtime when using the progression-based approach compared to the automata-based approach. Since the progression-based approach is based on evaluating each



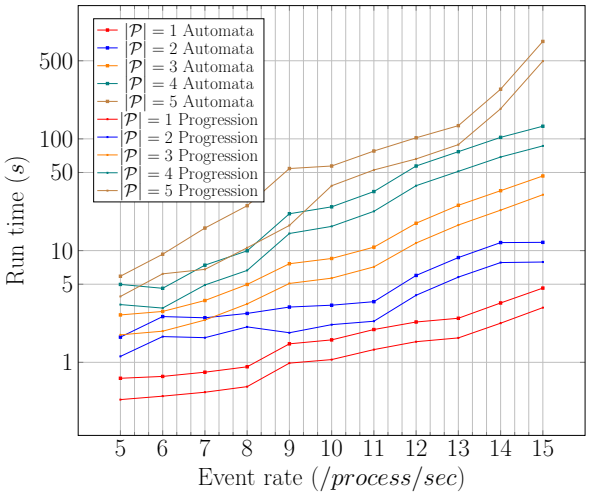
(a) Predicate Structure.



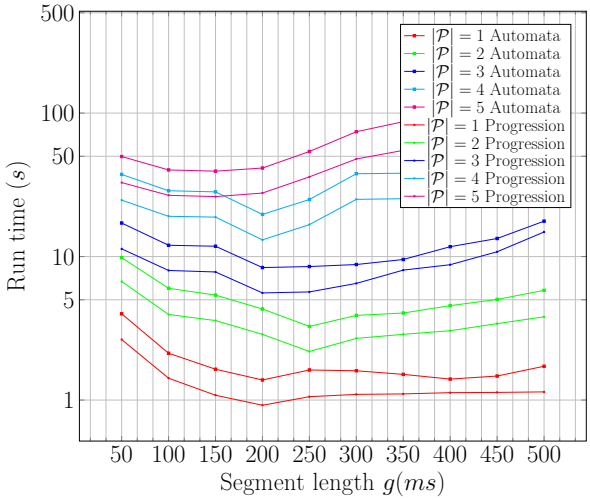
(b) LTL Formula.



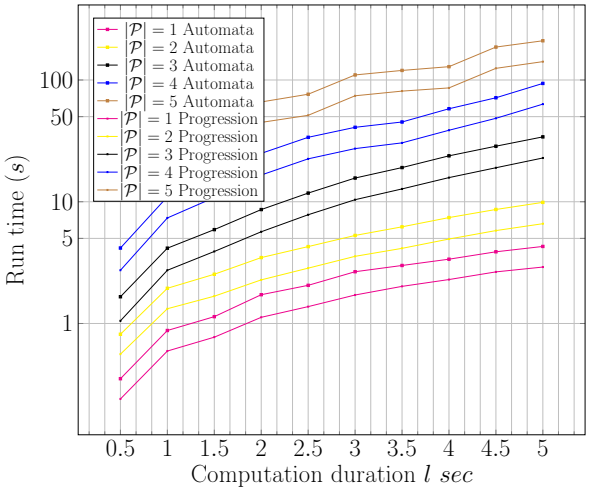
(c) Epsilon.



(d) Event Rate.



(e) Segment Length.



(f) Computation Duration.

Figure 3.5 Synthetic experiments - impact of different parameters.

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 Figure 3.5b. We experiment with the following LTL formula and the progression based technique achieved an average improvement of 32.8% compared to the automta-based one.

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

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

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

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

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

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

Impact of partial synchrony. Figure 3.5c depicts the anticipated outcome, wherein an exponential rise in the number of concurrent events across processes leads to longer runtime as clock skew ϵ grows. When comparing with the automata-based approach , the progression-based technique yields us an improvement of 33.36%.

Impact of event rate. Figure. 3.5d shows that our approach breaks even with the computation duration for $N = 3$ for an event rate of $5 events/process/sec$. However, increasing

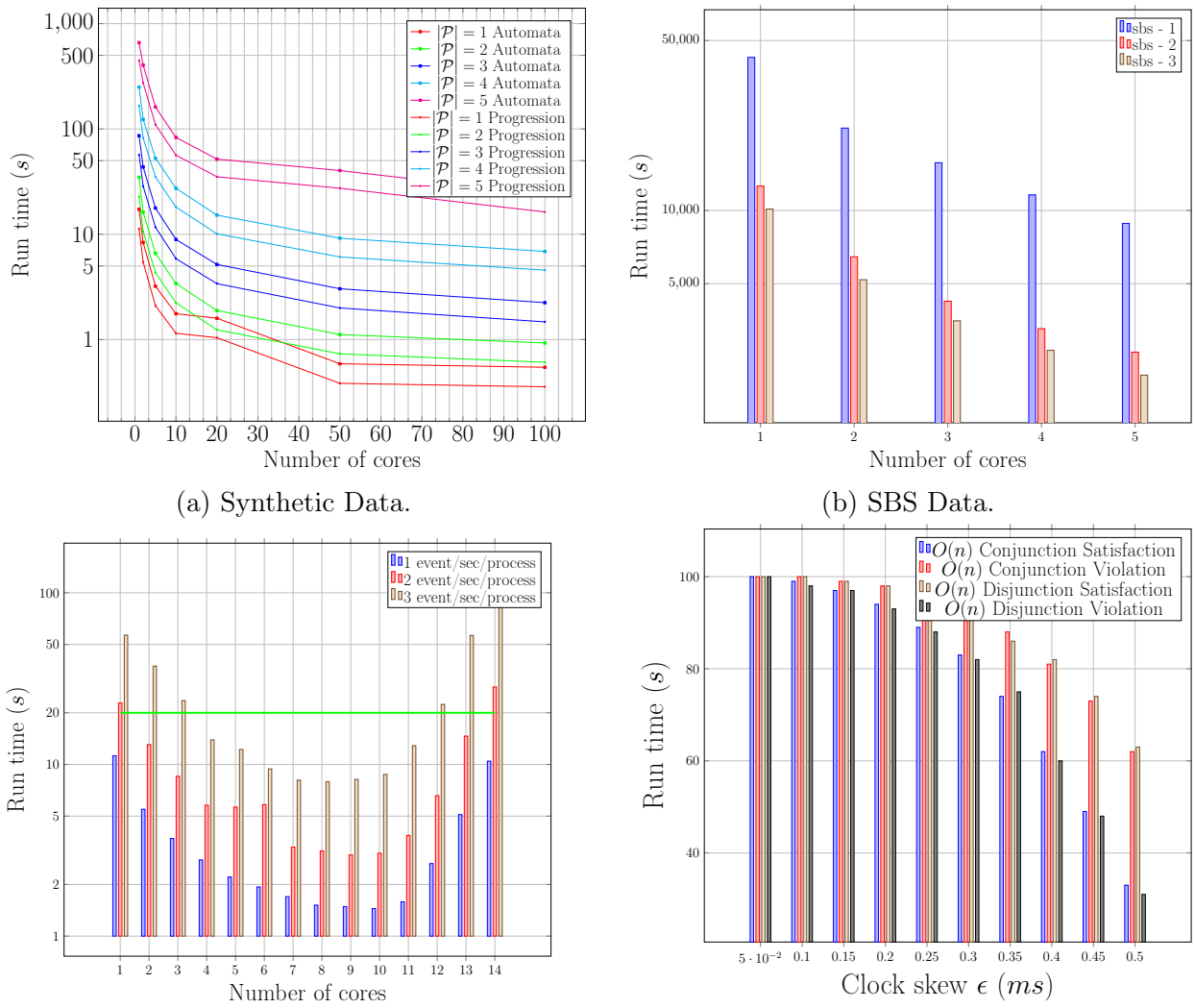
the event rate increases the search space for the SMT solver. Overall we improve by 34.4% by using the progression-based technique compared to the automata-based technique.

Impact of segment count. The number of events to be handled grows as segment length rises, exponentially lengthening the time our method takes to operate. Since there are not enough occurrences to have an effect, $N = 1, 2$ doesn't show significant improvement in Figure 3.5e. For a greater number of operations, we see improved performance with shorter segments. Due to the time required to construct a greater number of SMT encodings outweighing the performance benefit from smaller segments, it should be noted that the runtime rises for extremely short segment lengths. Here too, we notice an improvement of 32.6% for the progression-based technique over the automata-based technique.

Impact of computation duration. In Figure 3.5f, we lengthen computation and monitor the impact on runtime. The number of segments required to verify the lengthier computation grows as the duration of the computation rises, leading to a linear increase in runtime. The progression-based approach improves the runtime by 33.1% when compared to the automata-based approach.

Impact of parallelization. The technique performs significantly better when the verification is distributed over many cores. Figure 3.6a illustrates the dramatic improvement in performance that occurs when the number of cores is increased from 1 to 10. However, raising it further makes little progress since the time required to generate the SMT encodings begins to take precedence over the time required to solve it. An improvement of 33.8% is achieved for progression-based approach when compared to automata-based approach.

Impact of ϵ on false warnings. As discussed in Section 2.3, 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



(a) Synthetic Data.

(b) SBS Data.

(c) Google Data.

(d) False Warnings.

Figure 3.6 Impact of parallelization on different data.

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 3.6d, 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.

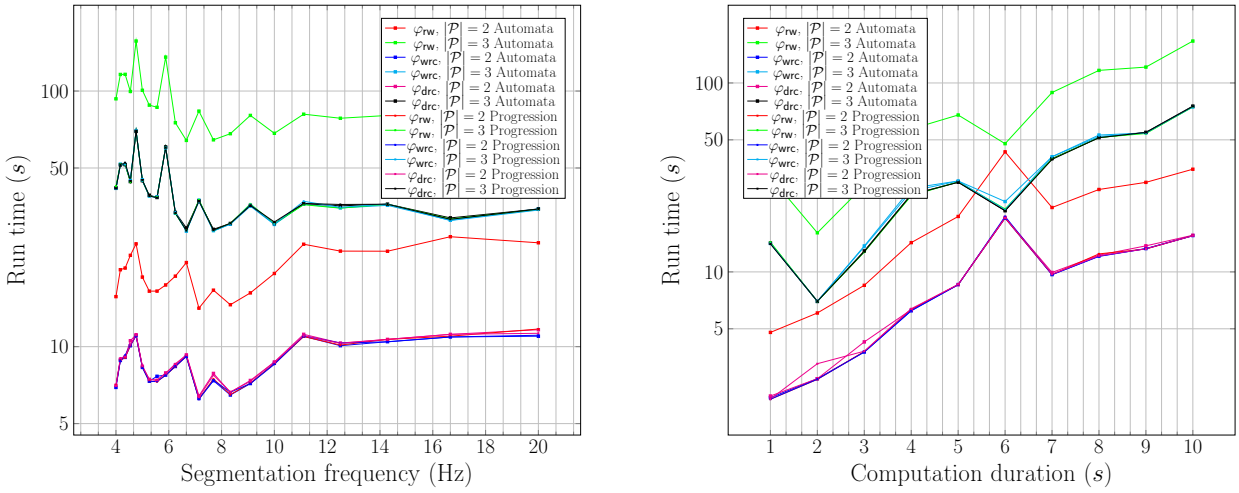
Additionally, we observe that the number of false warning is greatly influenced by the predicate structure of the LTL formula, as evident from Figure 3.6d. For $O(n)$ conjunctive satisfaction formula monitoring and $O(n)$ disjunctive violation formula monitoring, false warnings might occur if any one 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

In this case study, we observe read/write irregularities of a No-SQL distributed database management system called Cassandra [19, 72]. One node from each cluster serves as the seed node in our simulation of a distributed database with two data centers: one cluster with four nodes and the other with three. Each node in both clusters replicates all of the data. Each node runs on *Red Hat OpenStack Platform* using 4 VCPUs, 4GB RAM, Ubuntu 18.04, Cassandra 3.11.6, and Java 1.8.0_252. Additionally, we have simulated a system with numerous processes, each of which is in charge of the fundamental database operations (read, write and update). These processes are also capable of inter-process communication, which enables them to alert other processes in the event that they create a new database record.

We compared our system's latency against that of Google Cloud, Microsoft Azure, and Amazon Web Services in order to make our simulated database as realistic as possible. The quickest response was timed at $41ms$ compared to our system's $100ms$. The sluggish bandwidth and different infrastructure are to blame for the significant latency when compared to the industry norm. In all of our experiments, we consider a delay of $100ms$ into account.

Each of the processes is capable of reading, writing, or updating the database entries given the way the processes are designed. We choose the kind of operation that will be carried out by the process using a $(0, 2)$ uniform distribution. The other processes are informed



(a) Segment Length. (b) Computation Duration.

Figure 3.7 Cassandra experiments.

of any additions made by the write operation using the inter-process communications. No messages are believed to have been lost during transmission, and as soon as they are received, the receiving process reads each message.

Consistency level helps a database maintain the bare minimum number of replications required for an activity to be deemed successfully completed. In order to eliminate any potential of a read or write anomaly in the database, Cassandra recommends that the total of read and write consistency should be greater than the replication factor. Using runtime monitoring approaches, we want to keep an eye on and detect read/write irregularities in the database. 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 drawbacks of utilizing a distributed database like Cassandra is the absence of database normalization features. As a result, we intend to monitor both write and delete reference checks. We present two tables:

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

On these tables, we enforce the write and remove reference check. A write in the Enroll-

MENT table must always be followed by a write in the STUDENT table with the same *id*. Similarly, deleting from the STUDENT table should always be followed by deleting from the ENROLLMENT table with the same *id*. This ensures no insertion and deletion anomalies, resulting in the following LTL specification:

$$\varphi_{wrc} = \neg\Big(\neg write(\text{STUDENT}.id)\,\mathcal{U}\, write(\text{ENROLLMENT}.id)\Big)$$

$$\varphi_{drc} = \neg\Big(\neg delete(\text{ENROLLMENT}.id)\,\mathcal{U}\, delete(\text{STUDENT}.id)\Big)$$

Extreme load scenario. Fig 3.7b and Fig 3.7a depict runtime versus computation duration and runtime vs segmentation frequency under our network's maximum read/write load. These results are slightly noisier when compared to the results of the synthetic experiments. This is because the events in the synthetic experiments were uniformly distributed across the whole computation length, but they are not uniform here. Database operations requiring network communications (read, write, and update) require an average of $100ms$, whereas sending and receiving messages involve inter-process communication and take roughly $10ms$-$15ms$, resulting in a non-uniform event distribution. 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 Fig 3.7b, we were able to break even with as little as 2 processes. To find a real-world example with modest database activities, consider the Google Sheets API, which enables a maximum of 500 requests per $100s$ per project and a maximum of 100 requests per second per user, i.e., on average 5 events/sec per project and 1 event/sec per user. To see how our technique operates in such a scenario, we increase the number of processes and cores available to monitor such a system in order to investigate the time required to verify the trace created by such a system. In Figure 3.6c, we see that we break even at an event rate of 3 events/sec/user when using the progression-based strategy. Our algorithm operates effectively when the number of processes is 7, 8, or 9, which is far higher

than Google allows. This allows us to be certain that our technique can be implemented online in real-world scenarios.

3.5.4 Case Study 2: RACE

In this case study, we monitor a mutual separation property between multiple aircrafts. The dataset[1] for this case study was generated using the *Runtime for Airspace Concept Evaluation* (RACE) [85] framework developed by NASA. RACE is a framework for creating an event-based, reactive airspace simulation. This dataset consists of three data sets obtained on three distinct days. Each pair was captured at around 37° N Latitude and 121° W Longitude. The dataset contains all eight types of messages sent by the SBS unit when a Telnet application is used 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 and is used to verify the mutual separation of all pairs of aircraft.

We found that the time gap between the time the message was created and the time it was recorded was generally less than a second, thus we regarded an $\epsilon = 1s$ over the time the message was generated. Furthermore, calculating the distance between two locations is computationally intensive since we must account for characteristics such as earth curvature. We consider a constant latitude of $111.2km$ and longitude of $87.62km$ to speed up distance computations at the expense of a minuscule 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 assuming a minimum separation of $500m$ between each pair of aircrafts. According to the dataset, each aircraft generates a message at least once per 1 second. There are three distinct datasets: sbs-1 has 293 aircrafts, $168,283$ messages spread over 3 hours, 28 minutes, and 58 seconds; sbs-2 has 110 aircrafts, $64,218$ messages spread over 1 hour, 1 minute, and 46 seconds; and sbs-3 has 97 aircraft, $64,162$ messages spread over 49 minutes and 42 seconds.

In Figure 3.6b, we compare our obtained runtime to the three RACE datasets (labelled

[1] https://github.com/NASARace/race-data

sbs-1, sbs-2 and sbs-3). We monitor the data in real time, with segments of $10s$ and ϵ of $1s$. We put our approach to the test by increasing the number of cores on the CPU and utilizing all available cores, as described 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 Conclusion

We elected to start our work with discrete-event systems (as opposed to continuous-time systems) due to the fact that monitoring discrete-event systems are intuitively less expensive in terms of runtime and computational complexity, compared to similar continuous-time systems. Both of our proposed techniques take an LTL formula and a distributed computation as input and, assuming a bounded clock skew among all processes, chops the computation into multiple segments before applying either the automata-based monitoring algorithm, or the progression-based monitoring algorithm implemented as an SMT decision problem to verify the formula's correctness. In Section 3.5, we carried out extensive simulated 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. Furthermore, based on these experiments, we summarize that online monitoring is indeed possible with our techniques when distributed computations are properly segmented and parallelized. A natural course of action now would be to carry and apply the relevant aspects of this approach into monitoring continuous-valued systems; in other words, distributed CPS. We take the first steps into monitoring distributed CPS in the next chapter.

CHAPTER 4

PREDICATE MONITORING IN DISTRIBUTED CYBER-PHYSICAL SYSTEMS

In this chapter, we take first steps towards rigorously monitoring distributed CPS. To this end, we propose a monitoring technique to detect *Boolean predicates* over the *analog* (i.e. continuous-time and continuous-valued) signals generated by the agents in a distributed CPS. Similar to our approach described in Chapter 3, a clock synchronization algorithm guarantees a maximum clock skew across all signals generated by the agents.

In the following sections, we first define the analog signal transmission sampling method based on our signal model defined in Chapter 2. We then elaborate on our predicate detection approach for partially synchronous distributed CPS using a signal retiming technique.

4.1 Signal Transmission to the Monitor

Communication between nodes requires sampling the analog signal, sending the samples, and reconstructing the signal at the receiving node. Our goal is to monitor the reconstructed analog signals. This is not the same as monitoring a discrete-time signal composed of samples; the applications we are addressing are concerned with the value of the signal between samples and the possible violations revealed by it. Signal transmission methods, such as sampling and reconstruction, are common in communication theory. Errors caused by sampling and reconstruction (for example, owing to bandwidth constraints) can be addressed for by tightening the STL formula using the methods of [45]. The reconstruction algorithm is chosen based on the application and domain knowledge. For the sake of simplicity and generality, we assume that every output signal x_n is rebuilt as piece-wise linear between the samples, except in one experiment where we reconstruct a signal as both *piece-wise linear* and *piece-wise quadratic* to study the trade-offs. Other signal constructions, such as cubic splines, can also be employed with easy modifications to our algorithms at the cost of increased run time, provided that the signal structure chosen is orthogonal to our methodologies and the aims of this work. Since we assume the agents do not deadlock, this transmission happens in

segments of length T: at the k^{th} transmission, agent A_n transmits $x_n|_{[(k-1)T,kT]}$, the restriction of its output signal to the interval $[(k-1)T, kT]$ as measured by its local clock. The remainder of the work refers solely to the signal fragments received by the monitor during a specific transmission.

We now return to the constraint imposed on I_n in Definition 7, namely that it is a non-empty bounded interval. Non-emptiness models the absence of deadlock in computing. That is an interval I_n expresses that no events are missed, or equivalently, that signal reconstruction is perfect at the monitor. The restriction that it be bounded models the above monitoring setup: the monitor is only ever dealing with bounded signal fragments $x_n|_{[(k-1)T,kT]}$, therefore,

$$I_n = [(k-1)T, kT], \tag{4.1}$$

for every agent at the k^{th} transmission, *measured in local time*. By the bounded skew assumption, we have:

Lemma 7. For any two agents A_n, A_m, $|\min I_n - \min I_m| \leq \varepsilon$ and $|\max I_n - \max I_m| \leq \varepsilon$.

4.2 Problem Statement

Predicates are frequently used to encapsulate several system requirements (e.g., invariants). A *predicate* φ is a global Boolean-valued function over the signal values of agents. For instance, $\varphi(x_1, x_2) = (x_1 > 0) \wedge (\ln(x_2) < 3)$ is a predicate on two signals that evaluates to `true` when $x_1 > 0$ and $\ln(x_2) < 3$, otherwise `false`.

Because the agents are partially synchronized to within an ε, it is not possible to actually evaluate all signals at the same moment in global time. However, as noted above, the frontier of a consistent cut gives us a possible global state. Therefore, the monitoring problem can be worded as follows. Given a distributed signal $(E, \rightsquigarrow)$ over N agents, as defined in Definition 7, and a Boolean predicate φ, $(E, \rightsquigarrow)$ satisfies φ iff there exists a frontier of a consistent cut in $(E, \rightsquigarrow)$, where φ is satisfied. It should be noted that throughout this chapter, $(E, \rightsquigarrow)$ is used to denote distributed signals. We now define distributed satisfaction below.

Definition 10. [Distributed satisfaction] Given a distributed signal $(E, \rightsquigarrow)$ over N agents, and a predicate φ over the N agents, we say that $(E, \rightsquigarrow)$ *satisfies* φ iff for all consistent cuts $C \subseteq E$ with

$$\mathsf{front}(C) = \Big((t_1, x_1(t_1)), \ldots, (t_N, x_N(t_N)) \Big)$$

we have $\varphi\big(x_1(t_1), x_2(t_2), \ldots, x_N(t_N)\big) = \texttt{true}$. We write this as $(E, \rightsquigarrow) \models \varphi$. ■

Thus, we formally define the problem as follows.

Problem Statement

Continuous-Time Monitoring of Distributed CPS. Given a distributed signal $(E, \rightsquigarrow)$ and a predicate φ over N agents, determine whether $(E, \rightsquigarrow) \models \varphi$.

When a distributed signal $(E, \rightsquigarrow)$ does not satisfy a predicate φ, we say that $(E, \rightsquigarrow)$ *violates* φ and write $(E, \rightsquigarrow) \not\models \varphi$. In this dissertation, we want to detect whether there exists a consistent cut $C \subseteq E$, such that $(E, \rightsquigarrow) \not\models \varphi$.

The main challenge in monitoring distributed signals is that the monitor has to reason about signals that are subject to time asynchrony. For instance, consider two signals x_1 and x_2 and the case where $x_1(2) = 5$, $x_2(3) = 1$, $\varphi(x_1, x_2) = (x_1 > 4) \wedge (x_2 < 0)$, and $\varepsilon = 2$ so that time points 2 and 3 form a consistent cut. In this case, since the above signal values are at local times within the possible clock skew, one has to (conservatively) consider that the predicate is violated. In the next section, we present our solution to the problem.

4.3 SMT-based Monitoring Algorithm

In a nutshell, our solution has the following features:

- **Central monitor.** We assume that there is a *central* monitor that solves, at regular intervals, the monitoring problem described in Section 4.2.
- **Signal retiming.** As signals are measured using their local clocks, the monitor should somehow align them to detect possible violations of the predicate. To this end, we propose a *retiming* technique that establishes the happened-before relation in

the continuous-time setting, and stretches or compacts signals to align them with each other within the ε clock skew bound.

- **SMT encoding.** We transform the monitoring decision problem into an SMT-solving problem, whose components (like input signals and the happened-before relation) are modeled as SMT entities and constraints.

4.3.1 Retiming Functions

Our signal model is continuous-time, that is, the signals are maps from $\mathbb{R}_+$ to $\mathbb{R}_+$. Therefore, to model the approximate re-synchronizing action of the monitor, we use a *retiming function*.

Definition 11. [Retiming functions] A *retiming* function, or simply retiming, is an increasing function $\rho : \mathbb{R}_+ \to \mathbb{R}_+$. An ε-retiming is a retiming such that: $\forall t \in \mathbb{R}_+ : |t - \rho(t)| < \varepsilon$. Given a distributed signal $(E, \rightsquigarrow)$ over N agents and any two distinct agents A_i, A_j, where $i, j \in [N]$, a retiming ρ from A_j to A_i *respects* $\rightsquigarrow$ if we have $((t, x_i(t)) \rightsquigarrow (t', x_j(t'))) \;\Rightarrow\; (t < \rho(t'))$ for any two events $(t, x_i(t)), (t', x_j(t')) \in E$. An ε-retiming that respects $\rightsquigarrow$ is a *valid* retiming. ■

Figure 4.1 shows examples of retimings and how they relate to predicate monitoring. To detect predicate violation, we must first retime y to the t axis via a retiming map ρ. (c) shows three different retimings, including the identity. (d)-(e) show the retimed y. For the predicate $x > y$, (e)-(f) show no violations, but (d) does. The conservative monitoring answer is that the predicate is violated. An ε-retiming ρ maps $\mathbb{R}_+$ to itself, but it is easy to see that the restriction of ρ to a bounded interval I is an increasing function from I to $\rho(I)$ that respects the constraint $|t - \rho(t)| < \varepsilon$ for all $t \in I$. Thus, in what follows we restrict our attention to the action of ε-retimings on bounded intervals.

We now state and prove the main technical result of this chapter, which relates the existence of consistent cuts in distributed signals to the existence of retimings between the agents' local clocks.

Theorem 3. Given a predicate φ and distributed signals $(E, \rightsquigarrow)$ over N agents, there exists a consistent cut $C \subseteq E$ that violates φ *if and only if* there exists a finite A_1-local clock value

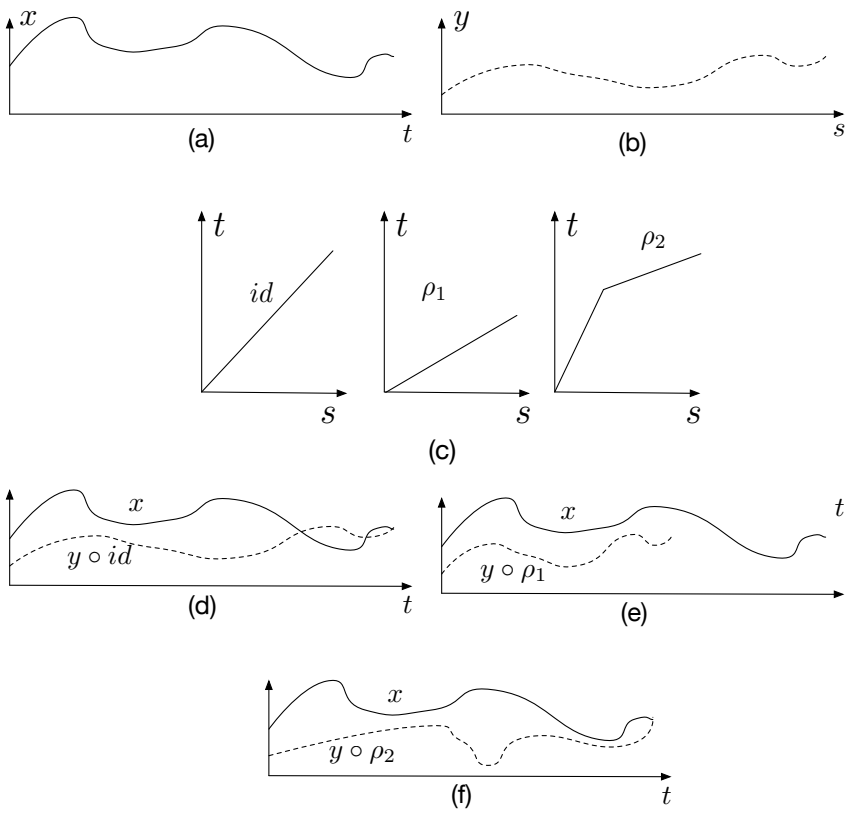


Figure 4.1 Predicate violation between two signals x and y measured using partially synchronized clocks t and s.

t and $N-1$ ε-retimings $\rho_n : I_n \to I_1$ that respect $\rightsquigarrow$, $2 \leq n \leq N$, such that:

$$\varphi\Big(x_1(t), x_2 \circ \rho_2^{-1}(t), \ldots, x_N \circ \rho_N^{-1}(t)\Big) = \texttt{false} \tag{4.2}$$

and such that $\rho_m^{-1} \circ \rho_n : I_n \to I_n$ is an ε-retiming for all $n \neq m$. Here, '$\circ$' denotes the function composition operator.

Proof. We distinguish the following cases:

Case 1: Suppose that such retimings exist. Define the local time values $t_1 := t$, $t_n = \rho_n^{-1}(t)$, $2 \leq n \leq N$, and the set $C = \{e_t^n t \leq t_n\}$. By the construction of C and the fact that the retimings respect $\rightsquigarrow$, it holds that if $e \in C$ and $f \rightsquigarrow e$ then $f \in C$. For every $n, m \geq 2, n \neq m$, it holds that $t_m = \rho_m^{-1}(\rho_n(t_n))$ so $|t_n - t_m| \leq \varepsilon$. Thus C is a consistent cut with frontier $(e_{t_n}^n)_{n=1}^N$ that witnesses the violation of φ.

Case 2: Suppose now that there exists a consistent cut C with frontier:

$$\mathsf{front}(C) = \Big((t_1, x_1(t_1)), \ldots, (t_N, x_N(t_N)) \Big)$$

that witnesses violation of φ. We need the following facts.

Fact 1. For every two events $e^n_{t_n}$ and $e^m_{t_m}$ in the frontier of a consistent cut, we have $|t_n - t_m| \leq \varepsilon$. Indeed, since $e^n_{t_n} \in \mathsf{front}(C)$, we have $e^m_s \in C$ for all s s.t. $s + \varepsilon \leq t_n$. Thus, $t_m \geq s$ for all such s and so $t_m \geq t_n - \varepsilon$. By symmetry of the argument, $t_n \geq t_m - \varepsilon$ holds as well.

Fact 2. Given intervals $[a, b]$ and $[c, d]$ s.t. $|a-c| \leq \varepsilon$ and $|b-d| \leq \varepsilon$, the map $L : [a, b] \to [c, d]$ defined by $L(t) = c + \frac{d-c}{b-a}(t - a)$ is a linear ε-retiming. This is immediate.

Suppose first that there are no message exchanges. For $2 \leq n \leq N$, we define the retiming $\rho_n : I_n \to I_1$ in two pieces. First, set $\rho_n(t_n) = t_1$. By preceding lemma, $|t_n - t_1| \leq \varepsilon$. Write $I_1 = [a, b]$ and $I_n = [c, d]$ for notational simplicity in this proof. Call a pair of intervals that satisfies the hypothesis of Fact 2 an *admissible pair*. Then, the following pairs are clearly admissible by Lemma 7: $[a, t_1]$ and $[c, t_n]$, and $[t_1, b]$ and $[t_n, d]$. Thus, there exist two linear retimings $L_n : [a, t_1] \to [c, t_n]$ and $L'_n : [t_1, b] \to [t_n, d]$, and we can define a piece-wise ρ_n: $\rho_n(t) = L_n(t)$ on $c \leq t \leq t_n$ and $\rho_n(t) = L'_n(t)$ on $t_n \leq t \leq d$. It is easy to establish that ρ_n is an ε-retiming.

It remains to show that $\rho_n^{-1} \circ \rho_m : I_m = [f, g] \to [c, d]$ is also an ε-retiming. This too can be established in parts, first over $[f, t_m]$ then over $[t_m, g]$, using the same arguments as above and exploiting the linearity of these retimings. For instance, if we write α_n for the slope of L_n, then over $[f, t_m]$

$$\begin{aligned} \rho_n^{-1}(\rho_m(s)) &= L_n^{-1}(L_m(s)) = L_n^{-1}(a + \alpha_m(s - c)) \\ &= \frac{1}{\alpha_n}[a + \alpha_m(s - c)] + f - a/\alpha_n = f + \frac{g - f}{d - c}(s - c) \end{aligned}$$

which is a linear ε-retiming by Fact 2. ■

If there are message exchanges, the above argument still applies but over a more fine-grained division of the timelines I_n obtained by partitioning each timeline at message transmission times.

Proof. For the admissible pair $I_1 = [a, b]$ and $I_n = [c, d]$, suppose the first message is sent from A_n to A_1 at local time $s < t_n$ and is received at local time $r < t_1$. Define $t_{(s)} := \min(s+\varepsilon, r)$. Then the pair $[a, t_{(s)}]$, $[c, s]$ is admissible. Upon repeating this process for all messages, a collection of admissible pairs is obtained that can be retimed to each other, as above, without violating the $\rightsquigarrow$ relation. These are concatenated to yield the desired retiming ρ_n.

■

Thus, finding a consistent cut that violates the predicate can be achieved by finding such retimings. The proof of Prop. 3 further shows that the retimings can always be chosen as piece-wise linear (rather than any increasing function), which yields significant runtime savings in the SMT encoding in the next section.

Remark 2. An interesting consequence of Fact 2 in the proof is that it is enough to use piece-wise linear retimings. This results in the following concrete problem.

Concrete Problem Statement

Given $\varepsilon > 0$, a distributed signal $(E, \rightsquigarrow)$ over N agents, and a predicate φ over the N agents, find $N-1$ piece-wise linear ε-retiming functions $\rho_2, \ldots, \rho_N$ that satisfy the hypotheses of Theorem 3 and s.t.

$$\varphi\Big(x_1(t_1), x_2(\rho_2^{-1}(t_1)), \ldots, x_N(\rho_N^{-1}(t_1))\Big) = \texttt{false} \tag{4.3}$$

4.3.2 SMT Formulation

We solve the monitoring problem by transforming it into an instance of *satisfiability modulo theory* (SMT) [6]. Specifically, we ask whether there exists $N-1$ retimings, such that (4.3) holds; equivalently, whether there exists a consistent cut that witnesses satisfaction of $\neg\varphi$.

Without loss of generality, we start with our encoding of two agents, A_1 and A_2 (shown in Figure 2.3). A_1 outputs signal x supported over the bounded timeline I_1, which is discretized to $D_1 \subset I_1$ and sent to the monitor. Similarly, A_2 outputs signal y supported over the bounded timeline I_2, which is discretized to $D_2 \subset I_2$ and sent to the monitor. D_1 and D_2 are finite. Let $\delta_k > 0$ be the *sampling period* of agent A_k, so two consecutive elements of D_k differ by δ_k, $k \in \{1, 2\}$.

Consider further that A_2 transmits a message at local time t_1 and it is received by A_1 at local time t_2, and that A_1 sends a message at local time t_3 which is received by A_2 at local time t_4. The distributed signal violates the predicate iff the following SMT problem returns SAT.

SMT entities. In our encoding, the entities are the retimings ρ_n included as *uninterpreted* functions (the solver will interpret), signals x and y, intervals I_1 and I_2, real numbers t, s, s', t_1, t_2, t_3, and t_4. All these entities have been defined in the previous sections. The following quantities are all constants in the encoding, since they are known to the monitor: the sampling time sets D_k and sampling periods δ_k, the sampled values $\{x(t_i) \mid t_i \in D_1\}$ and $\{y(s_i) \mid s_i \in D_2\}$, and the message transmission and reception local times.

SMT constraints. The encoding is a conjunction of the following constraints:

- *(Predicate violation)* The first constraint 'finds' local times t and s at which predicate φ is violated (upto ε-synchrony):

$$\exists \quad t \in I_1. \exists s \in I_2. \tag{4.4a}$$

$$\Big(\exists t^- \in D_1.\ t^- \leq t \leq t^- + \delta_1\Big) \ \wedge \tag{4.4b}$$

$$\Big(\exists s^- \in D_2\ .\ s^- \leq s \leq s^- + \delta_2\Big) \ \wedge \tag{4.4c}$$

$$\Big(\rho(s) = t\Big) \ \wedge \tag{4.4d}$$

$$\Big(\neg\varphi(x(t^-), y(s^-))\Big) \tag{4.4e}$$

  Eq. (4.4b) finds the time sample t^- such that $x(t) = x(t^-)$: this is the result of our assumption that signals are piece-wise constant. Eq.(4.4c) does the same for y.

Eq. (4.4d) specifies that s is retimed to t: this is what guarantees that $(x(t), y(s))$ is a possible global state as per Theorem 3. Eq. (4.4e) checks violation of the predicate at $(x(t), y(s)) = (x(t^-), y(s^-))$.

- *(Valid retiming)* Eq. (4.5) ensures that ρ is a valid ε-retiming from I_2 to I_1:

$$\forall s \in I_2.\ \exists t \in I_1.\ (\rho(s) = t) \wedge (|t - s| < \varepsilon) \tag{4.5}$$

  and Eq. (4.6) ensures that the retiming function is increasing:

$$\forall s \in I_2.\ \forall s' \in I_2.\ \Big(s < s' \ \Rightarrow\ \rho(s) < \rho(s')\Big) \tag{4.6}$$

- *(Happened-before)* Eq. (4.7) enforces the happened-before relation for message transmissions:

$$\Big(\rho(t_1) < t_2\Big) \ \wedge\ \Big(t_3 < \rho(t_4)\Big) \tag{4.7}$$

- *(Inverse retiming)* When there are more than 2 agents, we must also encode the constraint that for all $n \neq m$, $\rho_m^{-1} \circ \rho_n$ is an ε-retiming. Thus, for all $n \neq m$, letting f_m be the uninterpreted function that represents the inverse of the uninterpreted ρ_m, we add

$$\forall t \in I_n \cdot\ f_m(\rho_n(t)) = t \tag{4.8}$$

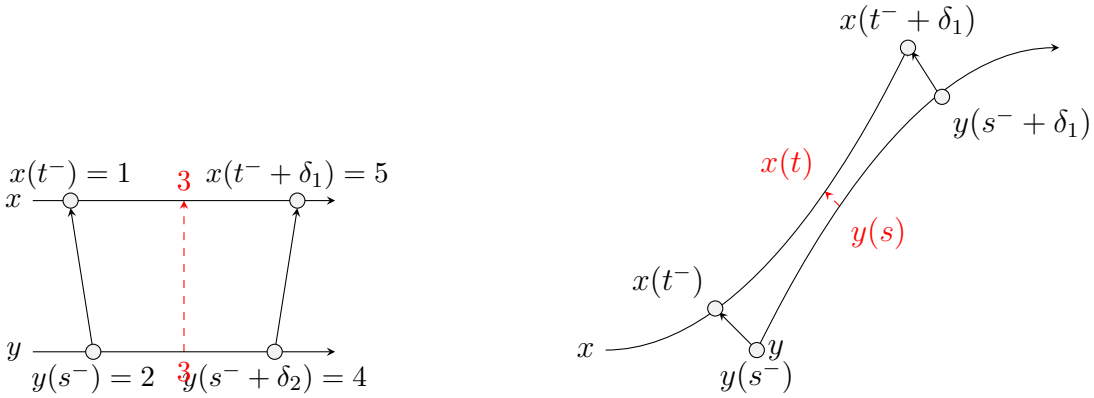
  in addition to the analogs of Eqs. (4.6) and (4.5) for $f_m \circ \rho_n$.

Other signal models. If output signals were piece-wise linear, say, Eq. (4.4e) would be modified accordingly:

$$\varphi\left(x(t^-) + \frac{x(t^- + \delta_1) - x(t^-)}{\delta_1}(t - t^-),\right. \tag{4.9}$$
$$\left. y(s^-) + \frac{y(s^- + \delta_2) - y(s^-)}{\delta_2}(s - s^-)\right) = \texttt{false}$$

Similarly, if output signals were piece-wise quadratic, Eq. (4.4e) would be modified as follows:

$$\varphi\left(x(t), y(s)\right) = \texttt{false} \tag{4.10}$$



(a) Piece-wise linear signals.

(b) Piece-wise quadratic signals.

Figure 4.2 Piece-wise interpolations.

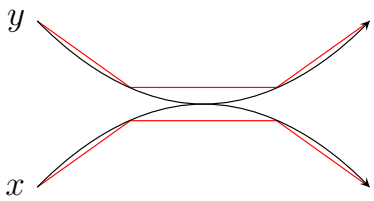


Figure 4.3 Piece-wise linear signals vs. piece-wise quadratic signals.

In some systems, piece-wise quadratic signals may be used to represent signals more accurately. For example, Figure 4.3 shows two piece-wise quadratic construction having the same value at some point in time, whereas their piece-wise linear counterpart signals do not. Our choice of signal models is limited by the SMT solver: it must be able to handle the corresponding interpolation equations, like the piece-wise linear interpolation in Eq. (4.9). As an example, in Figure 4.2a, let x and y be two signals, where the violating predicate ϕ to be monitored is $x(t) = y(s)$. Let ρ be a retiming of y on x, such that $\rho(s^-) = t^-$ and $\rho(s^- + \delta_2) = t^- + \delta_1$. It can be observed that although the discretized signal samples do not violate ϕ, due to the signals being piece-wise linear, it is easy to identify a violation at time t and s on signals x and y respectively, where $x(t) = 3$, $y(s) = 3$ and $\rho(s) = t$.

Another example is demonstrated in Figure 4.2b, where x and y are two signals expressed by their corresponding quadratic formulas. The violating predicate ϕ to be monitored is $d(x(t), y(s)) \leq 2$, where d is a function that yields the distance between any two points. Let ρ be a retiming of y on x, such that $\rho(s^-) = t^-$ and $\rho(s^- + \delta_2) = t^- + \delta_1$. Furthermore, let

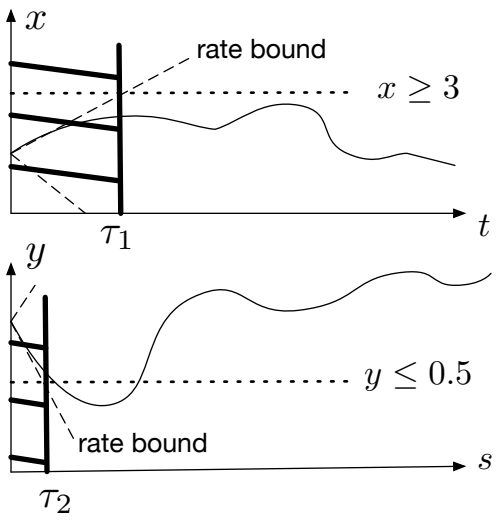


Figure 4.4 Leveraging dynamics.

evaluation of $d(x(t^-), y(s^-))$ be 3 and evaluation of $d(x(t^- + \delta_1), y(s^- + \delta_2))$ be 3. It can be observed that although the discretized signal samples do not violate ϕ, due to the signals being piece-wise quadratic, it is easy to identify a violation at time t and s on signals x and y respectively, where $d(x(t), y(s)) \leq 2$ and $\rho(s) = t$.

It is worth mentioning that restricting the SMT search to piece-wise linear retimings results in a significant decrease in run time, compared to the approach where the SMT is tasked with determining an interpolation. For example, for two UAVs with $\varepsilon = 1ms$ over $5s$-long signals, at segment count 5, the search for a general retiming requires $3.42s$, whereas searching for a piece-wise linear retiming requires only $1.01s$. Since, by Remark 2, there is no loss of generality in this restriction, from this point, all the reported experiments are obtained using the piece-wise linear retiming approach.

Remark 3. (i) $\rho_m^{-1} \circ \rho_n$ respects $\rightsquigarrow$ automatically so it is not necessary to encode that explicitly. (ii) Because we can restrict the SMT search to piece-wise linear retimings (see remark following proof of Theorem 3), constraint (4.8) can be simplified, namely, the expression for the inverse can be hard-coded. We do not show this to maintain clarity of exposition.

Data: Distributed signal $(E, \rightsquigarrow)$, ε, predicate φ, bounds $|\dot{x}_n| \leq b_n, n \in [N]$
Result: $(E, \rightsquigarrow) \models \varphi$
Set $t_n = \min I_n, n \in [N]$ **while** *not done* **do**
  Get next violating assignment σ to the atoms of φ **if** *there are no more violating assignments* **then**
    done
  else
    for *every atom* a *in* φ **do**
      if $\sigma(a) =$ `true` **then**
        $\tau_n = \min\{\tau \mid x(t_n + \tau) \geq v_a\}, n \in [N]$
      else
        $\tau_n = \min\{\tau \mid x(t_n + \tau) < v_a\}, n \in [N]$
    end
    Set $\tau = \max_n \tau_n$ and $m = \text{argmax}_n \tau_n$ SMT-monitor the distributed signal E_σ made of the restrictions $x_n|_{[t_n+\tau-\varepsilon, \max I_n]}$, $n \neq m$ and $x_m|_{[t_m+\tau, \max I_m]}$ If `SAT`, done.
  end
end

Algorithm 4.1 Dynamics-aware monitoring.

4.4 Exploiting the Knowledge of System Dynamics

Physical processes in a CPS follow the laws of physics. A runtime monitor can leverage this knowledge of the CPS dynamics to make monitoring more efficient. We explain our idea by the following example (see Figure 4.4). From knowing the rate bound $|\dot{x}| \leq 1$ (shown by a dashed line), the monitor concludes that the earliest x can satisfy the atom $x \leq 3$ is τ_1. Similarly for y. Given that $\tau_1 > \tau_2$, the monitor discards, roughly speaking, the fragment $[0, \tau_2]$ from each signal and monitors the remaining pieces. Note that $x(0) = 1$ and $y(0) = 2$. Consider the predicate: $\varphi = \neg(a \vee b)$, where $a := x \geq 3$ and $b := y \leq 0.5$. Let a and b be *atoms* of predicate φ. There are 3 Boolean assignments to atoms a and b that falsify the predicate. Let us fix one such assignment, $a = b =$ `true`. If the monitor knows a uniform bound on the rate of change $\dot{x}$ of x, say $\forall t.|\dot{x}(t)| \leq 1$, then it can infer that $a =$ `true` cannot hold before $\tau_1 = 2$ (local time). Similarly, if the monitor knows that $|\dot{y}| \leq 3$, then $b =$ `true` cannot hold before $\tau_2 = 0.5$ (local time). Taking into account the ε-synchrony, the monitor can limit itself to monitoring $x|_{[2,T]}$ (the restriction of x to $[2, T]$) and $y|_{[2-\varepsilon, T+\varepsilon]}$.

Now, if this yields `UNSAT` in the SMT instance, we select the next Boolean assignment (in terms of atoms a and b) that falsifies predicate φ (e.g., $a =$ `false` and $b =$ `true`), derive

the useful portion of signals x and y, and repeat the same procedure until the answer to the SMT instance is affirmative or all falsifying Boolean assignments are exhausted. Of course, this requires exploring all such assignments to atoms of the predicate, but since we expect the number of atoms in realistic predicates to be relatively small, the exhaustive nature of falsifying Boolean assignments will not be a bottleneck. We generalize this idea to N agents and arbitrary predicates in Algorithm 4.1. We assume without loss of generality that every atom a that appears in φ is of the form $x_n \geq v_a$ for some n and $v_a \in \mathbb{R}$. A Boolean assignment is a map σ from atoms to $\{\texttt{false}, \texttt{true}\}$, and a violating assignment is one that makes the predicate false. Thus, given a violating assignment σ, for every atom a, $a = \sigma(a)$ iff $x_n \geq v_a$ (if $\sigma(a) = \texttt{true}$) or $x_n < v_a$ (if $\sigma(a) = \texttt{false}$). Obvious modification to Algorithm 4.1 allows the monitor to take advantage of knowing different rate bounds at different points along the signals.

4.5 Case Studies and Evaluation

In this section, we evaluate our technique using two case studies on networks of autonomous ground and aerial vehicles.

4.5.1 Case Study 1: Network of Ground Autonomous Vehicles

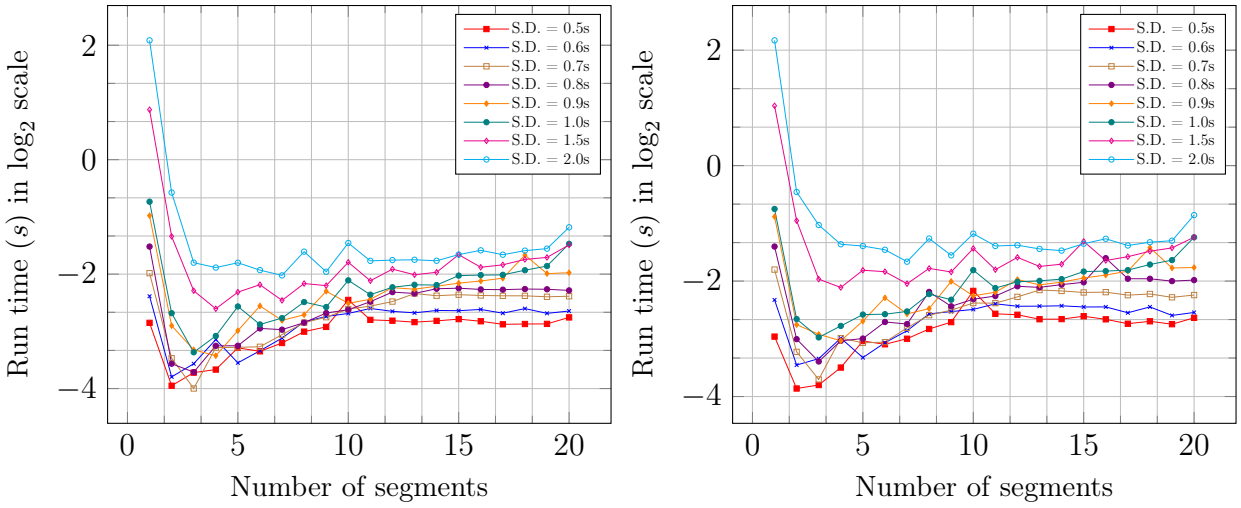
We collected data from two $1/10^{th}$-scale autonomous cars competing in a race around a closed track. Each car carries a LiDAR for perceiving the world, and uses Wi-Fi antennas to communicate with the central monitor. Each car is running a model predictive controller to track its racing line and RRT to adjust its path. The trajectory data is sampled at $25Hz$. In this application, the useful signal length to monitor is $1-2s$, as this is the control horizon (i.e., the controller repeatedly plans the next $1-2s$). Thus, in Eq. (4.1), $T = 1-2s$. A reasonable range for ε is interval $[1, 5]ms$, guaranteed by ROS clock synchronization based on NTP. Unless otherwise indicated, we monitor the predicate $d(x_1, x_2) > \delta \wedge d(x_1, x_2) \leq \Delta$.

4.5.2 Case Study 2: Network of UAVs

We use Fly-by-Logic [100], a path planner software for UAVs, to simulate the operation of two UAVs performing various reach-avoid missions. In a reach-avoid mission, each UAV must reach a goal within a deadline, and must avoid static obstacles as well as other UAVs. The path planner uses a temporal logic robustness optimizer to find the most robust trajectory. The trajectories are sampled at $20Hz$. In this application, the useful signal length to monitor is around $2s$, as this is the UAV's 'reaction time' (depending on current speed). Thus, in Eq. (4.1), $T \approxeq 2s$. A reasonable range for ε is again $1 - 5ms$, guaranteed by ROS. Unless otherwise indicated, we monitor the predicate $d(x_1, x_2) \geq \delta$.

4.5.3 Case Study 3: Water Distribution System

We use a model of a hybrid dynamic high pressure water distribution system consisting of two water tanks. Each water tank has an inlet pipe connected to some external water source, and an outlet pipe with a valve that can be used to regulate high pressure water outflow from each tank. A controller on each water tank operates its valve, and samples the outflow pressure at $20Hz$ using its local clock. We model such a system in Simulink, which is a simplified emulation of the Refueling Water Storage Tanks (RWST) module of an Emergency Core Cooling System (ECCS) of a Pressurized Water Reactor Plant [118] as shown in Figure 1.1. ECCS is tasked with providing core cooling to minimize fuel damage following a 'loss of coolant' accident by administering high pressure water injection from RWST. The water tanks, and by extension their controllers, operate even when the supply of power is lost to the plant. As a failsafe, ECCS also incorporates Cold Leg Accumulators that do not require power to operate. These tanks contain large amounts of borated water with a pressurized nitrogen gas bubble in the top. If the pressure of the outflow pressure drops below a certain threshold, the nitrogen will force the borated water out of the tank and into the reactor coolant system. A reasonable range for ε is $5ms - 500ms$ [13] depending on how often the local clocks of the water tanks are synced with global time. In this case study, we monitor the property that the cumulative pressure of the RWSTs always remains



(a) Network of cars.

(b) Network of UAVs.

Figure 4.5 Impact of signal segmentation on run time with varying signal duration (S.D.) and fixed $\varepsilon = 0.001s$.

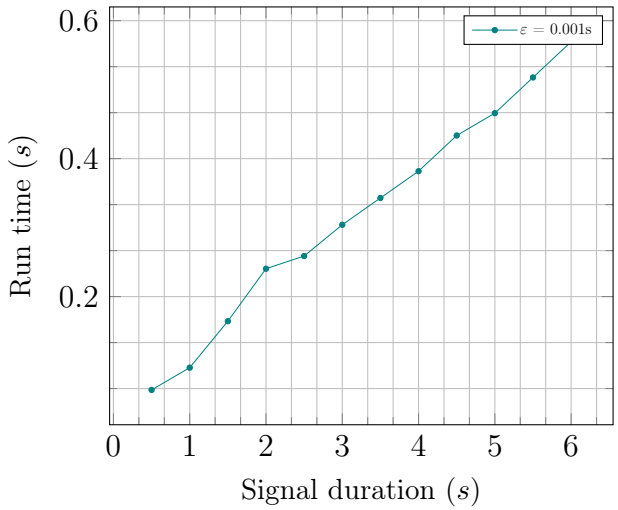


Figure 4.6 Best run time (network of cars) for different signal duration.

above a certain threshold.

Note that the SMT solver's effort is mostly spent on finding retiming, instead of predicate complexity. Thus, we pick simpler predicates for our experiments.

4.5.4 Experimental Setup

In our experiments, we choose the following parameters: (1) signal duration, (2) maximum clock skew ε, and (3) distribution of communication among agents. We measure the

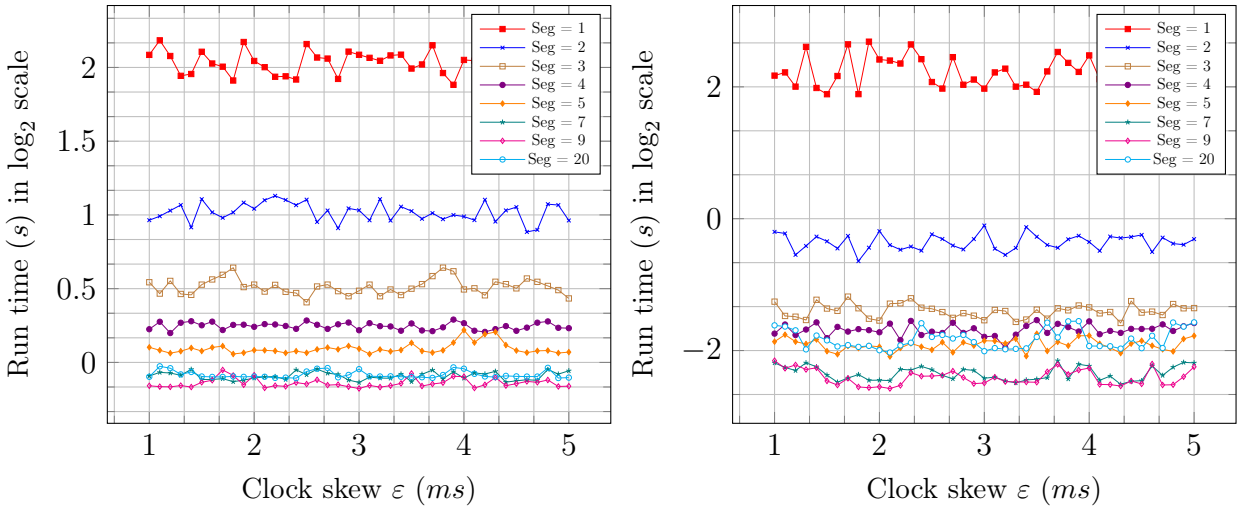
monitor run time. All experiments are replicated to exhibit %95 confidence interval to provide statistical significance. The experimental platform is a CentOS server with 112 Intel(R) Xeon(R) Platinum 8180 CPUs @ 3.80GHz CPU and 754G of RAM. Our implementation invokes the SMT-solver Z3 [97] to solve the problem described in Section 4.3.

4.5.5 Analysis of Results

Impact of signal segmentation Given a signal-to-be-monitored, we have a choice of either passing the entire signal to the monitor, or chopping it into segments and monitoring each segment separately (while accounting for ε-synchrony). Monitoring a signal in one shot is computationally more expensive than monitoring a number of shorter segments. Figure 4.5 shows the results of this claim. Note that all curves are plotted in $\log_2$ scale to provide more clarity. As can be seen, for any signal duration, chopping the signal and invoking the monitor for the shorter segments reduces the run time significantly. For example, in the case of the UAV network (Figure 4.5b), for a signal duration of $2s$, it takes $4.5s$ to monitor the signal in one shot, but only $0.55s$ if the monitor is invoked 20 times over the signal duration. We observe the same behavior in Figure 4.5a. This is due to the SMT-solver having to deal with much smaller search spaces in each invocation.

Figure 4.6 shows the best achievable run time for different signal durations by searching over the segment count of range $[1, 25]$. For example, segment count of 4 is obtained for $1s$ signal to get minimum run time of $0.17s$, while segment count of 18 is obtained for $5s$ signal to get minimum run time of $0.72s$. The best run time shown is achieved by distributing the monitoring tasks across all the available cores (4) on the monitoring device. Notice that our predicate detection algorithm can be parallelized trivially, assigning one or a pool of segments to a different core.

An important consequence of segmentation is that it enables us to monitor signals in real time, as for 3 or more segments, the run time of the monitor is less than the signal duration. For this reason, in all remaining experiments, the signal-to-monitor is chopped into 20 segments and each segment is monitored separately. Cumulative run times (of monitoring



(a) Network of cars.

(b) Network of UAVs.

Figure 4.7 Impact of clock skew on run time. Signal duration = $2s$.

all 20 segments) are reported.

Impact of clock skew We now study the impact of different choices of ε on monitoring run time. We choose realistic values for ε with millisecond resolution. Figure 4.7 shows the monitoring run time for a $2s$ signal chopped into $1-20$ segments. Both Figs. 4.7a and 4.7b show that high resolution clock synchronization results in very stable execution time for the monitor. This is a positive result, showing that for practical clock synchronization algorithms, the actual value of ε does not have an impact on the monitoring overhead. However, naturally ε has an impact on the number of violations detected, specifically false positives. To demonstrate this, we model the path of a pair of UAVs and a pair of cars, where the agents periodically reside within the given mutual separation threshold, and violate the mutual separation property.

Tables 4.1 and 4.2 show the results for two cars and two UAVs, respectively, in operation for half an hour. The experiments report (1) the number of *True Violations* as a baseline that was reverse calculated from the introduced clock drift ε; (2) the number of *Detected Violations* using our method, and (3) the number of *False Positives*, which is essentially the difference between the true violations and the detected violations. Note that there were no

Clock Skew (s)	True Violations	Detected Violations	False Positives
0.05	6	13	7
0.1	3	19	16
0.15	2	29	27
0.2	4	41	37
0.25	1	46	45
0.3	4	52	48
0.35	4	60	56
0.4	5	70	65
0.45	3	80	77
0.5	3	89	86

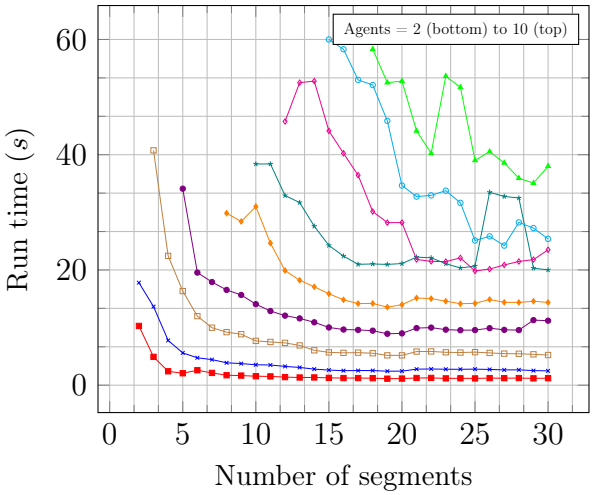
Table 4.1 Impact of clock skew in network of cars on verdicts using varying ε.

Clock Skew (s)	True Violations	Detected Violations	False Positives
0.05	6	11	5
0.1	6	20	14
0.15	8	30	22
0.2	4	39	35
0.25	2	46	44
0.3	1	48	47
0.35	7	62	55
0.4	2	66	64
0.45	5	76	71
0.5	6	84	78

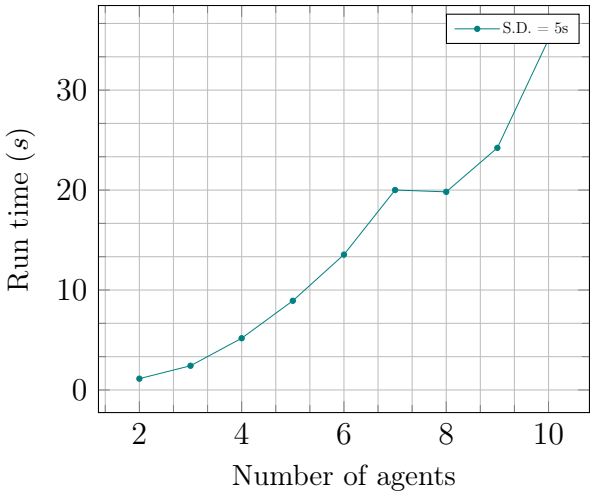
Table 4.2 Impact of clock skew in network of UAVs on verdicts using varying ε.

False Negatives. Furthermore, as the maximum clock skew is increased from $0.05s$ to $0.5s$, naturally the number of False Positives increase as well.

Impact of number of agents Now we observe the impact of the number of UAVs on the monitor. Figure 4.8a shows the effect on run time for increasing the number of agents from 2 to 10 with $\varepsilon = 1ms$ over $5s$-long signals. As each segment of a signal can be monitored independently, we improve our run time by distributing the monitoring tasks across all available cores on the monitoring device. Observe that initially the run time drastically improves as more segments are used. However, eventually the improvement becomes negligible, due to run time being dominated by non-SMT tasks, such as creating job queues, allocating jobs



(a) Signal Duration = 5s and $\varepsilon = 0.001$s.



(b) Signal Duration = 5s and $\varepsilon = 0.001$s.

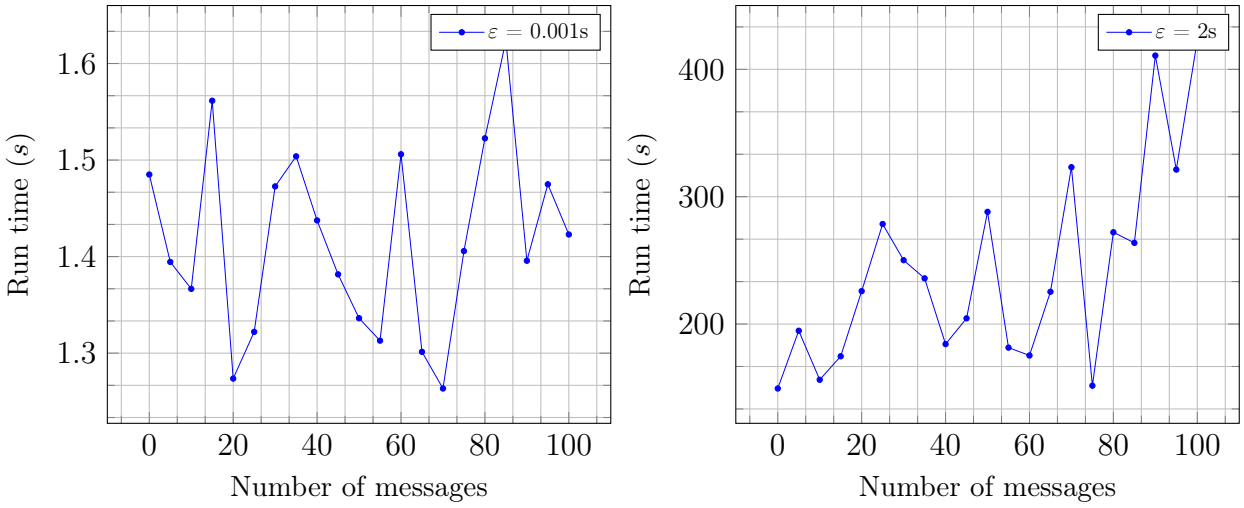
Figure 4.8 Impact of agents on run time.

to cores, and so on. We refer to this run time as the *best run time*. Figure 4.8b shows the best run times for different number of agents with $\varepsilon = 1ms$ over $5s$-long signals.

Impact of communication We examine whether the number of messages exchanged between agents has a significant impact on monitor run time. Two opposing mechanisms exist: on the one hand, messages impose an order between the send and receive moments and so reduce concurrency. In the discrete-time setting this normally reduces the asynchronous monitoring complexity. On the other hand, messages result in extra constraints in the SMT encoding via Eq. 4.7, which could increase SMT run time.

Figure 4.9 shows the results. In (a) we use $\varepsilon = 1ms$ and a $1s$-long signal. Run time varies with no clear trend, suggesting that neither of the above two opposing mechanisms dominates. In (b), we use $\varepsilon = 2s$ for a $2s$-long signal: i.e., *all* events are concurrent. One can see the order introduced by messages are slightly increasing the runtime, instead of decreasing it. No conclusion can be drawn, and future work should study this more closely.

Impact of piece-wise quadratic signals We now compare the effect on run time for piece-wise quadratic signals against piece-wise linear signals. To this end, we consider $1s$-long signals for each signal model generated by the network of cars with $\varepsilon = 0.001ms$. For quadratic signals, each formula is constructed by the corresponding agent with the help of



(a) Signal Duration = 1s and ε = 1s.

(b) Signal Duration = 2s and ε = 2s.

Figure 4.9 Impact of communication (between two agents) on run time.

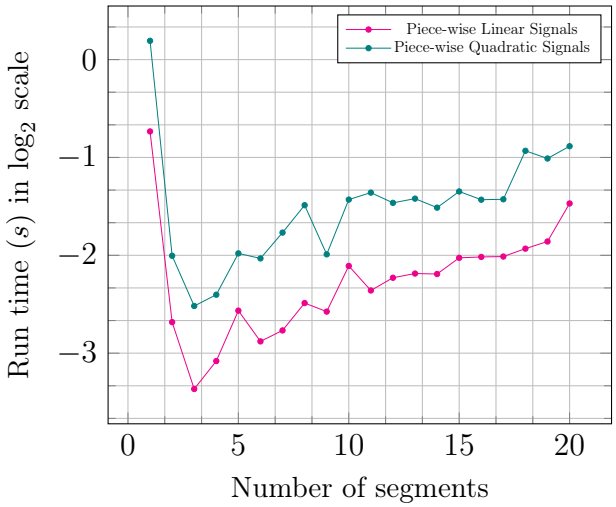
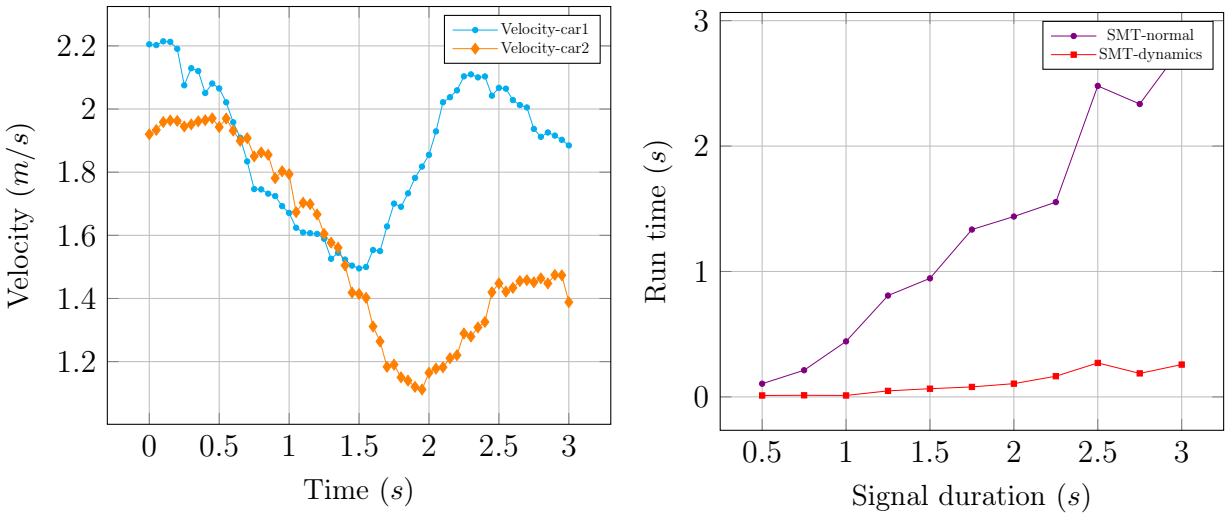


Figure 4.10 Run time (network of cars) vs. segment count.

an SMT solver, using signal value of at current local time, and signal values of last two samples. This formula is then sent to the monitor. The formulas are constructed at their corresponding agents instead of the monitor due to the fact that solving quadratic equations for all agents on each sample point can become an expensive task for the monitor, especially for higher number of agents. In Figure 4.10 we observe the runtime for varying segment counts for both signal models. The runtime for piece-wise quadratic signals is generally higher than its piece-wise linear counterpart due to quadratic signals having three discrete



(a) Velocity profile of two cars.

(b) Run time vs. signal duration.

Figure 4.11 Impact of Algorithm 4.1 on monitoring run time. $\varepsilon = 0.001s$.

sample points (longer) than linear signals with two sample points (shorter). In exchange for this cost in runtime, we achieve better accuracy, as a piece-wise quadratic signal model is a more accurate signal representation when compared to its piece-wise linear signal model counterpart (recall Figure 4.3).

Impact of knowledge of dynamics bounds Here the predicate of interest is $\varphi = (v_1 > 1.6) \vee (v_2 > 1.3)$, where v_i is the velocity of the i^{th} car. The acceleration limit from system dynamics is $1m/s^2$. The monitor samples the received signals (Figure 4.11b) at $0.25s$ intervals and applies the acceleration bounds as explained in Section 4.4 to discard irrelevant pieces of the signal. As shown in Figure 4.11, applying Algorithm 4.1 clearly reduces the monitor run time. In general, of course, the exact run time reduction varies. For instance, while the speedup is $\times 10$ for $3s$-long signals $3s$, it is $\times 15$ for $2s$-long signals.

Impact of segment duration and number of water tanks Let $\mathbf{P}_1$ and $\mathbf{P}_2$ denote the outflow pressure indicated by the respective valve controllers attached to each water tank. For simplicity, we assume all the pipes are of the same diameter. Therefore, the pressure exerted on the Cold Leg Accumulators is $\mathbf{P}_1 + \mathbf{P}_2$. In the experiment, we monitor the property $\varphi_{\mathbf{P}}$, which is, during an emergency, the outflow remains above the threshold

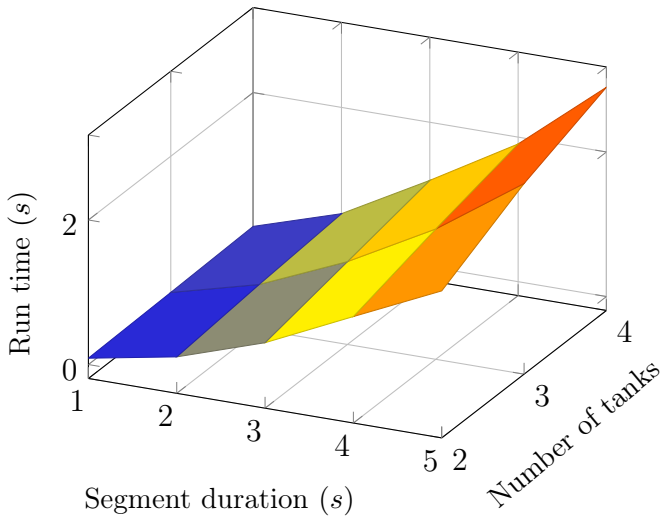


Figure 4.12 Effect of segment duration and the number of water tanks on runtime when $\varepsilon = 0.05s$.

pressure $600psig$ [117], that is, $\varphi_{\mathbf{P}} = \mathbf{P}_1 + \mathbf{P}_2 > 600psig$.

Figure 4.12 shows the effect on runtime for increasing the number of water tanks from 2 to 4 with $\varepsilon = 0.05s$ over segment duration ranging from $1s$ to $5s$. As expected, both segment duration and the number of water tanks contribute to driving up the runtime. We note that even when the monitor receives the distributed signals sent by the water tanks at a reasonable $1s$ intervals, the monitor is still able to verify the property under around half a second for four water tanks.

Impact of clock skew We now study the impact of different choices of ε on monitoring verdicts. To this end, we model two Refueling Water Storage Tanks with intentional 'faults', where the outflow pressures of either water tank can drop below the threshold pressure of the Cold Leg Accumulators. Therefore, if at some moment in time, both the tanks' pressures fall simultaneously, the Cold Leg Accumulators gets triggered. We also introduce a clock drift in the valve controller of one of the water tanks. We choose realistic values for clock drift with millisecond resolution.

Table 4.3 shows the results for two water tanks that were active for an hour. During this operation time, Tank 1 reported low pressures for a total of 35.5 seconds, and Tank 2 reported low pressures for a total of 36.1 seconds. The experiment reports number of

Tank 1 Total Low Pressure Duration (s)	Tank 2 Total Low Pressure Duration (s)	Clock Skew (s)	True Violations	Detected Violations	False Positives
35.5	36.1	0.05	9	25	16
35.5	36.1	0.1	4	42	38
35.5	36.1	0.15	12	65	53
35.5	36.1	0.2	11	80	69
35.5	36.1	0.25	4	86	82
35.5	36.1	0.3	7	99	92
35.5	36.1	0.35	5	112	107
35.5	36.1	0.4	7	127	120
35.5	36.1	0.45	10	145	135
35.5	36.1	0.5	7	160	153

Table 4.3 Impact of clock skew in water tanks on verdicts using varying ε.

True Violations as a baseline that was reverse calculated from the introduced clock drift ε, number of *Detected Violations* using our method, and the number of *False Positives*, which is essentially the difference between the True Violations and the Detected Violations. Note that there were no *False Negatives*. Furthermore, as the maximum clock skew is increased from $0.05s$ to $0.5s$, naturally the number of False Positives increase as well.

4.6 Conclusion

In this chapter, we demonstrated a new approach to online predicate detection for distributed signals that do not share a global clock. To make the problem tractable, in Section 4.5, we use causality analysis between real-valued signals, a reasonable assumption of maximum clock skew among local clocks, and some knowledge of system dynamics. We also studied the influence of signal dynamics information on monitoring efficiency. By experimenting on a real network of autonomous cars, a simulated network of UAVs, and a simulated water distribution system, we discovered that under certain circumstances, our method may be used to successfully monitor a distributed CPS in an online setting. However, this approach only considers Boolean predicates over distributed CPS, and by extension, does not capture more complex specifications, such as, nested and/or temporal properties. In the next chapter, we explore the avenue of monitoring temporal specifications in distributed CPS.

CHAPTER 5

MONITORING SIGNAL TEMPORAL LOGIC IN DISTRIBUTED CYBER-PHYSICAL SYSTEMS

In this chapter, we explore a runtime verification approach for partially synchronous distributed CPS, where we make use of the signal retiming mechanism from the predicate detection technique demonstrated in Chapter 4, and the idea of the progression-based formula rewitting technique demonstrated in Chpater 3. In Chapter 4, we proposed an online predicate monitoring approach for distributed CPS. As mentioned before, the approach is only able to detect Boolean predicates, and therefore, suffers being unable to handle formal specification languages.

5.1 Problem Statement

As distributed agents are partially synchronized within ε clock skew, a monitoring algorithm must explore all (infinite) possible reachable consistent cuts. We call the propagation of consistent cuts with respect to time a *consistent cut flow*. Our objective is to determine whether there exists some flow of moments that are within ε of each other for which at least one reachable consistent cut results in violation of a given STL formula. This intuition is formalized below, starting with the notion of a *consistent cut flow*.

Definition 12. [Consistent cut flow] Let $(E, \rightsquigarrow)$ be a distributed signal over N agents with time interval $[a, b]$, and S be the set of all events over E. A *consistent cut flow* is a function $\text{ccf} : [a, b] \to 2^S$ that maps each time $\chi \in [a, b]$ to the frontier of a consistent cut at time χ; i.e., $\text{ccf}(\chi) \in \{\mathsf{front}(C) \mid C \in \mathcal{C}(\chi)\}$. For each time $\chi' \in [a, b]$, and for each $n \in [N]$, if $\chi < \chi'$,

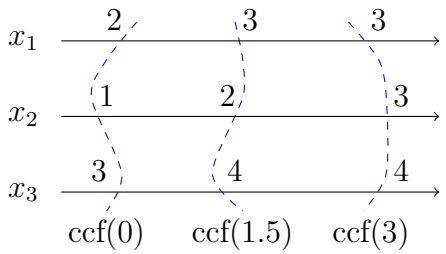


Figure 5.1 A valid ccf.

then for all events $(c_n(\chi), x_n(c_n(\chi))) \in \text{ccf}(\chi)$, and for all events $(c_n(\chi'), x_n(c_n(\chi'))) \in \text{ccf}(\chi')$, $(c_n(\chi), x_n(c_n(\chi))) \rightsquigarrow (c_n(\chi'), x_n(c_n(\chi')))$ hold. ■

Notice that a consistent cut flow induces a vector of N signals that are fully synchronized, and thus, can be verified against an STL formula φ at time t as $(\text{ccf}, t) \models \varphi$ using the semantics described in Section 2.5. That is, for a consistent cut flow ccf on $(E, \rightsquigarrow)$, individual signals $(x'_1, \ldots, x'_N)$ can be constructed, such that, for all $1 \leq i \leq N$ and for all $\chi \in [a, b]$, if $(c_i(\chi), x_i(c_i(\chi))) \in \text{ccf}(\chi)$, then $x'_i(\chi) = x_1(c_n(\chi))$. For example, let $(E, \rightsquigarrow)$ be a distributed signal consisting of signals x_1, x_2, and x_3 as shown Figure 5.1. For an STL formula $\square_{[0,3]}(x_1 + x_2 + x_3 \leq 10)$, ccf is a valid consistent flow on $(E, \rightsquigarrow)$. Note that a distributed signal $(E, \rightsquigarrow)$ encodes uncountably infinite consistent cut flows. Let us denote the set of all consistent cut flows by CCF. Our decision problem consists of determining whether there is a violation of a given STL formula by some consistent cut flow.

Definition 13. [Distributed satisfaction] Let φ be an STL formula, $(E, \rightsquigarrow)$ be a distributed signal over N agents and CCF be the set of all induced consistent cut flows. We say that $((E, \rightsquigarrow), 0)$, or simply $(E, \rightsquigarrow)$ *satisfies* φ, iff for each $\sigma \in \text{CCF}$, we have $\sigma \models \varphi$. ■

Problem Statement

Given maximum clock skew $\varepsilon > 0$, a distributed signal $(E, \rightsquigarrow)$ over N agents, and an STL formula φ, decide whether there exists a consistent cut flow $\sigma \in \text{CCF}$ where $\sigma \not\models \varphi$.

5.2 Monitoring Algorithm

In this chapter, we assume the monitor receives output signals from x_n as *piece-wise linear* signals (this is by choice and other forms of discretization will not change the core monitoring algorithm). This transmission happens in *segments* of length T: at the kth transmission, agent A_n transmits $x_n|_{[(k-1)T, kT]}$, the restriction of its output signal to the interval $[(k-1)T, kT]$ as measured by its local clock. In the rest of this chapter, we refer exclusively to the signal fragments received by the monitor in a given transmission.

We now revisit the restriction placed on I_n in Definition 7, namely, that the monitor only deals with non-empty bounded signal fragments $x_n|_{[(k-1)T,kT]}$, therefore, $I_n = [(k-1)T, kT]$ for every agent at the k^{th} transmission, *measured in local time*. By the bounded skew assumption, we have:

Lemma 8. [Bounded skew lemma] For any two agents A_n, A_m with the intervals $I_n = [\min I_n, \max I_n]$ and $I_m = [\min I_m, \max I_m]$, $|\min I_n - \min I_m| \leq \varepsilon$ and $|\max I_n - \max I_m| \leq \varepsilon$.

Proof. Assume $|\min I_n - \min I_m| > \varepsilon$. However, both $\min I_n$ and $\min I_m$ are lower bounds of I_n and I_m respectively, at the k^{th} transmission. Therefore, by definition of partial synchrony, the difference of their values must not exceed the maximum clock skew ε. Therefore, our assumption is not possible. Thus, $|\min I_n - \min I_m| \leq \varepsilon$. Similarly, we can show that $|\max I_n - \max I_m| \leq \varepsilon$. ■

Since online monitoring happens in segments, at the end of each segment the monitor either returns $\top$ (formula already satisfied), $\bot$ (already violated), or *unknown*, and the next segment is processed. For simplicity, our solution employs a *central monitor*. Our monitoring algorithm involves three key ideas: (1) formula progression, (2) signal retiming, and (3) SMT-based implementation, explained in the following sections.

5.2.1 Formula Progression

Let φ be an STL formula and $(E, \rightsquigarrow)$ be a distributed signal. Without loss of generality, let this signal be split into two segments: prefix $(E_1, \rightsquigarrow)$ and suffix $(E_2, \rightsquigarrow)$. That is, $(E, \rightsquigarrow) = (E_1 E_2, \rightsquigarrow)$. Thus, the monitor first evaluates φ on $(E_1, \rightsquigarrow)$. If the verdict yields true or false, then this verdict is returned and monitoring for $(E, \rightsquigarrow)$ is already complete. Otherwise, the monitor computes a new *progressed formula* φ' which will be evaluated for segment $(E_2, \rightsquigarrow)$.

Definition 14. [Formula progression] Let $(E_1, \rightsquigarrow)$ be a finite distributed signal starting at time 0 whose duration is denoted by $|(E_1, \rightsquigarrow)|$, and $(E_2, \rightsquigarrow)$ be a finite or infinite extension of $(E_1, \rightsquigarrow)$. We say STL formula φ' is a *progression* of STL formula φ for $(E_1, \rightsquigarrow)$ if and only

if:

$$((E_1 E_2, \rightsquigarrow), 0) \models \varphi \Leftrightarrow ((E_2, \rightsquigarrow), 0) \models \varphi'.$$

It stands to reason that if $((E_1, \rightsquigarrow), 0) \models \varphi$ (resp., $((E_1, \rightsquigarrow), 0) \not\models \varphi$), then the progression of φ is trivially $\varphi' = \top$ (resp., $\varphi' = \bot$). ■

5.2.2 Signal Retiming

Recall that signals are measured using their local clocks. Since the signals in our setting are partially synchronized within an ε, it is not possible to evaluate all signals at the same moment in global time. Rather, the best a monitor can do is explore all valid alignments of the concurrent local moments (i.e., those moments that are within ε of each other) and determine whether at least one such alignment violates the formula. This intuition is formalized below, starting with the notion of a *retiming function* borrowed from Chapter 4 that establishes the happened-before relation in the continuous-time setting, and stretches or compresses signals to align them with each other within the ε clock skew bound.

A valid retiming formalizes the notion of alignment of timelines: given two ε-synchronous timelines t and s (on two agents), we treat moments t and $s = \rho(t)$ as being simultaneous. Thus, the signal $x(t) = [x_1(t), x_2(\rho(t))]$ is now a fully synchronous signal. An ε-retiming ρ maps $\mathbb{R}_+$ to itself, but the restriction of ρ to a bounded interval I is an increasing function from I to $\rho(I)$ that respects the constraint $|t - \rho(t)| < \varepsilon$ for all $t \in I$. Thus, we restrict our attention to ε-retimings on bounded intervals. Between 2 agents, we need one retiming $\rho : I_2 \to I_1$, and between N agents, we need $N - 1$ retimings $I_n \to I_1$. In general there is an infinity of valid retimings, any of which might reveal a potential violation. The next theorem establish the fundamental condition on ε-retimings among agents and violation of an STL formula.

Theorem 4. Given a distributed signal $(E, \rightsquigarrow)$ over N agents, and an STL formula φ with time interval $[a, b]$, there exists a violation at time $t \in \mathbb{R}_+$, if and only if there exists $N - 1$

ε-retimings $\rho_n : I_n \to I_1$ that respect $\rightsquigarrow$, where $2 \leq n \leq N$, such that:

$$\Big(\big(x_1, x_2 \circ \rho_2^{-1}, \ldots, x_N \circ \rho_N^{-1}\big), t\Big) \not\models \varphi \tag{5.1}$$

Here, $\rho_m^{-1} \circ \rho_n : I_n \to I_m$ is an ε-retiming for all $n \neq m$, and '$\circ$' denotes the function composition operator, where given two functions f and g, $h = g \circ f$ such that $h(x) = g(f(x))$.

Proof. We distinguish the following cases:

Case 1: Suppose that such retimings exist. We define local time values for each time $\chi \in [t + a, t + b]$ for agents A_1, A_2, ..., A_N respectively as $t_1^\chi = c_1(\chi)$, $t_2^\chi = \rho_2^{-1}(c_1(\chi))$, ..., $t_N^\chi = \rho_N^{-1}(c_1(\chi))$, where $2 \leq n \leq N$. In other words, $t_1^\chi = c1(\chi)$, $t_2^\chi = \rho_2^{-1}(c_1(\chi))$, ..., $t_N^\chi = \rho_N^{-1}(c_1(\chi))$ are the local times of agents A_1, A_2, ..., A_N respectively at global time χ. Furthermore, define $C_\chi = \{(t_n, x_n(t_n)) \mid t_n \leq t_n^\chi \wedge n \in N\}$. By the construction of C_χ, and the fact that the retimings respect $\rightsquigarrow$, it holds that if $e \in C_\chi$ and $f \rightsquigarrow e$, then $f \in C_\chi$. For every $n, m \geq 2$ and $n \neq m$, it holds that $t_m^\chi = \rho_m^{-1}(\rho_n(t_n^\chi))$ so $|t_n^\chi - t_m^\chi| \leq \varepsilon$. Thus, C_χ is a consistent cut, and the flow of frontiers $\mathsf{front}(C_\chi)$, where $\chi \in \mathbb{R}_+$, is a consistent cut flow $\sigma \in \mathrm{CCF}$ that witnesses the violation of φ.

Case 2: Suppose $\sigma \in \mathrm{CCF}$ is a consistent cut flow that violate φ. By definition, there must be consistent cuts in σ that violate φ. Let C_χ denote such consistent cuts, and let $\mathsf{front}(C_\chi)$ denote their frontiers. For every two events $(t_n, x_n(t_n))$ and $(t_m, x_m(t_m))$ in $\mathsf{front}(C_\chi)$, we have $|t_n - t_m| \leq \varepsilon$. Since $(t_n, x_n(t_n)) \in \mathsf{front}(C_\chi)$, we have $(s, x_m(s)) \in C_\chi$ for all s s.t. $s + \varepsilon \leq t_n$. Thus, $t_m \geq s$ for all such s and so $t_m \geq t_n - \varepsilon$. By symmetry of the argument, $t_n \geq t_m - \varepsilon$ holds as well, implying a retiming indeed does exist. ■

5.2.3 SMT Encoding

We solve the monitoring problem by transforming it into an instance of the *satisfiability modulo theory* (SMT). Specifically, we ask whether there exists $N - 1$ retimings, such that (5.1) holds; equivalently, whether there exists a consistent cut flow that witnesses satisfaction of $\neg\varphi$. That is, the distributed signal violates φ iff the following SMT problem is satisfiable. This transformation to SMT solving is the focus of the next section.

5.3 SMT-based Monitoring Algorithm

The SMT formulation part of our solution is constructed by encoding both formula progression and signal retiming into a single SMT-solving problem, and then solving it using an SMT-solver. First, we define the SMT entities and constraints, then demonstrate our monitoring approach with two complete examples. In both examples, we consider a distributed signal $(E, \rightsquigarrow)$ comprised of two individual 10 time unit long signals x_1 and x_2, generated by agents A_1 and A_2 respectively, with a clock skew bound $\varepsilon = 1$. Our running examples involve monitoring formulas $\neg\varphi_1 = \Diamond_{[0,10]}\, p$ and $\neg\varphi_2 = \Diamond_{[0,10]}(p \wedge \Box_{[0,5]}\, \neg q)$.

5.3.1 SMT Entities

In our encoding, N signals and time intervals are defined in the same fashion as the mathematical representation in previous sections. We also include ρ_n retiming functions, where $2 \leq n \leq N$, a consistent cut flow function ccf as an *uninterpreted* function, and real numbers t, s, and χ. Identifying interpretations of these functions will be the output SMT solving and, hence, the verdict of monitoring. The sampled signal values are constants in the encoding that are known to the monitor: $\{x_n(t_n) \mid t_n \in I_n\}$.

5.3.2 SMT Constraints

Recall from Section 5.1 that $(\text{ccf}, t) \models p$ denotes a consistent cut flow at time t on signals $(x'_1, \ldots, x'_N)$ satisfies the atom p. To express this as an SMT problem, we encode $(\text{ccf}, t) \models p$ as $f(x'_1[t], \ldots, x'_n[t]) > 0$, where $(x_1[t], \ldots, x_n[t]) \in \mathbb{R}^n$ is a vector of signal values at time t, and $f : \mathbb{R}^n \to \mathbb{R}$ is a function that evaluates a vector of signal values. The SMT constraints are primarily comprised of (1) a set of constraints that ensures valid consistent cut flow, (2) a set of constraints that find violation, and (3) a set of constraints that enforce valid retimings under a given clock skew.

Consistent cut flow constraints. In order to ensure that ccf identifies a valid consistent cut flow on $(E, \rightsquigarrow)$ over time interval $[a, b]$, first we define the happened-before $(\rightsquigarrow)$ notation in SMT according to Definition 7, and ensure that the events in the consistent cuts mapped

by ccf respect the happened-before relation:

$$SMT_{\mathsf{flow}_1} = \forall \chi \in [a,b] \;.\; \forall (t_n, x_n(t_n)), (t'_n, x_n(t'_n)) \in E \;.$$
$$\Big(\big((t'_n, x_n(t'_n)) \rightsquigarrow (t_n, x_n(t_n)) \big) \;\wedge\; \big((t_n, x_n(t_n)) \in \mathrm{ccf}(\chi) \big) \Big)$$
$$\Rightarrow \Big((t'_n, x_n(t'_n)) \in \mathrm{ccf}(\chi) \Big).$$

And that the consistent cuts mapped by ccf always increase and never intersect:

$$SMT_{\mathsf{flow}_2} = \forall \chi, \chi' \in [a,b] \;.\; \forall n \in [N] \;.\; \Big(\chi < \chi' \Rightarrow c_n(\chi) < c_n(\chi') \Big).$$

Thus, the SMT constraint for consistent cut flow is the following:

$$SMT_{\mathsf{flow}} = SMT_{\mathsf{flow}_1} \;\wedge\; SMT_{\mathsf{flow}_2}.$$

Retiming constraints over ccf. We ensure

$$SMT_{\mathsf{retime}_1} = \forall \chi \in [a,b] \;.\; \forall c_2(\chi) \in I_2 \;.\; \exists c_1(\chi) \in I_1 \;.$$
$$\Big(\rho(c_2(\chi)) = c_1(\chi) \big) \;\wedge\; \big(|c_1(\chi) - c_2(\chi)| < \varepsilon \Big)$$

And that ρ is always increasing:

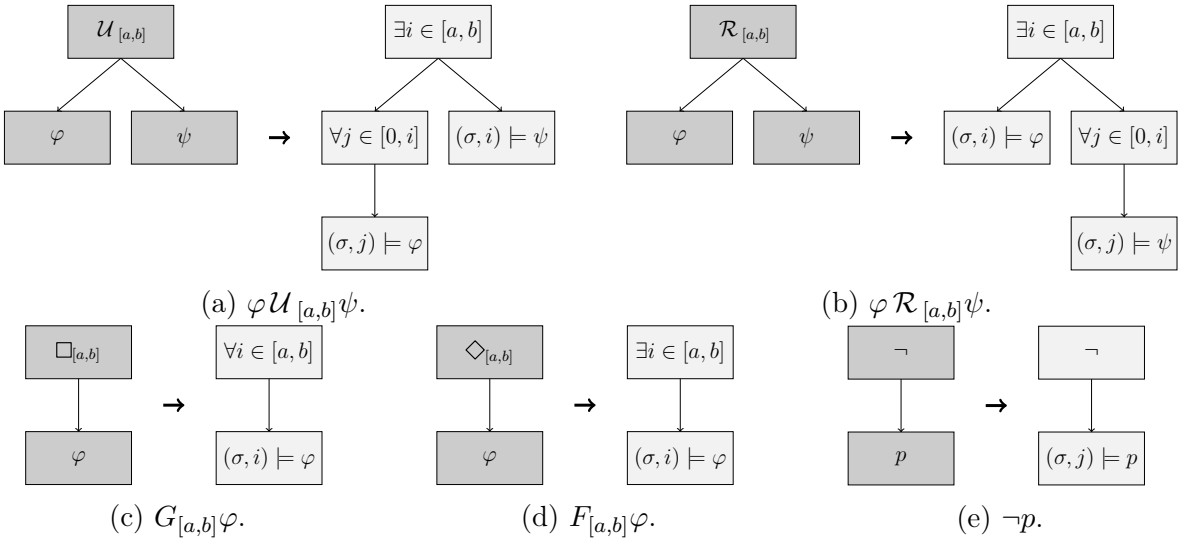
$$SMT_{\mathsf{retime}_2} = \forall \chi, \chi' \in [a,b] \;.\; \forall c_2(\chi), c_2(\chi') \in I_2 \;.$$
$$\Big(c_2(\chi) < c_2(\chi') \Rightarrow \rho(c_2(\chi)) < \rho(c_2(\chi')) \Big)$$

When there are more than 2 agents , we must also encode the constraint that for all $n \neq m$, $\rho_m^{-1} \circ \rho_n$ is an ε-retiming. Therefore, for all $n \neq m$, denoting f_m as the uninterpreted function that represents the inverse of the uninterpreted c_m:

$$SMT_{\mathsf{retime}_3} = \forall t \in I_n \;.\; f_m(\rho_n(t)) = t$$

Thus, the SMT constraint for signal retiming if the following:

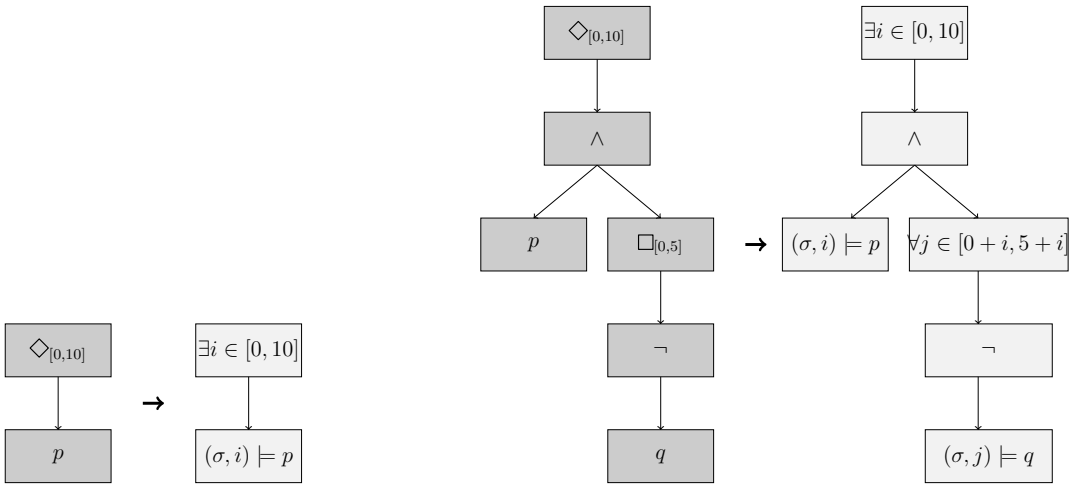
$$SMT_{\mathsf{retime}} = SMT_{\mathsf{retime}_1} \;\wedge\; SMT_{\mathsf{retime}_2} \;\wedge\; SMT_{\mathsf{retime}_3}.$$



(a) $\varphi \mathcal{U}_{[a,b]} \psi$. (b) $\varphi \mathcal{R}_{[a,b]} \psi$.

(c) $G_{[a,b]}\varphi$. (d) $F_{[a,b]}\varphi$. (e) $\neg p$.

Figure 5.2 Conversion of STL syntax trees to their corresponding SMT syntax tree.



(a) $\neg\varphi_1 = F_{[0,10]}p$. (b) $\neg\varphi_2 = F_{[0,10]}(p \wedge G_{[0,5]}\neg q)$.

Figure 5.3 SMT syntax tree of STL formulas $\neg\varphi_1$ and $\neg\varphi_2$.

Violation constraints over $(E, \rightsquigarrow)$. Let γ_φ be the *syntax tree* representation of an STL formula φ, where each internal node represents an operator, and each leaf node represents an atomic proposition. We convert γ_φ to its *SMT syntax tree* representation τ_φ. An SMT syntax tree τ_φ is a tree obtained from an STL syntax tree γ_φ by replacing each temporal operator in the non-leaf node of γ_φ with its corresponding SMT encoding. In τ_φ as well, each leaf represents an atomic proposition. The purpose of converting an STL formula φ to its SMT syntax tree representation τ_φ is to be able to easily manipulate the syntax tree and parse its corresponding SMT encoding. Figure 5.2 shows the process of converting all five

subtrees with STL operators to their corresponding SMT syntax tree representations. For nested formulas, this process is done for every formula in the STL syntax tree, starting from the root of the tree.

For example, Figs. 5.3a and 5.3b show creating SMT syntax trees of $\neg\varphi_1$ and $\neg\varphi_2$ using the technique shown in Figure 5.2. Let $\tau_{\neg\varphi_1}$ (resp., $\tau_{\neg\varphi_2}$) be the SMT syntax trees created from $\neg\varphi_1$ (resp., $\neg\varphi_2$). Let us first consider the case where the monitor has the whole distributed signal $(E, \rightsquigarrow)$ (i.e., no segmentation). The case of a segmented signal will be handled by formula progression explained in Section 5.3.3. Thus, we keep the SMT syntax trees unchanged and we denote the corresponding SMT constraint by SMT_{τ_φ}. From Figure 5.3a, for $\neg\varphi_1$, the distributed signal $(E, \rightsquigarrow)$, and the SMT syntax tree $\tau_{\neg\varphi_1}$, we have:

$$SMT_{\tau_{\neg\varphi_1}} = \exists i \in [0, 10].((\text{ccf}, i) \models p).$$

Recall from the beginning of this section '$(\text{ccf}, i) \models p$' is replaced with the $f(.) > 0$ in the SMT constraint. For $\neg\varphi_2$, we have:

$$SMT_{\tau_{\neg\varphi_2}} = \exists i \in [0, 10].((\text{ccf}, i) \models p \ \wedge \ \forall j \in [0 + i, 5 + i](\neg((\text{ccf}, j) \models q))).$$

Putting everything together. The final SMT constraint is the following:

$$FinalSMT = SMT_{\mathsf{flow}} \ \wedge \ SMT_{\mathsf{retime}} \ \wedge \ SMT_{\tau_{\neg\varphi}}.$$

Obviously, since there is logical equivalence between an STL formula φ and its corresponding SMT encoding SMT_{τ_φ}, for any given a distributed signal $(E, \rightsquigarrow)$ over N agents, we have $(E, \rightsquigarrow) \not\models \varphi$ if and only if $FinalSMT$ is satisfiable (assuming all time intervals of temporal operators are within $[0, |(E, \rightsquigarrow)|]$).

5.3.3 Formula Progression

We now consider the case where the monitor does not have the entire distributed signal and receives it in segments, or, time intervals of some temporal operators are not within $[0, |(E, \rightsquigarrow)|]$. Given a segment $(E, \rightsquigarrow)$ and formula φ, our goal is to obtain a *progressed formula* φ' such that any (finite or infinite) extension $(E', \rightsquigarrow)$ will be evaluated for φ'.

Data: SMT syntax tree τ_φ, partition time t
Result: SMT syntax tree τ'_φ
Let $root_\tau$ be the root node of τ_φ and n_τ be a node
Function `PartitionTree(`n_τ`):`

```
if n_τ has a quantifier with range '[a, b]' then
    if a < t ≤ b then
        Let n'_τ be an empty node
        if n_τ has quantifier '∀' then
            Label n'_τ as '∧'
        end
        if n_τ has quantifier '∃' then
            Label n'_τ as ∨
        end
        n'_τ.leftchild ← copy subtree rooted at n_τ  Set '[a, min(b, t))' as the quantifier
         range of n'_τ.l
        n'_τ.rightchild ← copy subtree rooted at n_τ  Set '[max(a, t), b]' as the quantifier
         range of n'_τ.r
        if n_τ ≠ root_τ then
            n_τ.parent.child ← n'_τ
        end
        else
            n_τ ← n'_τ
        end
    end
end
foreach n_τ.child in n_τ do
    PartitionTree(n_τ.child)
end
return PartitionTree(root_τ)
```

Algorithm 5.1 Function Λ.

We define function Λ, that takes as input an SMT syntax tree τ_φ and a segment duration $|(E, \rightsquigarrow)|$ and returns as output (see Algorithm 5.1) an SMT syntax tree $\tau'_\varphi = \Lambda(\tau_\varphi, |(E, \rightsquigarrow)|)$. We construct an SMT syntax tree τ'_{φ_μ} from τ'_φ such that the following properties hold:

- The root of τ'_{φ_μ} is the topmost (and leftmost if there are two) node of τ'_φ which has a quantifier label.
- For every subsequent nodes, in τ'_{φ_μ}, if the node n has the label $\wedge$ or $\vee$ with children labelled with quantifiers, remove the node and only keep the left child by doing $n.parent = n.leftchild$.

As examples, let us partition the SMT syntax trees in $\tau_{\neg\varphi_1}$ (Figure 5.3a) and $\tau_{\neg\varphi_2}$ (Figure 5.3b) at time $t = 5$ using Algorithm 5.1. For $\tau_{\neg\varphi_1}$, since the starting node n_τ, which is the root node in this case, is labelled '$\exists i \in [0, 10]$', we create a node n'_τ and label it '$\vee$'. Now we create two copies of the tree at n_τ, change the ranges to '$[0, 5)$' (resp., '$[5, 10]$'), and attach them to left (resp., right) children of n'_τ. n'_τ is our new n_τ. Now, we repeat the process for each child of n_τ. However, as none of the children nodes are labelled with quantifiers, $\tau'_{\neg\varphi_1} = n_\tau$ is our desired partitioned tree from $\tau_{\neg\varphi_1}$ at time $t = 5$, shown in Figure 5.2. Following the same process, we get $\tau'_{\neg\varphi_2}$ as our partitioned tree from $\tau_{\neg\varphi_2}$ at time $t = 5$, shown in Figure 5.3.

Lemma 9. [SMT partition tree lemma] Let $(E, \rightsquigarrow)$ be a distributed signal and φ be an STL formula. $FinalSMT$ for $(E, \rightsquigarrow)$ and τ_φ is satisfiable if and only if $FinalSMT$ for $(E, \rightsquigarrow)$ and $\Lambda(\tau_\varphi, |(E, \rightsquigarrow)|)$ is satisfiable.

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, $\varphi = p$.

($\Rightarrow$) The SMT encoding for $(E, \rightsquigarrow)$ and τ_p is:

$$(\text{ccf}, 0) \models p$$

In other words, when the encoding above is satisfied, the events in the frontier of the consistent cut at time 0 satisfies p. Now, as the SMT syntax tree for p does not have any quantifiers, Algorithm 5.1 never enters succeeds $a < t \leq b$. Hence, the SMT syntax tree for p remains unchanged, and the SMT encoding using E and $\tau'_\varphi = \Lambda(\tau_\varphi, |E|)$ is:

$$(\text{ccf}, 0) \models p$$

($\Leftarrow$) Trivial.

Case 2: 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 $\varphi = \varphi_1 \wedge \varphi_2$.

($\Rightarrow$) The SMT encoding for $(E, \rightsquigarrow)$ and $\tau_{\varphi_1 \wedge \varphi_2}$ is:

$$(\text{ccf}, 0) \models \varphi_1 \wedge \varphi_2$$

In other words, when the encoding above is satisfied, the events in the frontier of the consistent cut at time 0 satisfies $\varphi_1 \wedge \varphi_2$. Now, as the SMT syntax tree for φ does not have any quantifiers, Algorithm 5.1 never succeeds $a < t \leq b$. Hence, the SMT syntax tree for φ remains unchanged, and the SMT encoding using E and $\tau'_{\varphi_1 \wedge \varphi_2} = \Lambda(\tau_{\varphi_1 \wedge \varphi_2}, t')$ is:

$$(\text{ccf}, 0) \models (\varphi_1 \wedge \varphi_2) \wedge \texttt{true}$$

($\Leftarrow$) Trivial.

Case 3: 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 $\varphi = \varphi_1 \vee \varphi_2$.

($\Rightarrow$) The SMT encoding for $(E, \rightsquigarrow)$ and $\tau_{\varphi_1 \vee \varphi_2}$ is:

$$(\text{ccf}, 0) \models \varphi_1 \vee \varphi_2$$

In other words, when the encoding above is satisfied, the events in the frontier of the consistent cut at time 0 satisfies $\varphi_1 \vee \varphi_2$. Now, as the SMT syntax tree for φ does not have any quantifiers, Algorithm 5.1 never succeeds $a < t \leq b$. Hence, the SMT syntax tree for φ remains unchanged, and the SMT encoding for $(E, \rightsquigarrow)$ and $\tau'_{\varphi_1 \vee \varphi_2} = \Lambda(\tau_{\varphi_1 \vee \varphi_2}, t')$ is:

$$(\text{ccf}, 0) \models \varphi_1 \vee \varphi_2$$

($\Leftarrow$) Trivial.

Case 4: Assume that the proof has been established for the cases when the formulas are $\varphi = \varphi_1$ and $\varphi = \varphi_2$. We consider the case where the formula is $\varphi = \varphi_1 \mathcal{U}_{[a,b]} \varphi_2$.

($\Rightarrow$) The SMT encoding for $(E, \rightsquigarrow)$ and $\tau_{\varphi_1 \mathcal{U}_{[a,b]} \varphi_2}$ is:

$$\exists i \in [a, b] \Big((\text{ccf}, i) \models \varphi_2 \wedge \forall j \in [0, i) \big(\text{ccf}, j\big) \models \varphi_1 \big) \Big)$$

If the above encoding is SAT, then both $\exists i \in [a,b]\big((\text{ccf},i) \models \varphi_2\big)$ and $\exists i \in [a,b]\forall j \in [0,i)\big((\text{ccf},j) \models \varphi_1\big)$ are SAT. For $a < |E| \leq b$, this can be written as:

$$\exists i_1 \in [a,|E|)\Big((\text{ccf},i_1) = \varphi_2 \wedge \big(\forall j_1 \in [0,i_1]((\text{ccf},j_1) = \varphi_1)\big)\Big)$$

$$\vee$$

$$\exists i_2 \in [|E|,b]\Big((\text{ccf},i_2) = \varphi_2 \wedge \big(\forall j_2 \in [|E|,b]((\text{ccf},j_2) = \varphi_1)\big)\Big)$$

Note that this is the SMT encoding for $(E,\rightsquigarrow)$ and $\tau'_{\varphi_1 \mathcal{U}_{[a,b]}\varphi_2} = \Lambda(\tau_{\varphi_1 \mathcal{U}_{[a,b]}\varphi_2}, |E|)$, when $a < |E| \leq b$. For any other value of $a < |E| \leq b$, the SMT syntax tree remains unchanged. When the SMT encoding of $\tau_{\varphi_1 \mathcal{U}_{[a,b]}\varphi_2}$ is SAT, either (1) $\varphi_1 \mathcal{U}_{[a,|E|]}\varphi_2$ is satisfied, or (2) φ_1 is satisfied throughout $[0,|E|)$, and $\varphi_1 \mathcal{U}_{[|E|,b]}\varphi_2$ is satisfied. If $\varphi_1 \mathcal{U}_{[a,|E|]}\varphi_2$ is satisfied, then the first part of the SMT encoding of $\tau'_{\varphi_1 \mathcal{U}_{[a,b]}\varphi_2}$ becomes SAT, and if φ_1 is satisfied throughout $[0,|E|)$, and $\varphi_1 \mathcal{U}_{[|E|,b]}\varphi_2$ is satisfied, then the second part of the SMT encoding of $\tau'_{\varphi_1 \mathcal{U}_{[a,b]}\varphi_2}$ becomes SAT. Therefore, in all possible cases, if the SMT encoding of $\tau_{\varphi_1 \mathcal{U}_{[a,b]}\varphi_2}$ yields SAT, then the SMT encoding of $\tau'_{\varphi_1 \mathcal{U}_{[a,b]}\varphi_2}$ will also yield SAT.

($\Leftarrow$) Trivial.

Case 5: Assume that the proof has been established for the cases when the formulas are $\varphi = \varphi_1$ and $\varphi = \varphi_2$. Finally, we consider the case where the formula is $\varphi = \varphi_1 \mathcal{R}_{[a,b]}\varphi_2$.

($\Rightarrow$) The SMT encoding for $(E,\rightsquigarrow)$ and $\tau_{\varphi_1 \mathcal{R}_{[a,b]}\varphi_2}$ is:

$$\exists i \in [a,b]\Big((\text{ccf},i) \models \varphi_1 \wedge \forall j \in [0,i)\big(\text{ccf},j) \models \varphi_2\big)\Big)$$

If the above encoding is SAT, then both $\exists i \in [a,b]\big((\text{ccf},i) \models \varphi_1\big)$ and $\exists i \in [a,b]\forall j \in [0,i)\big((\text{ccf},j) \models \varphi_2\big)$ are SAT. For $a < |E| \leq b$, this can be written as:

$$\exists i_1 \in [a,|E|)\Big((\text{ccf},i_1) = \varphi_1 \wedge \big(\forall j_1 \in [0,i_1]((\text{ccf},j_1) = \varphi_2)\big)\Big)$$

$$\vee$$

$$\exists i_2 \in [|E|,b]\Big((\text{ccf},i_2) = \varphi_1 \wedge \big(\forall j_2 \in [|E|,b]((\text{ccf},j_2) = \varphi_2)\big)\Big)$$

Note that this is the SMT encoding for $(E, \rightsquigarrow)$ and $\tau'_{\varphi_1 \mathcal{R}_{[a,b]} \varphi_2} = \Lambda(\tau_{\varphi_1 \mathcal{R}_{[a,b]} \varphi_2}, |E|)$, when $a < |E| \leq b$. For any other value of $a < |E| \leq b$, the SMT syntax tree remains unchanged. When the SMT encoding of $\tau_{\varphi_1 \mathcal{R}_{[a,b]} \varphi_2}$ is SAT, either (1) $\varphi_1 \mathcal{R}_{[a,|E|]} \varphi_2$ is satisfied, or (2) φ_2 is satisfied throughout $[0, |E|)$, and $\varphi_1 \mathcal{R}_{[|E|,b]} \varphi_2$ is satisfied. If $\varphi_1 \mathcal{R}_{[a,|E|]} \varphi_2$ is satisfied, then the first part of the SMT encoding of $\tau'_{\varphi_1 \mathcal{R}_{[a,b]} \varphi_2}$ becomes SAT, and if φ_2 is satisfied throughout $[0, |E|)$, and $\varphi_1 \mathcal{R}_{[|E|,b]} \varphi_2$ is satisfied, then the second part of the SMT encoding of $\tau'_{\varphi_1 \mathcal{R}_{[a,b]} \varphi_2}$ becomes SAT. Therefore, in all possible cases, if the SMT encoding of $\tau_{\varphi_1 \mathcal{R}_{[a,b]} \varphi_2}$ yields SAT, then the SMT encoding of $\tau'_{\varphi_1 \mathcal{R}_{[a,b]} \varphi_2}$ will also yield SAT.

($\Leftarrow$) Trivial.

■

Given a distributed signal $(E', \rightsquigarrow)$ and an STL formula φ, the following theorem shows that the subtree τ'_{φ_μ} of $\Lambda(\tau_{\neg\varphi}, |(E, \rightsquigarrow)|)$ allows computing the progressed formula by discharging τ'_{φ_μ}.

Theorem 5. [Partial evaluation theorem] Let $(E, \rightsquigarrow)$ be a distributed signal and φ be an STL formula. It is the case that $(E, \rightsquigarrow) \models \varphi_\mu$ if and only if $FinalSMT$ for $(E, \rightsquigarrow)$ and τ'_{φ_μ} is satisfiable.

Proof. Let us assume that $\tau'_\varphi = \Lambda(\tau_\varphi, |E|)$, $E \models \varphi_\mu$, and $FinalSMT$ for $(E, \rightsquigarrow)$ and τ'_{φ_μ} is not satisfiable. This implies that τ'_{φ_μ} has at least one subtree, where the root node is the n^{th} nested quantifier with an interval $[\alpha_n, \beta_n]$ and $\beta_n > |E|$. However, while constructing τ'_{φ_μ}, only the left child is kept for any node that has the label $\wedge$ or $\vee$ with children labelled with quantifiers (see Section 3.3). Furthermore, In Algorithm 5.1, the maximum range of the quantifier labelled on the left child is $\min(\beta_n, |E|)$. Therefore, $\beta_n > |E|$ is not possible. Therefore, such a subtree cannot exist, and by extension τ'_{φ_μ} cannot exist. Thus, $E \models \varphi_\mu$ if and only if $FinalSMT$ for $(E, \rightsquigarrow)$ and τ'_{φ_μ} is satisfiable. ■

Simply evaluating $FinalSMT$ for $(E, \rightsquigarrow)$ and τ'_{φ_μ} is not enough, as we must ensure that there is no loss of information when modifying τ'_φ using the said evaluation results. For example, in Figure 5.4b, Since $(\sigma, j_2) \models \neg q$ cannot be evaluated on the first segment, finding

only one value of i_1 in this segment may lead to loss of information, as this may ignore other valid values of i_1 that are required to evaluated $(\sigma, j_2) \models \neg q$ on the next segment.

Note that any modification to τ'_φ would naturally occur only in its τ'_{φ_μ} subtree. To this end, we define a function υ, that takes as inputs an SMT syntax tree τ'_{φ_μ} and a distributed signal $(E, \rightsquigarrow)$, and returns an SMT syntax tree τ'_{φ_υ}, such that, upon replacing τ'_{φ_μ} with τ'_{φ_υ} in τ'_φ, τ'_φ can sufficiently evaluate $(E', \rightsquigarrow)$. In other words, the STL representation of τ'_φ becomes the desired progression of φ on $(E, \rightsquigarrow)$. However, before defining υ, we specify the following shorthand notations we will be using throughout its definition:

- '$\tau_\varphi = p$': The root of the tree τ_φ is labelled $p \in \mathsf{AP}$.
- $\tau_\varphi = \tau_{\varphi_1} X \tau_{\varphi_2}$, where $X = \{\wedge, \vee\}$: The root of the tree τ_φ is labelled X, and it has two children τ_{φ_1} and τ_{φ_2}.
- $\tau_\varphi = \Box_{[a,b]}\, \tau_\psi$: The root of the tree τ_φ contains label $\forall i \in [a, b]$, and it has a child τ_ψ.
- $\tau_\varphi = \Diamond_{[a,b]}\, \tau_\psi$: The root of the tree τ_φ contains label $\exists i \in [a, b]$, and it has a child τ_ψ.
- $((E, \rightsquigarrow), t) \models \tau_\varphi$: At time instance t, $FinalSMT$ for $(E, \rightsquigarrow)$ and τ_φ is satisfiable.

Now we define υ in a case-by-case manner for the relevant STL operators:

Atomic propositions. Let $\tau_{\varphi_\mu} = p$ for some $p \in \mathsf{AP}$. We have:

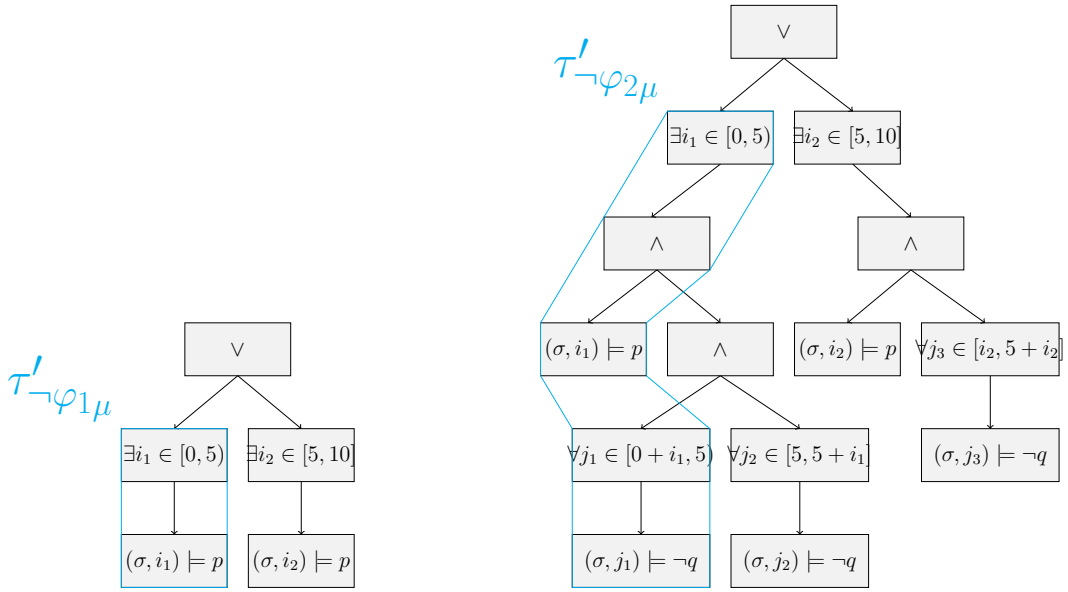
$$\upsilon((E, \rightsquigarrow), \tau_{\varphi_\mu}) = \begin{cases} \top & \text{if} \quad ((E, \rightsquigarrow), 0) \models p \\ \bot & \text{otherwise} \end{cases}$$

Conjunction. Let $\tau_{\varphi_\mu} = \tau_{\varphi_{\mu_1}} \wedge \tau_{\varphi_{\mu_2}}$. We have:

$$\upsilon((E, \rightsquigarrow), \tau_{\varphi_\mu}) = \upsilon((E, \rightsquigarrow), \tau_{\varphi_{\mu_1}}) \wedge \upsilon((E, \rightsquigarrow), \tau_{\varphi_{\mu_2}})$$

Disjunction. Let $\tau_{\varphi_\mu} = \tau_{\varphi_{\mu_1}} \vee \tau_{\varphi_{\mu_2}}$. We have:

$$\upsilon((E, \rightsquigarrow), \tau_{\varphi_\mu}) = \upsilon((E, \rightsquigarrow), \tau_{\varphi_{\mu_1}}) \vee \upsilon((E, \rightsquigarrow), \tau_{\varphi_{\mu_2}})$$



(a) Partitioned SMT syntax tree for $\tau'_{\neg\varphi_1}$. (b) Partitioned SMT syntax tree for $\tau'_{\neg\varphi_2}$.

Figure 5.4 Examples of partitioned SMT syntax tree of STL formulas $\neg\varphi_1$ and $\neg\varphi_2$ at $t = 5$.

Always operator. Let $\tau_{\varphi_\mu} = \Box\tau_{\varphi'_\mu}$. In this case, the transformation of τ_{φ_μ} is fairly straightforward:

$$\upsilon((E, \rightsquigarrow), \tau_{\varphi_\mu}) = \begin{cases} \Box_{[a,b]}\, \tau_{\varphi'_\mu} & \text{if} \quad \forall k \in [a,b].((E, \rightsquigarrow), k) \models \tau_{\varphi'_\mu} \\ \bot & \text{if} \quad \exists k \in [a,b].((E, \rightsquigarrow), k) \not\models \tau_{\varphi'_\mu} \end{cases}$$

Eventually operator. Let $\tau_{\varphi_\mu} = \Diamond\tau_{\varphi'_\mu}$. In this case, instead of finding a single time instance where $FinalSMT$ for $(E, \rightsquigarrow)$ and $\tau_{\varphi'_\mu}$ is satisfiable, a valid range $[k, b]$ must be identified, where $k \in [a, b]$ is the earliest time instance where $FinalSMT$ for $(E, \rightsquigarrow)$ and $\tau_{\varphi'_\mu}$ is satisfiable:

$$\upsilon((E, \rightsquigarrow), \tau_{\varphi_\mu}) = \begin{cases} \Diamond_{[k,b]}\, \tau_{\varphi'_\mu} & \text{if} \quad \operatorname{argmin}_{k\in[a,b]}(((E, \rightsquigarrow), k) \models \tau_{\varphi'_\mu}) \\ \bot & \text{if} \quad \forall k \in [a,b].((E, \rightsquigarrow), k) \not\models \tau_{\varphi'_\mu} \end{cases}$$

Remark 4. Since Until (Figure 5.2a) and Release (Figure 5.2b) operators are expressed using existential and global quantifiers in SMT syntax trees, the definition of υ does not need cases for them.

Now that we have defined υ, we state the necessary steps required to compute the progression of some STL formula φ on a distributed signal $(E, \rightsquigarrow)$ as follows:

- First, we create the SMT syntax tree τ_φ that corresponds to the STL formula φ using the methods detailed in Figure 5.2. As examples, let us consider the SMT syntax trees for the STL formulas, $\neg\varphi_1 = \Diamond_{[0,10]}\, p$ (Figure 5.3a) and $\neg\varphi_2 = \Diamond_{[0,10]}(p \wedge \Box_{[0,5]}\, \neg q)$ (Figure 5.3b).
- Next, we partition τ_φ at time $|(E, \rightsquigarrow)|$ using Algorithm 5.1, and obtain $\tau'_\varphi = \Lambda(\tau_\varphi, |(E, \rightsquigarrow)|)$, such that τ_{φ_μ} is the subtree in τ_φ that can be evaluated on $(E, \rightsquigarrow)$. In our example, we consider the case where the monitor only has the first 5 time units, that is, $|(E, \rightsquigarrow)| = 5$. Figure 5.4a (resp., Figure 5.4b) shows the partitioned SMT syntax tree for Figure 5.3b (resp., Figure 5.3b) at time instance $|(E, \rightsquigarrow)| = 5$ with subtrees $\tau'_{\neg\varphi_{1_\mu}}$ (resp., $\tau'_{\neg\varphi_{2_\mu}}$) that can be evaluated on $(E, \rightsquigarrow)$.
- Finally, we partially evaluate φ on $(E, \rightsquigarrow)$ by transforming τ'_{φ_μ} to $\tau'_{\varphi_\upsilon} = \upsilon((E, \rightsquigarrow), \tau'_{\varphi_\mu})$. The STL representation of this new SMT syntax tree τ'_φ is our desired progression of φ on the extension of $(E, \rightsquigarrow)$. In our first example, $\neg\varphi_{1_p}$ is of the form $\Diamond_{[0,5]}\, \neg\varphi'_{1_p}$. Now, let us assume that p is never `true` in $(E, \rightsquigarrow)$. In that case, according to the rules specified for υ, The label of the root of $\tau'_{\neg\varphi_{1_\mu}}$ stays unchanged, and the child becomes `false`. Therefore, the progression becomes $(\Diamond_{[0,5]}\, \texttt{false}) \vee (\Diamond_{[5,10]}\, p)$, which is, $\Diamond_{[5,10]}\, p$ upon simplification. In our second example, $\neg\varphi_{2_p}$ is of the form $\Diamond_{[0,5]}\, \neg\varphi'_{2_p}$. Now, let us assume that the minimum i for which $\exists i \in [0, 5)((((E, \rightsquigarrow), i) \models p) \wedge (\forall j \in [i + 0, \min(i + 5, 5)](((E, \rightsquigarrow), i) \models \neg q)))$ is satisfied at time 3.5. In that case, according to the rules specified for υ, The label of the root of $\tau'_{\neg\varphi_{2_\mu}}$ is changed to $\exists i_1 \in [3.5, 5)$. Therefore, the progression becomes $(\Diamond_{[3.5,5)}(\Box_{[0,5]}\, \neg q)) \vee (\Diamond_{[5,10]}(p \wedge (\Box_{[0,5]}\, \neg q)))$.

5.4 Case Studies and Evaluation

In this section, we evaluate our algorithm for monitoring STL specifications on distributed signals using two case studies.

5.4.1 Case Study 1: Network of UAVs

In a similar manner as Section 4.5, we use the Fly-by-Logic framework [100], a path planner software for UAVs, to simulate flight path of two UAVs that take off after $1.5s$,

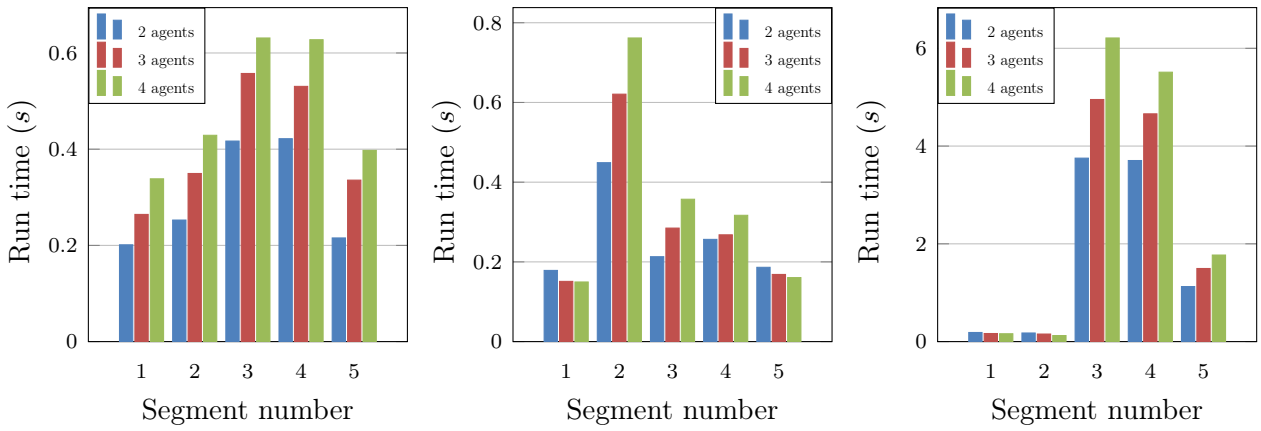
hover, and then land after $4.5s$. The trajectories are sampled at $20Hz$ as x_n, y_n, and z_n coordinates for each UAV A_n, with an ε ranging between $1 - 5ms$.

5.4.2 Case Study 2: Water Distribution System

We use the same model of a hybrid dynamic high pressure water distribution system consisting of two water tanks that we used in Section 4.5. Therefore, the specifications of the water tank model is identical to that of mentioned above. We use an ε range of $5-500ms$. However, despite using the same model, we will be verifying the system against STL, and observe different results from what we have witnessed in Chapter 4.

5.4.3 Experimental Setup

In our UAV related experiments, we monitor three STL properties: (1) mutual separation between UAVs never falls below a threshold; (2) all UAVs take off simultaneously from standby state and hover at the same altitude, and (3) all UAVs eventually land simultaneously. The monitor receives a distributed signal every second, and we measure its execution time for each formula progression to verify truthfulness of the given formulas. In our water tank related experiments, we simulate a plant failure where the RWST in the ECCS is triggered upon receiving an emergency actuation signal. The monitor receives a distributed signal at varying time intervals from multiple water tanks. Our goal is to find possible violations caused by clock drift, where the water pressure falls below threshold required to keep the failsafe CLA from triggering. In other words, we want to monitor the property during an emergency, when the outflow pressure reaches above the threshold pressure and remains above the threshold pressure forever. All experiments are replicated to exhibit 95% confidence interval to provide statistical significance. The experimental platform is a CentOS server with an Intel(R) Xeon(R) Platinum 8180 CPU @ 3.80GHz clock rate and 754G of RAM. Our implementation invokes the SMT-solver Z3 [97] to solve the problem described in Section 4.3.



(a) φ_{ms} (Mutual separation). (b) φ_{eh} (Eventually hover). (c) φ_{el} (Eventually land).

Figure 5.5 Effect of number of segments and agents on run time for different flight properties.

5.4.4 Analysis of Results

Mutual separation. This property states that the distance between every pair of UAVs in fleet always remain above a given threshold δ. The corresponding STL formula φ_{ms} is:

$$\bigwedge_{i,j\in[N],i\neq j} \square_{[0,\infty]} \left(\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} > \delta\right).$$

Figure 5.5a shows the run time for each segment for evaluation of φ_{ms} on the distributed signal. In each segment the progression formula remains unchanged. However, the first segment shows minimal run time due to the fact that the UAVs are stationary throughout the entirety of that segment and, therefore, require very few 'unique' distance calculations. The run time for the second segment and the last segment are slightly higher than that of the first segment because of the same reason; the UAVs are partially grounded throughout these two segments. Note that despite φ_{ms} seemingly being a simple STL formula, the average run time per segment is relatively higher (compared to the run time of other formulas) due to requiring quadratic equations to be solved.

Eventually hover. This property states that the UAVs in fleet are eventually (within $2s$) airborne and hover within a λ height margin. Formally, the corresponding STL formula φ_{eh}

is:

$$\bigwedge_{i,j\in[N],i\neq j} \Diamond_{[0,2]} \Big(z_i, z_j > 0\Big) \Rightarrow \Box_{[0,\infty]} \Big(|z_i - z_j| < \lambda)\Big).$$

Figure 5.5b shows the run time for each segment for evaluation of φ_{eh} on the distributed signal. The first segment has the lowest run time as the UAVs are stationary. The second segment has a higher run time because $(z_i, z_j > 0)$ is observed and progression is needed for the following segments, where the progressed formula simply becomes $\Box_{[0,\infty]}(z_i = z_j)$.

Eventually land This property states that the UAVs in fleet eventually land on the ground simultaneously. Formally, the corresponding STL formula φ_{el} is:

$$\bigwedge_{i,j\in[N],i\neq j} \Diamond_{[2,\infty]} \Big(z_i = 0 \wedge z_j = 0\Big).$$

Figure 5.5c shows the run time for each segment for evaluation of φ_{el} on the distributed signal. The temporal interval of φ_{el} is intentionally $[2, \infty]$ instead of $[0, \infty]$ since the UAVs are on the ground at the start of the distributed signal. The behavior in run time shown in this figure is opposite of what we have witnessed in Figure 5.5b. In segments 3 and 4, the UAVs are airborne, and therefore, the search-space for the SMT problem is exhaustively traversed. However, in segment 5, φ_{el} is satisfied and the progression becomes `true`.

Impact of segment duration and number of water tanks. Let $\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_N$ denote the outflow pressures of N number of water tanks. For simplicity, we assume all the pipes are of the same diameter. Thus, the pressure exerted on the CLA is $\mathbf{P}_1 + \mathbf{P}_2 + \ldots + \mathbf{P}_N$. We monitor the property that states outflow pressure remains above the threshold pressure $600psig$ [117] indefinitely. The corresponding STL formula $\varphi_{\mathbf{P}}$ is:

$$\Box_{[0,\infty]} \Big(\sum_{n=1}^{N} P_n \geq 600\Big).$$

Figure 5.6 shows the effect on run time for increasing the number of tanks from 2 to 4 with $\varepsilon = 0.05s$ over segment duration ranging from $1s$ to $5s$. As expected, both segment duration and the number of tanks drive up the run time. We note that even when the monitor receives

Clock Skew (s)	True Violations	Detected Violations	False Positives	False +ve Percentage
0.05	9	25	16	64%
0.1	4	42	38	90.48%
0.15	12	65	53	81.54%
0.2	11	80	69	86.25%
0.25	4	86	82	95.35%
0.3	7	99	92	92.93%
0.35	5	112	107	95.54%
0.4	7	127	120	94.49%
0.45	10	145	135	93.1%
0.5	7	160	153	95.63%

(a) Water tanks.

Clock Skew (s)	True Violations	Detected Violations	False Positives	False +ve Percentage
0.05	6	11	5	45.45%
0.1	6	20	14	70%
0.15	8	30	22	73.33%
0.2	4	39	35	89.74%
0.25	2	46	44	95.65%
0.3	1	48	47	97.92%
0.35	7	62	55	88.71%
0.4	2	66	64	96.97%
0.45	5	76	71	93.42%
0.5	6	84	78	92.86%

(b) UAVs.

Table 5.1 Impact of ε.

the distributed signals sent by the water tanks at a reasonable $1s$ intervals, the monitor is still able to verify the property online under around half a second for four tanks.

Impact of clock skew. In order to study the impact of ε on monitoring verdicts, we model two RWST modules with intentional 'faults', where the outflow pressures of either tank can drop below the threshold pressure of the CLA. Thus, if both tanks' pressures fall simultaneously, the CLA gets triggered. We also introduce a clock drift in the valve controller of one of the tanks. Table 5.1a shows the results for two tanks that were active for an hour. During this time, Tank 1 and Tank 2 reported low pressures for a total of $35.5s$ and $36.1s$ respectively. Although generally we are interested in finding a single violation, in order to

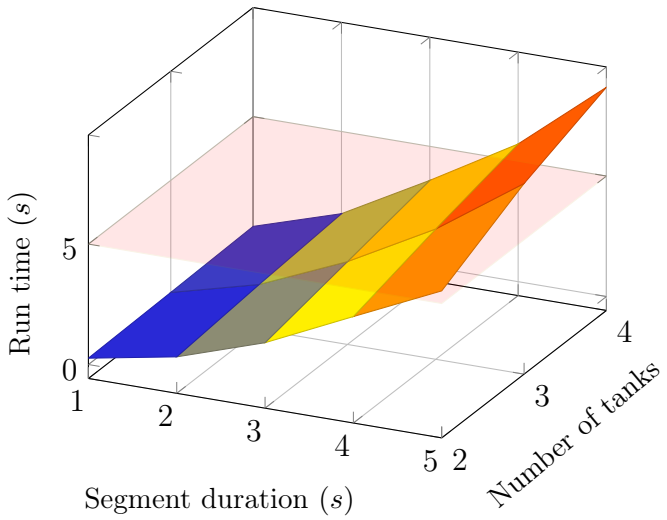


Figure 5.6 Effect of segment duration and the number of water tanks on runtime for $\varphi_{\mathbf{P}}$.

demonstrate the effect of clock skew, we find multiple violation instances in this experiment by tallying up pairs of piece-wise linear interpolations between samples where violations are detected. We report the number of *true violations* as a baseline that was reverse calculated from the introduced clock drift ε, number of *detected violations* using our method, and the number of *false positives*, which is essentially the difference between the true violations and the detected violations[1]. Note that there are no *false negatives*. Furthermore, as the clock drift is increased from $0.05s$ to $0.5s$, the number of false positives increase as well. Similarly, we model a path for a pair of UAVs, where the agents periodically reside within the given mutual separation threshold, and violate the mutual separation property. Table 5.1b shows the results for two UAVs in operation for half an hour. We again report the number of true violations, detected violations, and false positives.

5.5 Conclusion

In this chapter, we developed a mechanism for monitoring requirements defined in signal temporal logic (STL) for distributed CPS, where continuous-time and continuous-valued signals from a group of agents do not share a global clock. Our method relies on an off-the-shelf clock synchronization algorithm, such as NTP, to ensure a maximum constrained clock

[1]We emphasize that due to the uncertainty caused by asynchrony, the existence of false positives is inevitable, as there is no global clock to ensure a total order of events.

skew across all agents in the system. We also presented a signal retiming approach, borrowed from Chapter 4, that effectively aligns continuous signals in order to detect potential STL specification breaches. To address the complexity, we compress our runtime monitoring problem to an SMT solution problem and cut the distributed signals into a series of smaller parts. To that purpose, similar to that of Chapter 3, we presented a formula progression approach that takes a distributed signal and an STL formula as input and outputs another STL formula that depicts the formula's progress over the signals.

Furthermore, in Section 5.4, we presented experimental results from the monitoring of an unmanned aerial vehicle (UAV) fleet, and a water distribution system. These experiments indicate that for certain cases, it is indeed possible to monitor STL formulas online on a distributed signal using our technique.

CHAPTER 6

DECENTRALIZED PREDICATE DETECTION OVER PARTIALLY SYNCHRONOUS CONTINUOUS-TIME SIGNALS

In this chapter, we set our sights toward decentralized monitoring of distributed CPS. The natural first step would therefore be decentralized monitoring of predicate violations over continuous-time and continuous-valued signals under partial synchrony.

To this end, we propose a decentralized monitoring algorithm to detect *all Boolean predicates* over the *analog* (i.e. continuous-time and continuous-valued) signals generated by the agents in a distributed CPS. Similar to our approaches described in previous chapters, a clock synchronization algorithm (see Subsection 2.3) guarantees a maximum clock skew across all signals generated by the agents.

It is helpful to overview our algorithm and key notions via an example before delving into the technical details. An example is shown in Figure 6.1. Three agents produce three signals x_1, x_2, x_3. The decentralized detector consists of three local detectors D_1, D_2, D_3, one on each agent. Each x_n is observed by the corresponding D_n. The predicate $\phi = (x_1 \geq 0) \wedge (x_2 \geq 0) \wedge (x_3 \geq 0)$ is being detected. It is true over the intervals shown with solid black bars; their endpoints are measured on the local clocks. The detector only knows that the maximal clock skew is $\epsilon = 1$, but not the actual value, which might be time-varying.

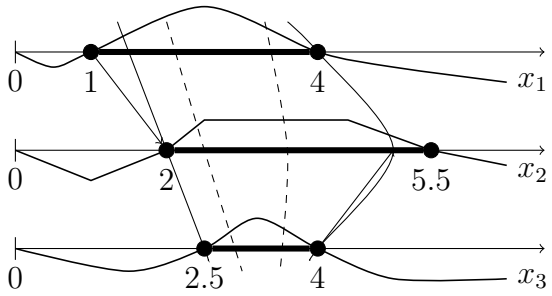


Figure 6.1 An example of a continuous-time distributed signal with 3 agents. Three timelines are shown, one per agent. The signals x_n are also shown, and the local time intervals over which they are non-negative are solid black. The skew ϵ is 1. The Happened-before relation is illustrated with solid arrows, e.g. between $e_1^1 \rightsquigarrow e_2^2$, and $e_3^4 \rightsquigarrow e_2^5$. Some satisfying cuts for the predicate $\phi = (x_1 \geq 0) \wedge (x_2 \geq 0) \wedge (x_3 \geq 0)$ are shown as dashed arcs, and the extremal cuts as solid arcs. All extremal cuts contain root events, and leftmost cut A also contains non-root events.

Because of clock skew, any two local times within ϵ of each other must be considered as potentially concurrent, i.e. they might be measured at a truly synchronous moment. For example, consistent cut at local times $[4, 4.5, 3.6]$ might have been measured at the global time 4, in case the true skews were 0,0.5 and -0.4 respectively. The detector's task is to find all consistent cuts that satisfy the predicate. In continuous time, there can be uncountably many, as in Figure 6.1; the dashed lines show two satisfying consistent cuts, or satcuts for short.

In this example, the detector outputs two satcuts, $[1.5, 2, 2.5]$ and $[4, 5, 4]$, shown as thin solid lines. These two have the special property (shown in this chapter) that every satcut lies between them, and every cut between them is a satcut. For this reason we call them *extremal satcuts*, which is formally defined later. Thus these two satcuts are a finite representation of the uncountable set of satcuts, and encode all the ways in which the predicate might be satisfied.

We note three further things: the extremal satcuts are not just the endpoints of the intervals, and simply inflating each interval by ϵ and intersecting them does not yield the satcuts. Each local detector must somehow learn of the relevant events (and only those) on other agents, to determine whether they constitute extremal satcuts.

In the following sections, we state some necessary technical definitions, establish fundamental properties of the uncountable set of events satisfying the predicate, methodology for computing finite representation of the uncountable of events, complexity analysis, and finally, implementation and experiments.

6.1 Problem Statement

Before we state the problem statement, we define the class of predicates we monitor using our decentralized monitoring algorithm. This chapter focuses on specifications expressible as *conjunctive predicates* φ, which are conjunctions of N linear inequalities.

$$\varphi := (x_1 \geq 0) \wedge (x_2 \geq 0) \wedge \ldots \wedge (x_N \geq 0). \tag{6.1}$$

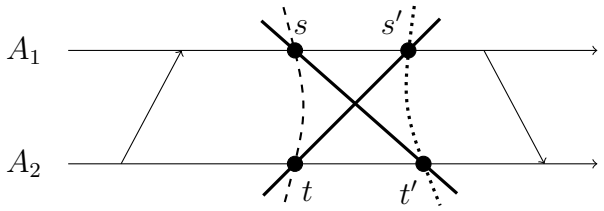


Figure 6.2 Two satcuts for a pair of agents A_1 and A_2, shown by the crossed solid lines (s, t') and (s', t). Their intersection is (s, t), shown by a dashed arc, and their union is (s', t'), shown by a dotted arc. For a conjunctive predicate φ, the intersection and union are also satcuts, forming a lattice of satcuts.

These predicates model the simultaneous co-occurrence, in global time, of events of interest, like 'all drones are dangerously close to each other'. Equation 6.1 also captures the cases where some conjuncts are of the form $x_n \leq 0$ and $x_n = 0$. If N numbers (a_n) satisfy predicate φ (i.e., are all non-negative), we write this as $(a_1, \ldots, a_N) \models \varphi$. Henceforth, we say 'predicate' to mean a conjunctive predicate in this chapter.

Definition 15. [Distributed Satisfaction; S_E] Given a predicate φ, a distributed signal $(E, \rightsquigarrow)$ over N agents, and a consistent cut C of E with frontier

$$\mathsf{front}(C) = \Big((t_1, x_1(t_1)), \ldots, (t_N, x_N(t_N)) \Big)$$

we say that C *satisfies* φ iff $\big(x_1(t_1), x_2(t_2), \ldots, x_N(t_N)\big) \models \varphi$. We write this as $C \models \varphi$, and say that C is a *satcut*. The set of all satcuts in E is written S_E. ■

6.1.1 Decentralized Predicate Detection

As stated before, our algorithm seeks to find *all* possible global states that satisfy a given predicate, i.e. all satcuts in S_E. In general, S_E is uncountable.

Architecture. The system consists of N agents with partially synchronous clocks with drift bounded by a known ϵ, generating a continuous-time distributed signal $(E, \rightsquigarrow)$. Agents communicate in a FIFO manner.

Problem Statement

Given $(E, \rightsquigarrow)$ and a conjunctive predicate φ, find a decentralized detection algorithm that computes a finite representation of S_E. The detector is decentralized, meaning that it consists of N local detectors, one on each agent, with access only to the local signal x_n (measured against the local clock), and to messages received from other agents' detectors.

By computing a representation of all of S_E (and not some subset), we account for asynchrony and the unknown orderings of events within ϵ of each other. One might be tempted to propose something like the following algorithm: detect all roots on all agents, then see if any N of them are within ϵ of each other. This quickly runs into difficulties: first, a satisfying cut is not necessarily made up of roots; some or all of its events can be interior to the intervals where x_n's are positive (see Figure 6.2). Second, the relation between roots and satcuts must be established: it is not clear, for example, whether even satcuts made of only roots are enough to characterize all satcuts (it turns out, they are not). Third, we must carefully control how much information is shared between agents, to avoid the detector degenerating into a centralized solution where everyone shares everything with everyone else.

6.2 The Structure of Satisfying Cuts

We establish fundamental properties of satcuts. In the rest of this chapter we exclude the trivial case $C = E$. Proposition 1 mirrors a discrete-time result [26].

Proposition 1. The set of satcuts for a conjunctive predicate is a lattice where the join and meet are the union and intersection operations, respectively.

Proof. Define the intersection $I = C \cap C'$ and let e be an element of I. Then by definition of a cut, every event that happened-before e is in C and in C', and therefore is in their intersection, so I is a cut. The frontier of I is made of events $(t_n, x_n(t_n))$ such that $t_n = \max\{t \mid (t, x_n(t)) \in C \cap C'\}$. In words, $(t_n, x_n(t_n))$ is the last event on signal x_n belonging to both satcuts, which implies it is the last event on at least one of the cuts, say C. Therefore

$(t_n, x_n(t_n))$ is on the frontier of C, and so $x_n(t_n) \geq 0$ by definition of a conjunctive predicate. Since this is true for every n in $[N]$, we have that the frontier of I is a consistent state that satisfies the predicate, and so $I \models \varphi$. The union $C \cup C'$ is also a satcut by similar arguments, so the set of satcuts is a lattice. ■

We show that the set of satcuts is characterized by special elements, which we call the leftmost and rightmost cuts.

Definition 16. [Extremal cuts] Let S_E be the set of all satcuts in a given distributed signal $(E, \rightsquigarrow)$. For an arbitrary $C \in S_E$ with frontier $(e_n^{t_n})_n$ and positive real α, define $C - \alpha$ to be the set of cuts whose frontiers are given by

$$(e_1^{t_1-\delta_1}, e_2^{t_2-\delta_2}, \ldots, e_N^{t_N-\delta_N}) \text{ s.t. for all } \; n : 0 \leq \delta_n \leq \alpha \text{ and } \exists n.\ \delta_n > 0$$

A *leftmost satcut* is a satcut $C \in S_E$ for which there exists a positive real α s.t. $C - \alpha$ and S_E do not intersect. A *rightmost cut* C (not necessarily sat) is one for which there exists a positive real α s.t. $C + \alpha$ and S_E do not intersect, and $C - \alpha \subset S_E$. We refer to leftmost and rightmost (sat)cuts as *extremal cuts.* ■

Intuitively, $C - \alpha$ is the set of all cuts one obtains by slightly moving the frontier of C to the left by amounts less than α. If doing so always yield non-satisfying cuts, then C is a leftmost satcut. Analogous intuition applies to rightmost cuts. If signals x_n are all continuous, then rightmost cuts are all satisfying as well.

In a signal, there are multiple extremal cuts. Figure 6.2 suggests, and Lemma 10 proves, that all satcuts live between a leftmost satcut and rightmost cut.

Lemma 10. [Satcut intervals] Every satcut of a conjunctive predicate lies in-between a leftmost satcut and rightmost cut, and there are no non-satisfying cuts between a leftmost satcut and the first rightmost cut that is greater than it.

Proof. Let C be a satcut, so that $x_n(t_n) \geq 0$ for every $(t_n, x_n(t_n))$ in its frontier. Let s_n be the biggest shift backwards in time preserving positivity:

$$s_n := \sup\{s \mid s \geq 0 \text{ and } \forall\ 0 \leq \sigma \leq s.\ x_n(t_n - \sigma) \geq 0\}. \tag{6.2}$$

By the starvation-freedom assumption derived from Assumption 2.2, s_n is finite and by the right-continuity of x_n, $x_n(t_n - s_n) \geq 0$. Now the cut with frontier $(e_n^{t_n - s_n})_n$ satisfies the predicate, but might not be consistent because it could be that $|t_n - s_n - (t_m - s_m)| > \epsilon$ for some n, m. Suppose without loss of generality that $t_1 - s_1$ is the largest of all the $t_n - s_n$'s. Define $b_n = \max(t_n - s_n, t_1 - s_1 - \epsilon)$ for all $n > 1$. Note that $b_n \leq t_n$ because C is consistent ($t_1 - t_n \leq \epsilon$ so a fortiori $t_1 - t_n \leq \epsilon + s_1$ and so $b_n = t_1 - s_1 - \epsilon \leq t_n$, whereas the other case, $b_n = t_n - s_n$, is immediate), and $x_n(b_n) \geq 0$. Then the cut L with frontier $(e_1^{t_1 - s_1}, e_2^{b_2}, \ldots, e_N^{b_N})$ is consistent and satisfies the predicate. It is also leftmost by construction of s_1. Therefore L is a leftmost satcut.

The reasoning for rightmost cuts follows the above lines, except for predicate satisfaction. Namely: let s_n now be the biggest shift forwards in time preserving positivity:

$$s_n := \sup\{s \mid s \geq 0 \text{ and } \forall\ 0 \leq \sigma \leq s.\ x_n(t_n + \sigma) \geq 0\}. \tag{6.3}$$

By the starvation-freedom assumption derived from Assumption 2.2, s_n is finite. Now the cut with frontier $(e_n^{t_n + s_n})_n$ might not be consistent because it could be that $|t_n + s_n - (t_m + s_m)| > \epsilon$ for some n, m. Suppose without loss of generality that $t_1 + s_1$ is the smallest of all the $t_n + s_n$'s. Define $b_n = \min(t_n + s_n, t_1 + s_1 + \epsilon)$ for all $n > 1$. Note that $b_n \geq t_n$ because C is consistent. Then the cut R with frontier $(e_1^{t_1 + s_1}, e_2^{b_2}, \ldots, e_N^{b_N})$ is consistent, but does not necessarily satisfy the predicate because of possible discontinuities (Namely, if $t_n + s_n$ is a point of discontinuity for x_n then possibly $x_n(t_n + s_n) < 0$). R is also rightmost by construction of s_1 and the b_n. Therefore R is a rightmost cut.

Thus every satcut is between a leftmost satcut and rightmost cut. Also by construction of L and R (specifically, Equation 6.2 and 6.3), there is no cut in-between that does not satisfy the predicate. That is, there is no C s.t. $L \sqsubseteq C \sqsubseteq R$ and $C \not\models \varphi$. (Here $\sqsubseteq$ is the ordering relation on the lattice of cuts).

■

Thus we may visualize satcuts as forming N-dimensional intervals with endpoints given by the extremal cuts. The main result of this section states that there are finitely many extremal

satcuts in any bounded time interval, so the extremal satcuts are the finite representation we seek for S_E.

Theorem 6. A distributed signal has finitely many extremal satcuts in any bounded time interval.

In order to prove the above theorem, we will need the following definitions: the *leftmost event of a cut* C is an event $e_n^t \in \mathsf{front}(C)$ where $t \leq t'$ for all other events $e_m^{t'} \in \mathsf{front}(C)$. With β a real number, an event $e_m^{t'}$ is said to be β-offset from e_n^t if and only if $t' = t + \beta$.

We will need the following three lemmas.

Lemma 11. The leftmost event of a rightmost cut is a right root.

Proof. Consider the leftmost event e_n^t of a rightmost cut C. Because C is rightmost, then $x_n(t-\delta) \geq 0$ for all sufficiently small positive δ. Assume for a contradiction that e_n^t is not a right root, so $x_n(t+\alpha) \geq 0$ for all sufficiently small $\alpha \geq 0$, say all α strictly less than some $\overline{\alpha}$. Since e_n^t is leftmost, we can add the events $e_n^{t+\alpha}$, $0 \leq \alpha \leq \min\{\frac{\epsilon}{2}, \frac{\overline{\alpha}}{2}, \gamma\}$, to C to form a new cut C'. Choosing γ small enough guarantees that C' is consistent. C' is also satisfying because we only added events such that $x_n(t+\alpha) \geq 0$. This shows C is not a rightmost cut, which contradicts our choice of C. ■

Lemma 12. All events of the frontier of a rightmost cut are either right roots or ϵ-offset from a right root.

Proof. Let e_n^t be the leftmost event in the frontier of a rightmost cut C. By Lemma 11 this event is a right root. Now consider any other event $e_m^{t'} \in \mathsf{front}(C)$ which is *not* a right root, and assume for contradiction that $t' \neq t + \epsilon$. Then $x_m(t') \geq 0$ and (as in the proof of Lemma 11) $x_m(t'+\alpha) \geq 0$ for all sufficiently small α. If $t' < t + \epsilon$, then it is possible to add the events $\{e_m^{t'+\alpha} \mid \alpha \in [0, \gamma)\}$ to C, with γ small enough, to obtain a satcut to its immediate right, which contradicts C being rightmost. On the other hand if $t' > t + \epsilon$ this contradicts that e_n^t and $e_m^{t'}$ are part of the same frontier. Thus $t' = t + \epsilon$. ■

The next lemma (and its proof) parallels Lemma 11 and Lemma 12, but for leftmost satcuts.

Lemma 13. The rightmost event of a leftmost satcut is a left root. Moreover, every event of the frontier of a leftmost satcut is either a left root or is $(-\epsilon)$—offset from a left root.

Thus every extremal satcut has a left root or a right root as one its constituent events. Since there are only finitely many roots in any bounded interval, this gives us the desired conclusion.

Therefore, it is conceivably possible to recover algorithmically the extremal satcuts, and therefore all satcuts by Lemma 10. The rest of this chapter shows how.

6.3 The Abstractor Process

Having captured the structure of satcuts, we now define the distributed *abstractor process* that will turn our continuous-time problem into a discrete-time one, amenable to further processing by our modified version of the slicer algorithm of [26]. This abstractor also has the task of creating a happened-before relation. We first note a few complicating factors. First, this will not simply be a matter of sampling the roots of each signal. That is because extremal satcuts can contain non-root events, as shown in Figure 6.1. Thus the abstractor must somehow find and sample these non-root events as part of its operation. Second, as in the discrete case, we need a kind of clock that allows the local slicer to know the happened-before relation between events. The local timestamp of an event, and existing clock notions, are not adequate for this. Third, to establish the happened-before relation, there is a need to exchange event information between the processes, without degenerating everything into a centralized process (by sharing everything with everyone). This complicates the operation of the local abstractors, but allows us to cut the number of messages in half.

6.3.1 Abstractor Description

The abstractor is described in algorithm 6.1 on page 117. Its output is a stream of discrete-time events, their correct PVC values, and the relation $\rightsquigarrow$ between them - i.e., a

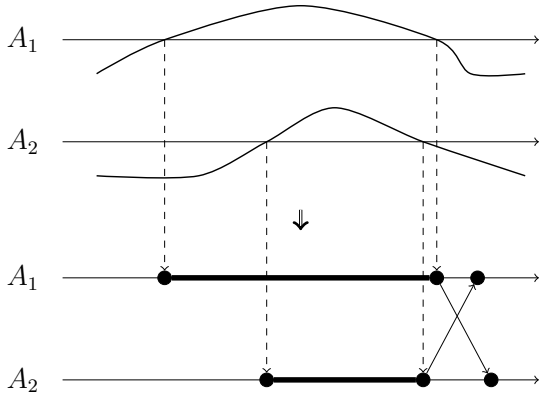


Figure 6.3 A distributed signal of two agents (top) and the output of the abstractor (bottom). The abstractor marks zero-crossings as discrete root events and creates new events (dark circles) to maintain consistency.

Data: Signal of agent A_n
Result: A stream of discrete events which are roots or ϵ-offset from roots

trigger *found a* root e_n^t *at local time* t**:**
- **add** e_n^t *info* (n, t, PVC, left or right root) to local buffer **if** e_n^t *is right root*:
  - **for each** *agent* $m \neq n$**:**
    - **send** e_n^t *info* to agent m

trigger *received message about* right root e_m^t *from agent* A_m**:**
- Set $t' := t + \epsilon$, where ϵ is the maximum clock skew **create** *local event* $e_n^{t'}$ **create** *relation* $e_m^t \rightsquigarrow e_n^{t'}$ (setting the PVC for $e_n^{t'}$ appropriately) **add** $e_n^{t'}$ *info* (n, t', PVC, left or right root) to local buffer `/* Visit events in the buffer, forwarding ones that are ready to the slicer. */`
- **for each** *event* e_n^s *in the local buffer*:
  - `/* Ready events are those whose PVCs will not be updated anymore. See text for details. */`
  - **if** A_n *received at least one message about a* right root $e_k^{t_k}$ *from every other agent* A_k *such that* $t_k \geq t$**:**
    - Set $\mathbf{v}_n^s[n] = s$ and $\mathbf{v}_n^s[k] = s - \epsilon$ for all $k \neq n$ Remove e_n^s from buffer and **send** it to local slicer

Algorithm 6.1 Local abstractor for agent A_n

discrete-time distributed signal. This signal is processed by the local slicers as it is being produced by the abstractor.

The abstractor runs as follows. It is decentralized, meaning that there is a *local* abstractor running on each agent. Agent A_n's local abstractor maintains a buffer of discrete events, and consists of two trigger processes. The first is triggered when a root is detected (by a

local zero-finding algorithm). It stores the root's information in a local buffer (for future processing). *If it is a right root*, it also sends it to the other agents. The second trigger process is triggered when the agent *receives* a right root information from some other process, at which point it does three things: it creates a local discrete event and a corresponding relation $\rightsquigarrow$ between events, it updates events in its local buffer to see which ones can be sent to the local slicer process (described later), and then it sends them. It is clear, by construction, that $\rightsquigarrow$ is a happened-before relation: it is the subset of $\rightsquigarrow$ needed for detection purposes.

Before an event e_n^t is sent to the slicer, it must have a PVC that correctly reflects the happened-before relation. This means that all events that happened-before e_n^t must be known to agent n, which uses them to update the PVC timestamps. This happens when events have reached agent A_n from every other agent, with timestamps that place them after e_n^t. This is guaranteed to happen by the starvation-free assumption.

The output of a local abstractor is a stream of discrete events, so that *the output of the decentralized abstractor as a whole is a distributed discrete-time signal.* See Figure 6.3.

Given that all right roots are assigned discrete events by the first trigger, and given that ϵ-offset events are also created from them by the second trigger,

Theorem 7. All events in rightmost cuts are returned by the abstractor. Moreover, a rightmost cut of E is also a cut of the discrete signal returned by the abstractor.

Thus the slicer process can find the rightmost satcuts when it processes the discrete signal. The leftmost satcuts will be handled by the slicer using the PVCs, as will be shown in the next section. Doing it this way relieves the abstractor from having to communicate the left roots between processes, thus saving on messages and the corresponding wait times.

6.4 The Slicer Process for Detecting Predicates

The second process in our detector is a decentralized *slicer process*, so-called to keep with the common terminology in discrete distributed systems [53]. The slicer is decentralized: it consists of N local slicers $\mathcal{S}_n$, one per agent. The slicer runs in parallel with the abstractor and processes the abstractor's output as it is produced. Recall that the abstractor's output

consists of a stream of discrete events, coming from the N agents. These events are either roots or ϵ-offset from roots. If an event is a left root or ϵ-offset from a left root, we will call it a left event. We define right events similarly. We will write F_n for those events, output by the abstractor, that occurred on A_n.

Every slicer $\mathcal{S}_n$ maintains a *token* T_n, which is a constant-size data structure to keep track of satcuts that contain A_n events. Specifically, for every event e_n^t in F_n, the token T_n is forwarded between the agents, collecting information to determine whether there exists a satcut that contains e_n^t. We say the slicer is trying to *complete* e_n^t. The token's updates are such that it will find that satcut if it exists, or determines that none exists; either way, it is then reset and sent back to its parent process A_n to handle the next event in F_n.

Let e_n^t be an event that the slicer is currently trying to complete. The token's updates vary, depending on whether it is currently completing a left event, or a right event. *If T_n is completing a right event*, the token is updated as follows. The token currently has a cut whose frontier contains e_n^t, which is either a satcut or not. If it is, the token has successfully completed the event and is returned to A_n to handle the next event in F_n. If not, then by the property of *regular* predicates [26], there exists a *forbidden event* e_m^s on the frontier of the cut which either prevents the cut from being consistent or from satisfying the predicate. T_n is sent to the process A_m containing this forbidden event. T_n's so-called target event, whose inclusion may give T_n a satcut, is the event on A_m following the forbidden e_m^s. If the token does not find a next event following e_m^s, then the token is kept by $\mathcal{S}_m$ until it receives the next event from the abstractor (which is guaranteed to happen under the starvation-free assumption). After the token retrieves the next event, the updates to the token and progression of $\mathcal{S}_n$ then follow the CGNM slicer [26]. Space limitations make it impossible to describe the CGNM slicer here, and we refer the reader to the detailed description in [26].

If handling a left event, the token is updated as follows. First, as before, T_n is sent to the process A_m which generates the forbidden e_m^s – i.e., which prevents T_n from completing e_n^t. T_n's target event may not be the next event on that process following e_m^s: this is because

if e_n^t is a left root, there may exist a left event $e_m^{t-\epsilon}$ on A_m which is part of a continuous-time leftmost satcut (by Definition 7), but which was not created by the abstractor. In this case, if the token were to follow the updates for a right event, it would skip a potential satcut. Instead, the slicer $\mathcal{S}_m$ will create this event: namely, if $\mathcal{S}_m$ sees a new event $e_m^{s'}$ where $s' > t - \epsilon$, it knows that $e_m^{t-\epsilon}$ has not and will not show up (will not be produced by the abstractor) because messages are FIFO. The slicer at this point creates the new event $e_m^{t-\epsilon}$. This is valid since in continuous-time, by definition, every moment has a corresponding event on every agent. Once the token retrieves this created $e_m^{t-\epsilon}$ as its new target, the updates to the token and progression of $\mathcal{S}_n$ follow the CGNM slicer [26], similarly to the right event scenario.

Correctness of $\mathcal{S}$. We will show that all extremal cuts of the continuous-time signal are included in the discrete lattice. Since the CGNM slicer computes the discrete lattice, this means in particular that it computes the extremal cuts that are in it. From these extremal cuts, we can then recover the continuous-time satcuts by Lemma 10.

Lemma 14. For all events e_n^t that are left roots, the token T_n incorporates all $e_m^{t-\epsilon}$ for all $m \neq n$.

Proof. For a left root e_n^t, by Theorem 2 its PVC is $\mathbf{v}_n^t = [t-\epsilon, \ldots, t-\epsilon, t, t-\epsilon, \ldots, t-\epsilon]$. Since a token T_n is tasked with identifying consistent cuts, for each $m \neq n$ it must incorporate the leastmost event on A_m which can form a consistent cut with e_n^t. The PVC identifies this event as $e_m^{t-\epsilon}$. Therefore, T_n incorporates all $e_m^{t-\epsilon}$ events where e_n^t is a left root on A_n. ■

Lemma 15. The modified slicer processes all events of a leftmost satcut.

Proof. By Lemma 13, all events of a leftmost satcut are either at time t or $t - \epsilon$, where t is the time of a left root. Since by Lemma 14 every token T_n will visit the $t - \epsilon$ event for any left root at t on A_n, every $t - \epsilon$ will be processed for any left root. Thus, all events of a leftmost satcut will be processed. ■

Theorem 8. Our slicer returns all extremal cuts.

Proof. The abstractor creates discrete events for all roots, as well as ϵ-offsets from right roots. By Lemma 15, the slicer creates all events of a leftmost satcut. This means that all events of leftmost and rightmost satcuts are processed by the slicer. Therefore, since the modified slicer returns a lattice of satcuts, the extremal satcuts are included. ■

We give the space and time complexity of the overall detector.

Since this is an online detector which runs forever (as long as the system is alive), we must fix a time interval for the analysis.

Theorem 9. The time complexity for each agent is $O(2RN)$, where R is the number of right roots in the given analysis interval. The detector consumes $O(N^3)$ memory to store the tokens. If roots are uniformly distributed, then the local buffers of the abstractor and slicer grow at the most to size $O(N^2)$.

Proof. We distinguish the following cases:

Time complexity. The calculations in our algorithm come from the abstractor, and the modification to the CGNM slicer. Finding a root of a signal x_n takes constant time in the system parameters. The abstractor has every process send right root info to every other process, for a complexity of $N-1$ per right root, and total complexity of $(N-1)R$ where R is the number of right roots in the system in a given bounded window of time.

Consider slicer $\mathcal{S}_n$, which is hosting token T_m. The slicer creates a new event, for every target event of T_m that was not produced by the abstractor of A_m. Event creation is $O(N)$ since it requires the creation of a size-N PVC assigned to the event. Event storage takes constant time if the new event is simply appended at the end of the local buffer, or $O(k)$ if the event is inserted in-order in the sorted local buffer of size k. Either one works: the first one is cheaper, but an unsorted buffer costs more to find events in it. The latter is more expensive up-front, but the sorted buffer can be searched faster. Either way, the slicer modification costs a total of $O(N \cdot M)$ in a given bounded window of time with M missed events in the system.

Now the number of target events requiring creation is on the order of the number of right roots since they result from left roots, and there are equal numbers of left and right roots. Thus $M = O(R)$. Therefore, the total complexity for our algorithm in a given bounded window of time is $O(R(N-1+N))$. Of course, this is then added to the complexity of running the modified slicer, which is $O(N^2D)$, where D is the number of events in the discrete-time signal. At the most, there are $2R$ events. So finally the total time complexity is $O(R(N-1+N)+2N^2R)$, or $O(R(2N+2N^2)/N) = O(2RN)$ per agent.

Space complexity. Indeed, a PVC timestamp has size $O(N)$ (since it is an N-dimensional vector). This is in fact the optimal complexity for characterizing causality [23]. One token stores N PVCs at all times and token updates replace old PVC values by new ones. Therefore one token has size $O(N^2)$, and all N tokens (one per agent) require $O(N^3)$ space.

How long events stay in the abstractor's local buffers depends on message transmission times, since events are removed from the buffers after the appropriate messages are received (see Algorithm 6.1). It also depends on the distribution of events within the interval of analysis, not just their rate $1/R$. E.g. if roots are uniformly distributed in the analysis interval, then the n^{th} abstractor's local buffer grows at the most to size $O(N^2)$, as it receives roots from the other $N-1$ agents and stores the $O(N)$ PVC timestamp for each root. Then event removal starts as A_n receives target events. Similar considerations apply to the slicer's local buffers. In such a case the detector's total space complexity is $O(N^3+2N^2)$. ■

Finally, there is no bound on detection delay, since we do not assume any bounds on message transmission time. Assuming some bound on transmission delay easily yields a corresponding bound on detection delay.

6.4.1 Worked-out example

We now work through an example execution of the detector on Figure 6.4. We focus on agent A_2, its abstractor $\mathcal{A}_2$, slicer $\mathcal{S}_2$ and its token T_2.

1. Agent A_2 encounters a left root in the signal at local time 3.5. This information is

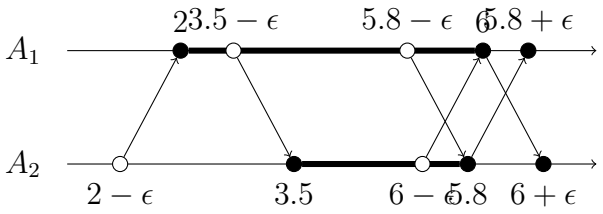


Figure 6.4 Example of subsection 6.4.1. Bold intervals are where the local signals are non-negative. The happened-before relation is illustrated with solid arrows. The predicate is $\phi = (x_1 \geq 0) \wedge (x_2 \geq 0)$. Solid circles represent discrete events returned by the abstractor; hollow circles are those created by the slicers. The leftmost satcut of this example is $[3.5 - \epsilon, 3.5]$ and the rightmost is $[6, 5.8]$.

forwarded to the abstractor.

2. The abstractor $\mathcal{A}_2$ adds the new root to its buffer with a PVC $=[3.5 - \epsilon, 3.5]$.

3. A_2 finds a right root in the signal at local time 5.8 and forwards it to $\mathcal{A}_2$.

4. The abstractor sends the root information to agent A_1. It then adds this root to its buffer with a PVC timestamp of $[5.8 - \epsilon, 5.8]$.

5. Abstractor $\mathcal{A}_2$ receives a message from A_1 about a right root at A_1's local time 6. Note that this is the first knowledge A_2 has about anything that is occurring on A_1, even though A_1 has already found a left root.

6. $\mathcal{A}_2$ uses A_1's message to create a new local event at $6 + \epsilon$ with PVC $[6, 6 + \epsilon]$.

7. $\mathcal{A}_2$ also adds this new local event to its buffer. Since all messages are FIFO, A_2 knows that there will be no new messages which will create events before $6 + \epsilon$. Thus, it can remove both of the events 3.5 and 5.8 from the buffer and forward them to its local slicer $\mathcal{S}_2$. At this point both of A_1's events have been forwarded to *its* slicer, although A_2 has no knowledge of this.

8. The slicer $\mathcal{S}_2$ receives an event with a PVC $[3.5 - \epsilon, 3.5]$. Token T_2 is waiting for the next event, so it adds this event to its potential cut.

9. The token is processed with its new potential cut. The cut is found to be inconsistent since T_2 has no information about any A_1 events.

10. The token's target is set to be $3.5 - \epsilon$ on A_1 and the token is sent to A_1.

11. A_1 receives T_2. It walks through its local events 2 and 6 and determines that T_2's target event is between the two.

12. $\mathcal{S}_1$ creates a new event $e_2^{3.5-\epsilon}$ and notes that $x_2(3.5 - \epsilon) \geq 0$.

13. Token T_2 incorporates the new event to its potential cut. The new potential cut is consistent and satisfies the predicate. It is then sent back to A_2.

14. A_2 receives T_2. T_2 indicates a satisfying cut, which the agent outputs as a result. It then advances T_2 to its next event at time 5.8.

15. T_2 has the current cut of $[3.5 - \epsilon, 5.8]$. This is not consistent, so it is given the target $5.8 - \epsilon$ on A_1. It is then sent to A_1.

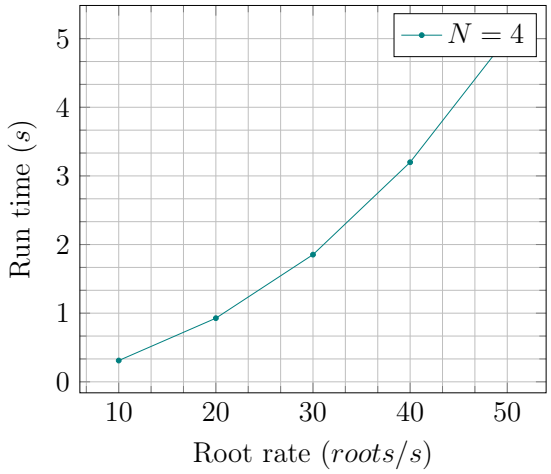
16. A_1 receives the token. $\mathcal{S}_1$ walks through its local events and finds that the token's target is between the left root and the right root.

17. $\mathcal{S}_1$ creates a new event at $5.8 - \epsilon$ and notes that $x_1(5.8 - \epsilon) \geq 0$.

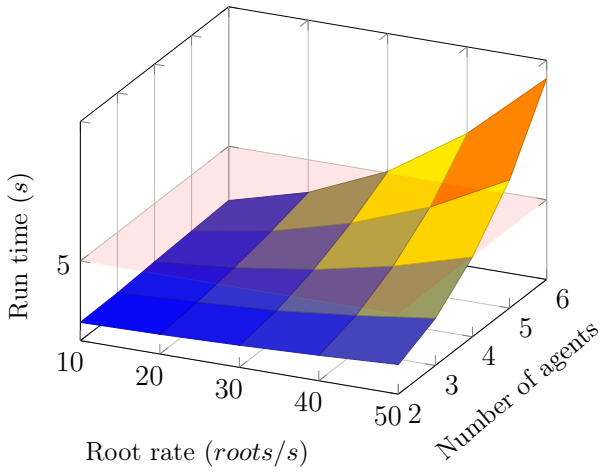
18. The token adds the event to its potential cut. It finds that its new potential cut is consistent and satisfies the predicate. It is then sent back to A_2.

19. A_2 receives T_2 and outputs the satcut. The algorithm then continues with new events as they occur.

Through this example, agent A_2 discovered the satcuts $[3.5 - \epsilon, 3.5]$ and $[5.8 - \epsilon, 5.8]$. The first is the leftmost satcut of the interval of satcuts. A_1 discovered an additional satcut $[6, 6 - \epsilon]$. Joining this satcut with A_2's second satcut returns a result of $[6, 5.8]$, which is the rightmost satcut of the interval of satcuts.



(a) Runtime vs root rate on 4 synthetic signals.



(b) Online monitoring. The red horizontal plane indicates the runtime threshold (namely, 5s) below which it is possible to do online detection.

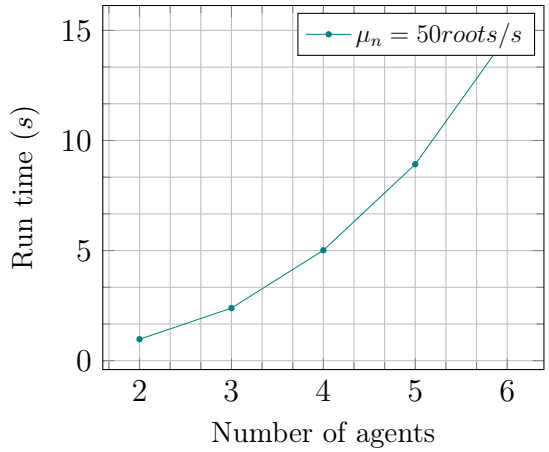
Figure 6.5 Runtime vs root rate and N on synthetic data.

6.5 Case Studies and Evaluation

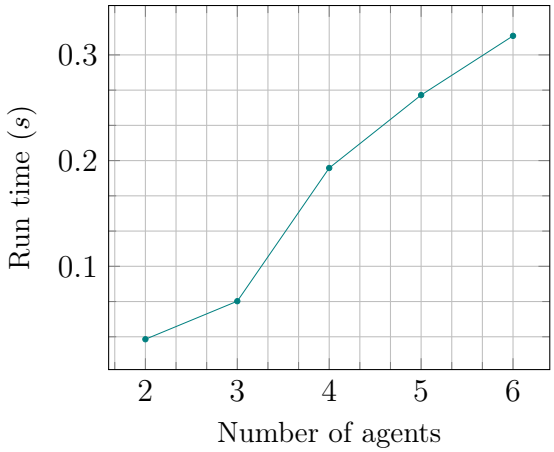
We implemented our detection algorithm and ran experiments to 1) illustrate its operation, and 2) observe runtime scaling with number of agents and with average rate of events. The detector was implemented in Julia for ease of prototyping, but future versions will be in C for speed. All experiments are replicated to exhibit 95% confidence interval. Experiments were ran on a single thread of an Ubuntu machine powered by an AMD Ryzen 7 5800X CPU @ 3.80GHz.

We consider two sources of data: the first is a set of N synthetically generated signals, $N = 1...6$. Each signal has a $5s$ duration, and is generated randomly while ensuring an average root rate of μ_n. That is, on average, μ_n roots exist in every second of signal x_n. For the second source of data, we use the Fly-by-Logic toolbox [100] to control up to 6 simulated UAVs (i.e., drones) performing various reach-avoid missions. Their 3-dimensional trajectories are recorded over 6 seconds. We monitor the predicate "All UAVs are at a height of at least $10m$ simultaneously". Maximum clock skew ϵ is set to 0.05s.

Effect of root rate (μ_n) on run time. We use 4 synthetic signals of $5s$ duration, and measure the detection runtime as the root rate for all signals is varied between $10 roots/s$



(a) Detection of synthetic signals at 50 roots/s.



(b) Detection of UAV signals.

Figure 6.6 Runtime vs number of agents.

and $50 roots/s$. Figure 6.5a shows the results. Naturally, as μ_ne increases, so does the run time due to having to process more tokens.

Online detection. We want to identify when it is possible for us to perform online detection with the Julia implementation, i.e. such that the detector finishes before the end of the signal being processed. To this end, we use the synthetic signals of duration 5s and vary both root rate and number of agents. Figure 6.5b shows the results: all combinations of root rates and number of agents with runtimes under the threshold of 5s can be performed online.

Effect of number of agents on run time. Figure 6.6 shows the effect of number of agents N on runtime. As expected, the runtime increases with N.

CHAPTER 7

RESOURCE OPTIMIZATION OF STREAM PROCESSING IN LAYERED SENSOR NETWORKS

In this chapter, we set our sights on monitoring reliability by optimizing resource consumption in a generalized class of CPS. In Chapters 3, 4, 5, and 6 we proposed different monitoring techniques for distributed systems with respect to different specifications under both centralized and decentralized monitoring settings. However, solely monitoring formal specification on a distributed CPS is not enough to guarantee its functionality. For example, in a decentralized monitoring setup, if one or more monitors start reporting erroneous results, then it is possible to reach false positive and/or false negative verdicts on the distributed CPS against some specification. Therefore it is imperative that we ensure the reliability of all monitors, and by extension, the reliability of the distributed CPS, that is, the network of monitors or agents, as a whole.

However, determining reliability of a distributed CPS depends on an array of factors, including the type of agents in the network. For example, the method for computing reliability of a network of UAVs will vastly vary from the method for computing reliability of a network of medical equipment.

To this end, we present a generalized model of a class of CPS, where each monitor is represented by an (IoT) device or an agent in a layered network of producers and consumers. We elaborate our technique for monitoring reliability of layered stream processing networks, while optimizing for minimal resource consumption by its nodes.

7.1 Producer-Consumer Network with Resource Constraints

Before talking about our problem statement, we present our model that is used to capture a layered network of nodes tasked with stream processing jobs subject to resource constraints, flows, and target reliability.

7.1.1 Resource Bounds

We first present the notions of *reusable* and *consumable* resources in our model:

- *Reusable resources* are not depleted when an item is processed. Examples of reusable resources are CPU, power, memory, network bandwidth, and quality. These resources are instantly reclaimed once an item is processed. We denote the finite set of reusable resources in the system as follows:

$$\mathbb{R} = \{R_1, R_2, \ldots, R_n\},$$

for some $n \geq 1$.

- *Consumable resources* are depleted once an item is processed. Examples of consumable resources are energy, time, and reliability. For instance, once error is encountered during the processing of an item along its path in the network, it cannot be reclaimed. We denote the finite set of consumable (depletable) resources as follows:

$$\mathbb{D} = \{D_1, D_2, \ldots, D_m\}$$

for some $m \geq 1$.

Our model supports bounding resources on both nodes and edges. Let $G = (\mathbb{V}, \mathbb{E})$ be a producer-consumer network. A bound on a resource $res \in \mathbb{R} \cup \mathbb{D}$ for a subset of nodes $V \subseteq \mathbb{V}$ (respectively, a subset of edges $E \subseteq \mathbb{E}$) is denoted by b_V^{res} (respectively, b_E^{res}). We also set $b_V^{res} = \langle lb, ub \rangle$ (respectively, $b_E^{res} = \langle lb, ub \rangle$) as a pair that implies the sum of resource $res \in \mathbb{R} \cup \mathbb{D}$ unit (e.g., power) consumed by all nodes (respectively, edges) in V (respectively, E) must reside within the lower bound lb and the upper bound ub. Finally, we denote the set of all resource bounds for all resources in $\mathbb{R} \cup \mathbb{D}$ and for any subsets of nodes and edged by $\mathbb{B}$.

For instance, if a node v has 8 cores, using the conventional notation of multi-core systems, the maximum CPU usage is 800%. In this case, a bound $b_{\{v\}}^{\text{CPU}} = \langle 0, 800 \rangle$ is applied. Another example is applying power bounds to a cluster of nodes. This implies that the sum of power consumed by all nodes in the cluster should not exceed a specific value. A bound $b_{\{v_1,v_2,v_3\}}^{\text{PWR}} = \langle 0, 500 \rangle$ denotes the total power consumption of nodes v_1, v_2 and v_3 should not exceed 500 watts.

Let μ_v^{res} (respectively, μ_e^{res}) be the amount of resource $res \in \mathbb{R} \cup \mathbb{D}$ unit consumed by node $v \in \mathbb{V}$ (respectively, edge $e \in \mathbb{E}$). Formally, a bound $b_V^{res} = \langle lb, ub \rangle$ on vertices $V \subseteq \mathbb{V}$ and a resource res enforces the following:

$$lb \leq \sum_{v \in V} \mu_v^{res} \leq ub.$$

Likewise, a bound $b_E^{res} = \langle lb, ub \rangle$ on edges $E \subseteq \mathbb{E}$ and a resource res enforces the following:

$$lb \leq \sum_{e \in E} \mu_e^{res} \leq ub.$$

7.1.2 Configurations

There are various configuration parameters that impact the resource usage of a node and the reliability of its output.

- *Sampling rate.* Some systems depend on sampling from continuous-time and continuous-valued signals and the amount of resources consumed by a node is proportional to the sampling rate [18]. Lower sampling rate is usually associated with reduced reliability or confidence. Hence, sampling rate is a configuration parameter that controls the tradeoff between resource usage and reliability.
- *Outgoing data rate.* The outgoing data rate of a node impacts the resource usage of subsequent nodes [18, 113]. If subsequent nodes decide to sample this data, then reliability is negatively impacted.
- *Precision.* Some algorithms support controllable precision. For instance, image processing may be accomplished with high or low precision [86]. The work in [84] demonstrates how configurable precision impacts accuracy and resource usage.
- *Algorithm alternatives.* In some systems, there are different algorithms that can be used to process the data, with varying degrees of resource usage and reliability [70, 76]. For instance, data loss prevention (DLP) systems employ different classifiers for malicious activity that are designed to have different processing costs [93].

To simplify our model, we abstract all the above parameters into a single *quality* symbol. This symbol encompasses sampling, buffering, precision, and algorithmic alternatives. Given a producer-consumer network $G = (\mathbb{V}, \mathbb{E})$, let us associate each node $v \in \mathbb{V}$ with a finite set of *quality levels*:

$$\mathbb{Q}_v = \Big\{ \mathsf{Qual}^1(v), \mathsf{Qual}^2(v), \ldots, \mathsf{Qual}^k(v) \Big\}$$

where the number of levels k can be different for each node v. A node v can use each quality level $\mathsf{Qual}^i(v)$, where $1 \leq i \leq k$ to process items that are being received at some input data rate in $\mathsf{IRate}(v)$ and being produced at some outgoing data rate in $\mathsf{ORate}(v)$. Part of our stream optimization (see Section 7.2) is to find the best quality for the possible input/output data rates. To this end, for each node $v \in \mathbb{V}$, let

$$\vartheta_v : \mathsf{IRate}(v) \times \mathbb{Q}_v \rightarrow \mathsf{ORate}(v)$$

be a function that maps an incoming data rate and a quality level to an outgoing data rate. That is, we have:

$$\mathsf{ORate}_v = \vartheta\Big(\mathsf{IRate}(v), \mathsf{Qual}^i(v) \Big)$$

where $\mathsf{Qual}^i(v)$ is the i^{th} quality level of node v.

7.1.3 Reliability

Quantifying reliability is generally a challenging task. Reliability of each node not only depends on its quality level, but on other *environmental factors* as well. For example, the reliability of a node that captures video streams may vary based on the time of the day and the surrounding lighting conditions. Another example would be the case where a node may become less reliable once it nears the end of its average life cycle. Let us assume each node $v \in \mathbb{V}$ is influenced by m_v number of environmental factors. We denote $\mathsf{U}_v^j \in [0, 1]$ where $1 \leq j \leq m_v$ as the j^{th} environmental factor of the node v. An environmental factor of 1 indicates the best possible reliability when other factors (as well as the quality) remain unchanged, whereas an environmental factor of 0 indicates the worst. All the intermediary values are determined by the node's architecture. In a similar manner, we denote $\mathsf{U}_v^{\mathsf{Qual}} \in [0, 1]$ as the

quality factor of the node v. A quality factor of 1 maps to the highest quality level $\mathsf{Qual}^{\max}(v)$ supported by the implementation of the node's code. This could be a configuration where a computationally intensive algorithm is used, input data is not sampled or buffered, and numerical precision is set to the maximum supported precision. On the other hand, a quality factor of 0 maps to the lowest quality level $\mathsf{Qual}^{\min}(v)$ supported by the node. This should be a configuration below which the system becomes unusable. Quality factor for the remaining quality levels in $\mathbb{Q}_v - \{\mathsf{Qual}^{\max}(v), \mathsf{Qual}^{\min}(v)\}$ are determined by the system design. Now that we have defined the quality factor $\mathsf{U}_v^{\mathsf{Qual}}$ and the environmental factors $\mathsf{U}_v^1, \mathsf{U}_v^2, \ldots, \mathsf{U}_v^{m_v}$ for a node v, we are ready to define its reliability $\alpha_v \in [0, 1]$ as follows:

$$\alpha_v = \frac{\mathsf{U}_v^{\mathsf{Qual}} + \mathsf{W}_v^1.\mathsf{U}_v^1 + \mathsf{W}_v^2.\mathsf{U}_v^2 + \ldots + \mathsf{W}_v^{m_v}.\mathsf{U}_v^{m_v}}{1 + \mathsf{W}_v^1 + \mathsf{W}_v^2 + \ldots + \mathsf{W}_v^{m_v}}$$

Where $\mathbb{W}_v = \{\mathsf{W}_v^1, \mathsf{W}_v^2, \ldots, \mathsf{W}_v^{m_v}\}$ are the respective weights of the environmental factors $\mathbb{U}_v = \{\mathsf{U}_v^1, \mathsf{U}_v^2, \ldots, \mathsf{U}_v^{m_v}\}$.

Note that it is difficult to determine the discrete quality levels of a node, and map the said quality levels to numerical quality factor values. This is mainly because different nodes in a producer-consumer network carry out different tasks, and therefore, require their own method of quality level determination. For example, the quality level of a node that is tasked with capturing and streaming video can be determined by its current video resolution. In other words, the maximum operational resolution can be considered as the highest quality and mapped to a quality factor of 1, the minimum operational resolution can be considered as the lowest quality and mapped to a quality factor of 0, and every other operational resolutions in between can be mapped between 0 and 1 based on their pixel count. However, this method will clearly fail determine the quality levels of a node that is tasked with detecting motion, where the polling interval rate could be a better representation of quality levels for the said node. Note that, we do not attempt to provide an absolute method for determining quality levels and mapping them to appropriate quality factor values. We merely propose an abstraction that allows tweaking the system into yielding desirable results.

7.1.4 Relationship between Configurations and Resources

We now define the relationships between configurations and resources. Let CRate_{res} denote the set of possible rates of consumption of resource res. Also, let

$$\varphi_v^{res} : \mathsf{IRate}(v) \times \mathbb{Q}_v \rightarrow \mathsf{CRate}(res)$$

be a function that maps the rate of incoming data and the quality level of node $v \in \mathbb{V}$ to a possible consumption rate value in $\mathsf{CRate}(res)$. For example, for a node v with a quality levels of $\mathsf{Qual}(v)$ that is receiving data at the rate of $\mathsf{IRate}(v)$, we determine the rate at which resource PWR is consumed on the said node using $\varphi_v^{\mathrm{PWR}}(\mathsf{IRate}(v), \mathsf{Qual}(v))$. Recall that resource res can be either reusable or non-reusable. Hence, each node defines a set of functions, in which the elements are functions φ_v^{res} for all resources $res \in \mathbb{R} \cup \mathbb{D}$ as follows:

$$\Phi_v = \left\{ \varphi_v^{res} \mid res \in \mathbb{R} \cup \mathbb{D} - \{\mathrm{REL}\} \right\}$$

where REL is the reliability resource. While reliability depends on the quality level of a node like other resources, it also depends on the reliability of incoming data, as well as the environmental factors. Therefore, we exclude reliability since it is defined differently.

We incorporate this notion of reliability to model systems where error is compound, i.e., receiving erroneous data may impact the reliability of produced data differently even at the same quality level, and under same environmental factors. This behavior is common in precision based quality levels, where rounding error is compounded as more mathematical operations are performed on a data path. Thus, we introduce the following recursive function ψ_v to determine the *compounded reliability* of node $v \in \mathbb{V}$ as follows:

$$\psi_v(\mathsf{Qual}(v)) = \begin{cases} \mathsf{comp}(\mathsf{Qual}(v), \mathbb{U}_v, \mathbb{W}_v, \\ \quad \{\psi_u(\mathsf{Qual}(u)) \mid u \in \mathsf{Pred}(v)\}) \ \text{ if } \ \mathsf{Pred}(v) \neq \emptyset \\ \alpha_v \ \text{ if } \ \mathsf{Pred}(v) = \emptyset \end{cases}$$

	ORate			PWR			TIME		
	q^1	q^2	q^3	q^1	q^2	q^3	q^1	q^2	q^3
v_1	100	75	50	80	65	40	10	13.3	20
v_2	90	60	30	75	55	35	11.1	16.6	33.3
v_3	100	70	40	80	65	40	10	14.3	25
v_4	120	90	60	85	70	40	8.3	11.1	16.6
v_5	110	80	70	85	75	36	10.3	15.1	21.1

Table 7.1 Nodes $v_{[1,5]}$ resource usage.

	PWR			TIME			α_v		
	q^1	q^2	q^3	q^1	q^2	q^3	q^1	q^2	q^3
v_6	85	70	55	16	20.1	25	100	90	82
v_7	80	65	50	18.2	18.2	38.3	100	92	84
v_8	88	55	50	15	17.7	29	100	88	79
v_9	90	70	55	13	14.5	19	100	93	83
v_{10}	100	80	60	17	22	45	–	–	–

Table 7.2 Nodes $v_{[6,10]}$ resource usage.

where α_v is the reliability of v when it is a source node (by system design) and comp denotes a function that computes the reliability of a node given its quality, environmental factors and their weights, and the reliability of its predecessors. For instance, comp could be instantiated with a function that computes the average (or maximum) reliability of all predecessors times the quality level of the node.

For example, we introduce the characteristics of the nodes in the network of Figure 7.2. Table 7.1 lists the production rate (ORate) for resources power (PWR) and response time (TIME) of nodes $v_{[1,4]}$ at different quality levels. We abbreviate the quality level Qual^i as q_i The quality level for nodes $v_{[1,4]}$ is designated by the sampling rate. Thus, the highest quality level q^1 has the highest rate of outgoing items, versus the lowest quality level q^3.

7.1.5 Revised Definitions

Based on the definitions introduced in the previous subsections, we now redefine a node as follows:

$$v = \Big\langle \mathsf{Pred}(v), \mathsf{Succ}(v), \mathsf{Qual}(v), \mathsf{ORate}(v), \Phi_v, \psi_v \Big\rangle$$

Thus, the node now includes a set of quality levels (i.e., $\mathsf{Qual}(v)$), a function that determines the rate of outgoing data (i.e., $\mathsf{ORate}(v)$), a set of functions that determine resource usage (i.e., Φ_v), and a function that determines the reliability of the node's output (i.e., ψ_v).

Finally, we redefine the graph as follows:

$$G = \left\langle \mathbb{V}, \mathbb{E}, \mathbb{R}, \mathbb{D}, \mathbb{B} \right\rangle$$

Thus, the graph now defines a set $\mathbb{D}$ of consumable resources, a set $\mathbb{R}$ of reusable resources, and a set $\mathbb{B}$ of bounds on resources.

7.2 Problem Statement

First, observe that in the model proposed in Section 7.1, quality levels may affect the following:

1. Node reliability ψ_v, which is a function of the quality level, environmental factors and their weights, and the incoming reliability values of all predecessors.

2. Resource consumption φ_v^{res}, which is a function of the quality level and the incoming data rate.

3. Production rate $\mathsf{ORate}(v)$, which is a function of the quality level and the incoming data rate.

The majority of the stream processing systems benefit greatly from knowing how to answer one or both of these two questions; (1) how the usage of available resources can be optimized to reach maximum reliability, and (2) how to minimize available resource usage while ensuring reliability is maintained above a target threshold. Thus, roughly speaking, our *multi-objective* problem statement is as follows. Given (1) a producer consumer network on which a set of bounds is defined, and (2) a *target reliability* for all consumer-only nodes,

- *Quality Maximization.* Our first objective is to identify a single quality level for every node, such that the reliability for consumer-only nodes, is maximized, while satisfying all bounds. For example, maximizing the efficiency of each device in a sensor network

of smart home devices while not exceeding the specified renewable resources like power, CPU, bandwidth etc.

- *Resource Usage Minimization.* Our second objective is to minimize consumption of a given resource for all nodes while achieving a target reliability. For example, minimizing the power usage of a producer-consumer network, that does not demand maximum reliability from its nodes.

Formally, our optimization problem is as follows:

Problem Statement

Given a producer-consumer network $G = \langle \mathbb{V}, \mathbb{E}, \mathbb{R}, \mathbb{D}, \mathbb{B} \rangle$, a resource $res \in \mathbb{R} \cup \mathbb{D}$, identify quality levels $\mathsf{Qual}(u)$ for all $u \in \mathbb{V}$ subject to:

$$\forall v \in \{v' \mid \mathsf{Succ}(v') = \emptyset\}. \max \Big(\psi_v \big(\mathsf{Qual}(v) \big) \Big)$$

and

$$\forall v \in \mathbb{V}. \min \Big(\varphi_v^{res} \big(\mathsf{IRate}(v), \mathsf{Qual}(v) \big) \Big)$$

In the next section, we will present our solution to solve the above optimization problem.

7.3 SMT-based Solution

In this section, we present our solution to solve the multi-objective optimization problem presented in Section 7.2. Our solution is based on a reduction to solving the satisfiability problem for SMT. Practically, we will utilize an SMT-solver in order to optimize reliability and resource consumption tradeoffs. The SMT problem is solved on a remote *network monitor* that is used to poll each sensor node in the network at a fixed interval in order to keep track of available resources, as well as control the quality levels of the said node.

Each SMT instance is described in terms of (1) SMT entities (e.g., variables, functions, constants, etc.) and (2) SMT constraints (e.g., Boolean conditions over first-order predicates).

7.3.1 SMT Entities

We now introduce the entities that are used to represent the components of our producer-consumer network $G = (\mathbb{V}, \mathbb{E})$. In some SMT entity definitions, we use *free variables* that the SMT solver can manipulate in order to provide a satisfaction verdict.

Nodes. In our SMT encoding, we represent the set of nodes $\mathbb{V}$ as a set of integers $\{1, 2, \cdots, |\mathbb{V}|\}$, where each element represents a node in $\mathbb{V}$.

Edges. We store the information of edges in $\mathbb{E}$ in the form of a $|\mathbb{V}| \times |\mathbb{V}|$ Boolean array edge such that:

$$\bigwedge_{i=1}^{|\mathbb{V}|} \bigwedge_{j=1}^{|\mathbb{V}|} \mathsf{edge}[i][j] = \begin{cases} \texttt{true} & \text{if} \quad (i,j) \in \mathbb{E} \\ \texttt{false} & \text{if} \quad (i,j) \notin \mathbb{E} \end{cases}$$

where $\mathsf{edge}[i][j]$ implies there exists an edge from node v_i to node v_j in G.

Successor Nodes. We encode the function Succ as an SMT function succ that maps a node to a set of successor nodes as follows:

$$\bigwedge_{i=1}^{|\mathbb{V}|} \mathsf{succ}(i) = \Big\{ j \mid (i,j) \in \mathbb{E} \Big\}$$

Predecessor Nodes. We encode the function Pred as an SMT function pred that maps a node to a set of predecessor nodes as follows:

$$\bigwedge_{i=1}^{|\mathbb{V}|} \mathsf{pred}(i) = \{ j \mid (j,i) \in \mathbb{E} \}$$

Node Resource Consumption. We define the function rcon that maps a resource and a node to a free variable that denotes the resource consumption of the said node as follows:

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \bigwedge_{i=1}^{|\mathbb{V}|} \mathsf{rcon}(res, i) = \mu_i^{res}$$

Edge Resource Consumption. We define the function rcoe that maps a resource and an edge to a free variable that denotes the resource consumption of the said edge as follows:

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \bigwedge_{(i,j) \in \mathbb{E}} \mathsf{rcoe}(res, (i,j)) = \mu_{(i,j)}^{res}$$

Resource Inflow. We define the function iflo that maps a resource and a node to a free variable that denotes the inflow of resource to the said node as follows:

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \bigwedge_{i=1}^{|\mathbb{V}|} \mathsf{iflo}(res, i) = \zeta_i^{res},$$

where ζ_v^{res} denotes the inflow of resource res into node v

Resource Outflow. We define the function oflo that maps a resource and a node to a free variable that denotes the outflow of resource from the said node as follows:

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \bigwedge_{i=1}^{|\mathbb{V}|} \mathsf{oflo}(res, i) = \xi_i^{res},$$

where ξ_v^{res} denotes the outflow of resource res from node v.

Edge Flow. We define the function eflo that maps a resource and an edge to a free variable that denotes the amount of flow going through the said edge as follows:

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \bigwedge_{(i,j) \in \mathbb{E}} \mathsf{eflo}(res, (i, j)) = \nu_{(i,j)}^{res}$$

where $\nu_{(i,j)}^{res}$ denotes the flow of resource res through edge (i, j).

Quality. We encode the function Qual as an SMT function qual that maps a node to all of its abstracted quality levels in disjunction (see Section 7.1) as follows:

$$\bigwedge_{i=1}^{|\mathbb{V}|} \left(\mathsf{qual}(i) = \bigvee_{j=1}^{|\mathbb{Q}|} \mathsf{Qual}^j(v_i)\right)$$

Reliability. We encode the function ψ as an SMT function rel that maps the quality of a node to its reliability as follows:

$$\bigwedge_{i=1}^{|\mathbb{V}|} \mathsf{rel}(\mathsf{qual}(i)) = \begin{cases} \mathsf{eval}(\mathsf{qual}(i), \mathbb{U}_v, \mathbb{W}_v, & \\ \qquad \{\mathsf{rel}(\mathsf{qual}(j)) \mid j \in \mathsf{pred}(i)\}) & \text{if} \quad \mathsf{pred}(i) \neq \emptyset \\ \alpha_i & \text{if} \quad \mathsf{pred}(i) = \emptyset \end{cases}$$

Note that when $\mathsf{pred}(i) = \emptyset$, node i is the source (producer-only) node in the producer-consumer network, and therefore, its reliability, α_i is known.

7.3.2 SMT Constraints

We now introduce the constraints that address our problem statement using the SMT entities we defined in the previous section.

Resource Inflow Constraint. For resources $res \in \mathbb{R} \cup \mathbb{D}$, the amount of a resource flowing into a node depends on the amount of flow carried over all its incoming edges. Traditionally, the inflow is the sum of all flows on incoming edges. That is,

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \bigwedge_{i=1}^{|\mathbb{V}|} \zeta_i^{res} = \sum \left\{ \mathsf{eflo}(res, (i,j)) \mid (i,j) \in \{(v', v) \mid v' \in \mathsf{pred}(v)\} \right\}$$

For instance, power is a resource that can be summed over incoming edges.

Resource Outflow Constraint. For resources in $res \in \mathbb{R} \cup \mathbb{D}$, the amount of a resource flowing out from a node is traditionally equal to the amount of the resource flowing in, which is the conservation of flow principle. This models renewable resources efficiently, yet does not capture non-renewable resources. We generalize the resource outflow constraint using function Ξ:

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \bigwedge_{i=1}^{|\mathbb{V}|} \mathsf{oflo}(res, i) = \Xi_i^r \left(\mathsf{iflo}(res, i), \mathsf{rcon}(r, i) \right)$$

For instance, power is a renewable resource, and thus,

$$\bigwedge_{i=1}^{|\mathbb{V}|} \Xi_i^{\mathrm{PWR}} \left(\mathsf{iflo}(\mathrm{PWR}, i), \mathsf{rcon}(\mathrm{PWR}, i) \right) = \mathsf{iflo}(\mathrm{PWR}, i)$$

However, energy is depletable, and therefore,

$$\bigwedge_{i=1}^{|\mathbb{V}|} \Xi_i^{\mathrm{EGY}} \left(\mathsf{iflo}(\mathrm{EGY}, i), \mathsf{rcon}(\mathrm{EGY}, i) \right) = \mathsf{iflo}(\mathrm{EGY}, i) - \mathsf{rcon}(\mathrm{EGY}, i)$$

Resource Bound Constraint on Nodes. For all resources $res \in \mathbb{R} \cup \mathbb{D}$, we enforce the given upper bound and lower bound on nodes as follows:

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \left(lb \leq \sum_{i=1}^{|\mathbb{V}|} \left\{ \mathsf{rcon}(res, i) \right\} \leq ub \right)$$

Resource Bound Constraint on Edges. We use bounds on edges to control the distribution of resources $res \in \mathbb{R} \cup \mathbb{D}$ across outgoing edges of a node. We identify two main methods of assigning flow to outgoing edges: *broadcast* and *distribution* resources.

Broadcast. In this case, nodes broadcast their outflow to outgoing edges. Reliability is broadcast, since all outgoing edges of a node carry data with the same reliability level that the node produces. We can enforce resources $res \in \mathbb{R} \cup \mathbb{D}$ to be broadcast using the following constraint:

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \bigwedge_{(i,j) \in \mathbb{E}} \Big(\mathsf{oflo}(res, i) \leq \mathsf{rcoe}(res, (i,j)) \leq \mathsf{oflo}(res, i) \Big)$$

Upon simplification, we have:

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \bigwedge_{(i,j) \in \mathbb{E}} \mathsf{rcoe}(res, (i,j)) = \mathsf{oflo}(res, i)$$

Distribution. In this case, the outflow is distributed across all outgoing edges. For instance, in a multiple consumer setting any one of a set of receiving nodes can process items. In this case, the outgoing data flow of the producer is distributed among all consumer nodes. The objective here is to determine the fraction of data flowing to each consumer such that resource bounds are respected and the usage of a specific resource is optimized. We can enforce resources $res \in \mathbb{R} \cup \mathbb{D}$ to be distributed using the following constraint:

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \bigwedge_{i=1}^{|\mathbb{V}|} \big(\mathsf{oflo}(res, i) \leq \sum_{j \in \mathsf{succ}(i)} \big\{ \mathsf{rcoe}(res, (i,j)) \big\} \leq \mathsf{oflo}(res, i) \big)$$

Upon simplification, we have:

$$\bigwedge_{res \in \mathbb{R} \cup \mathbb{D}} \bigwedge_{i=1}^{|\mathbb{V}|} \sum_{j \in \mathsf{succ}(i)} \big\{ \mathsf{rcoe}(res, (i,j)) \big\} = \mathsf{oflo}(res, i)$$

Data Flow Constraint. Data outflow can be either broadcast or distributed. We encode the function Out as an SMT function out that maps an edge to outdoing data rate of that edge.

Broadcast. In case the data outflow is broadcast, we enforce the following constraint on all edges:

$$\bigwedge_{(i,j) \in \mathbb{E}} \mathsf{out}((i,j)) = \mathsf{ORate}(v_i)$$

Distribution. In case the data outflow is distributed, we enforce the following constraint on all edges:

$$\bigwedge_{i=1}^{|\mathbb{V}|} \sum_{j \in \mathsf{succ}(i)} \{\mathsf{out}((i,j))\} = \mathsf{ORate}(v_i)$$

Reliability Maximization Constraint. Finally, let C denote the conjunction of all the above constraints. The constraint for maximizing reliability on sink (consumer only) nodes is as follows:

$$C \wedge \big(\bigwedge_{i \in \{j | \mathsf{succ}(i) = \emptyset\}} \max(\mathsf{rel}(\mathsf{qual}(i))) \big)$$

Resource Optimization Constraint. If we want to minimize the total consumption of some $res \in \mathbb{R} \cup \mathbb{D}$ across all nodes, while ensuring the reliability on sink (consumer only) nodes remain above a given threshold α, then we enforce the following constraint instead:

$$C \wedge \Big(\bigwedge_{i \in \{j | \mathsf{succ}(i) = \emptyset\}} \mathsf{rel}(\mathsf{qual}(i)) \geq \alpha \Big) \wedge \min \big(\sum_{i=1}^{|\mathbb{V}|} \{\mathsf{rcon}(res, i)\} \big)$$

7.3.3 Solver Optimization

Solving the Reliability Maximizing Constraint and the Resource Optimization Constraint both require a significant amount of computation power and time (see Figure 7.1a). This is mostly because C is a conjunction of a large set of constraints, coupled with the fact that it is a minimization or maximization problem. This means, there is only one solution for which the value of the object is maximized or minimized, which in turn means our SMT solver having to explore a large search space.

To this end, we employ some optimization techniques to our model in order to reduce run time for the SMT solver. In this subsection, we show one such technique and report the improvement it shows in terms of run time over the naive method.

Binary Probing: First, let A_α be an SMT constraint such that,

$$A_\alpha = C \wedge \big(\bigwedge_{i \in \{j | \mathsf{succ}(i) = \emptyset\}} \mathsf{rel}(\mathsf{qual}(i)) \geq \alpha \big)$$

Data: Producer-consumer network G, SMT constraint A_α, Target reliability α, Error margin ϵ
Result: Estimated Best Reliability α'

```
α_min ← 0  α_max ← 1  pivot ← .5  α' ← 0
found ← false  while ¬found do
    if |α_max − α_min| ≤ ε then
        found ← true
    end
    β ← solve(A_pivot, G)  if β ≠ −1 then
        if β > α' then
            α' ← β
        end
        α_min ← ⌊pivot⌋
    end
    else
        α_max ← ⌊pivot⌋
    end
    if found then
        break
    end
    pivot ← (α_min + α_max)/2
end
return α'
```

Algorithm 7.1 Best Reliability Estimation Algorithm

We say $A_\alpha \models G$ iff A_α is satisfied for α, otherwise $A_\alpha \not\models G$. Let solve be a function that, given G and A_α, returns some value β such that $\alpha \leq \beta \leq 1$ and $A_\alpha \models G$, otherwise $\beta = -1$. Formally,

$$\mathsf{solve}(A_\alpha, G) \begin{cases} \beta & \text{if} \quad A_\alpha \models G \wedge \beta \in [\alpha, 1] \\ -1 & \text{otherwise} \end{cases}$$

Now, using a *Binary Probing* technique described in Algorithm 7.1, we can invoke our SMT solver in a pattern similar to the traditional binary search, and find an estimated α', that is sufficiently close, that is, within the error margin ϵ of the *real* best reliability. Using a similar technique to this, we can also find the estimated minimum of any $res \in \mathbb{R} \cup \mathbb{D}$. It should be mentioned that even in worst case, after just five SMT invocations, the error from binary probing will only be $\approx 3.125\%$, which usually falls within the acceptable error margins for devices in sensor networks, as far as reliability and other resources are concerned.

7.4 Machine Learning-based Optimization

On top of our SMT-based solution described in Section 3.3, we employ a machine learning-based optimization technique to further improve our solution in terms of run time at the cost of negligible (details in the next section) accuracy.

7.4.1 Artificial Neural Network

We first create an *Artificial Neural Network* [2] (ANN) where neurons in the output layer denote the resources that need to be optimized, and the neurons in the input layer denote the remaining resources. For example, when solving the Quality Maximization problem, each neuron in the output layer represents each quality level of each node $v \in \mathbb{V}$, that is, the number of neurons in the output layer is $l_o = |\mathbb{V}|$. each neuron in the input layer represents remaining resources like power, CPU, memory, bandwidth etc., that is $l_i = |\mathbb{R} \cup \mathbb{D} - \{\text{QUAL}\}|$, where QUAL $\in \mathbb{R}$ is the quality resource. For determining the number of neurons in the hidden layer, we chose the method proposed by the authors in [129], that is, the number of neurons in the hidden layer is,

$$l_h = \left(\frac{2}{3} \times |\mathbb{R} \cup \mathbb{D} - \{\text{QUAL}\}|\right) + |\mathbb{V}|$$

As an example, let us consider the producer-consumer network in Figure 2.5. If we want to maximize the quality of the network with respect to power (PWR), CPU (CPU), memory (MEM) and bandwidth (BW), then the corresponding ANN should be of the form where $l_o = 9$, $l_i = 4$, and $l_h = 12$.

7.4.2 Training Dataset

Our training dataset for the ANN is generated using the SMT-based solution detailed in Section 3.3. For example, in order to generate the training data-set for the Quality Maximization problem, we find the best qualities for all nodes for resources with random values, and populate the data-set with the results. This allows us to carry out machine learning process in an *unsupervised* manner.

During training, while the traditional approach is to split the dataset into two subsets

(i.e. training dataset and testing dataset), for smaller datasets, this may introduce biased estimates [56]. As our model must be applicable to both small and large datasets, in order to reduce statistical bias, we employ *k-fold cross validation* [123] to train our ANN. The process of k-fold cross validation is as follows:

1. Split the dataset into randomized groups of equal sizes: $g_1, g_2, \ldots, g_k$.

2. For $i \in [1, k]$ do:

   - Assign group g_i as the test dataset.
   - Assign groups $g_1, g_2, \ldots, g_{i-1}, g_{i+1}, \ldots, g_k$ as the training dataset.
   - Train the model on the training set and evaluate it on the test dataset.

Using k-fold cross validation ensures that each group is used as the testing dataset once, and as the training dataset $k-1$ times. There various ways of selecting the value of k. However, in our work, we assign $k = 10$, as this is shown to generally yield minimal statistical bias [51, 69]. Note that for generating our dataset, we normalize the sample values to avoid unwanted weights. However, when we report the experimental results, we use the actual values for ease of camparison and understandability.

7.4.3 Model Accuracy

We determine the accuracy of our trained model by directly comparing its results with the results from SMT-based solution. For the Quality Maximization problem, we define the accuracy, acc_q of the model as follows:

$$acc_q = 1 - \frac{\sum_{v \in \mathbb{V}} |SMT_{q_v} - ML_{q_v}|}{\sum_{v \in \mathbb{V}} |\mathbb{Q}_v|}$$

Where the quality level reported by the SMT-based solution is the $SMT^{th}_{q_v}$ quality level of node v, the quality level reported by the machine learning model is the $ML^{th}_{q_v}$ quality level of node v, and $\mathbb{Q}_v$ is the set of quality levels of node v. For the Resource Minimization problem, we define the accuracy acc_r of the model as follows:

$$acc_r = \frac{\sum_{res \in \mathbb{R} \cup \mathbb{D}} \frac{|SMT_{res_v} - ML_{res_v}|}{MAX_{res_v} - MIN_{res_v}}}{|\mathbb{R} \cup \mathbb{D}|}$$

Where SMT_{res_v} is the value of resource res of node v reported by the SMT-based solution, ML_{res_v} is the value of resource res of node v reported by the machine learning model, and MAX_{res_v} (resp., MIN_{res_v}) denotes the upper bound (resp., lower bound) of resource res observed in the dataset. Note that, $0 \leq \{acc_q, acc_m\} \leq 1$, where 0 indicates the worst accuracy, and 1 indicates the best accuracy.

7.5 Case Studies and Evaluation

In this section, we evaluate our technique for resource optimization using synthetic data generated from a *simulated* layered network of nodes, as well as real world data collected from a network of embedded devices, where the nodes in the network are Raspberry Pi devices tasked with specific streaming objectives.

7.5.1 Synthetic Experiments

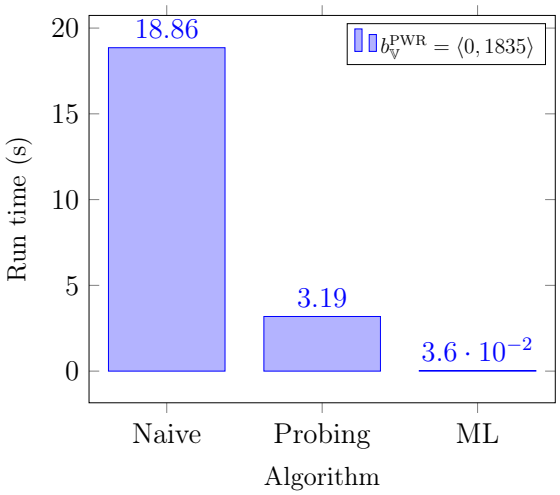
In this subsection, we introduce our synthetic experiments to demonstrate how our proposed model can be used to optimize various resources.

7.5.1.1 Experimental Setup

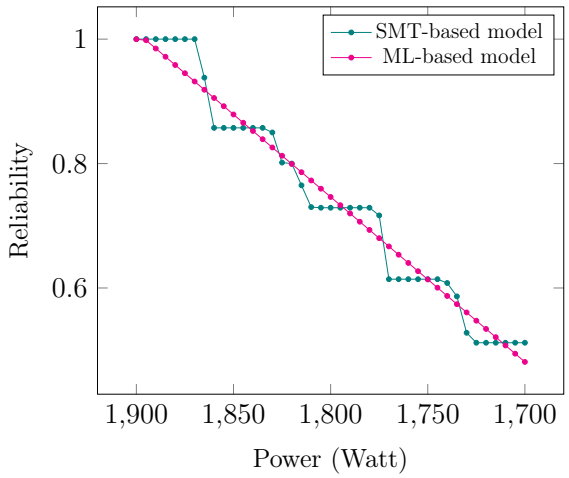
We construct our producer-consumer network using 8 nodes, $\mathbb{V} = \{v_1, v_2, \cdots, v_8\}$, with $v_{[1,7]}$ being the producer nodes, and $v_{[2,8]}$ being the consumer nodes. We add two placeholder nodes to the network, v_{in} and v_{out}, along with two edges (v_{in}, v_1) and (v_8, v_{out}). Figure 7.2 shows our producer-consumer network. We use edge (v_{in}, v_1) to regulate bounds on resources, and we use node v_{out} to compute network reliability.

7.5.1.2 Resource Bounds

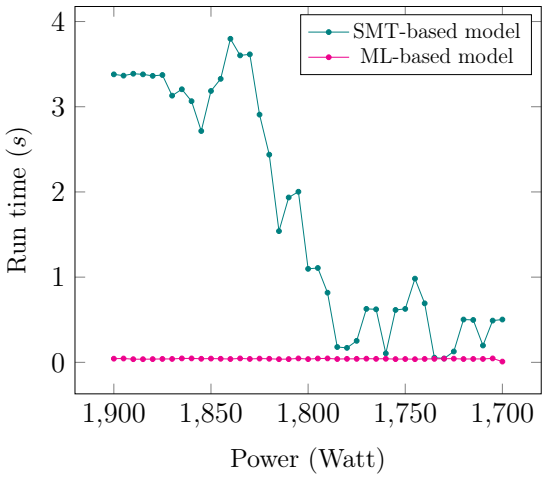
In this experiment, we consider the two resources power (PWR) and reliability (REL). Where PWR $\in \mathbb{R}$ and REL $\in \mathbb{D}$. We assign three possible quality levels to each node. Table 7.3 shows different power consumption values for all possible quality levels in each node.



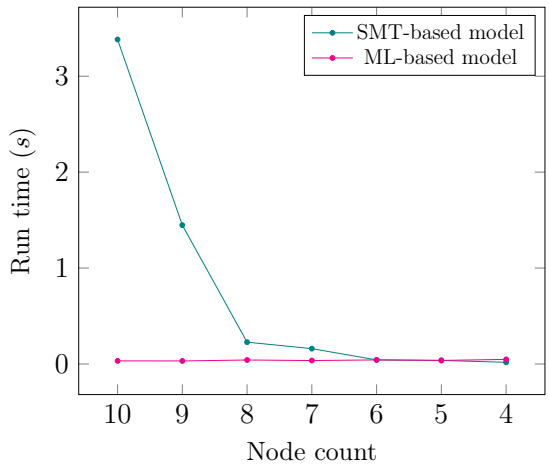
(a) Naive vs. optimized algorithm.



(b) Reliability vs. power.



(c) Run time vs. power.



(d) Run time vs. reliability.

Figure 7.1 Synthetic experiment results.

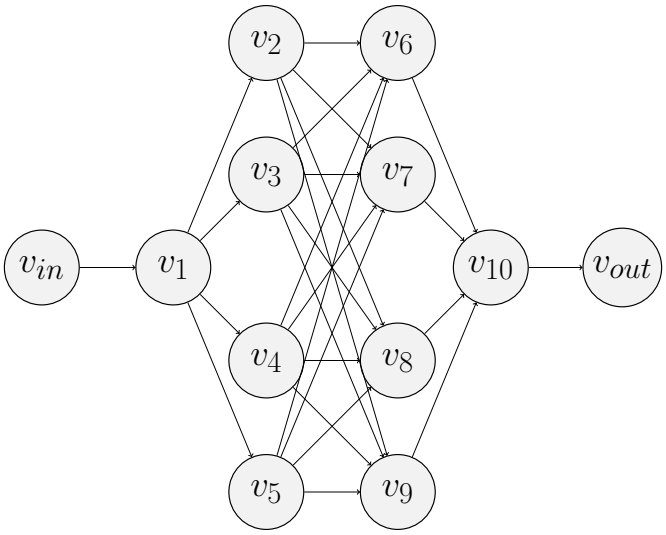


Figure 7.2 A producer-consumer network of 8 nodes.

	q_1	q_2	q_3	q_4	q_5
v_{in}	-	-	-	-	-
v_1	200	195	190	185	180
v_2	185	180	175	170	165
v_3	190	185	180	175	170
v_4	195	190	185	180	175
v_5	180	175	170	165	160
v_6	195	190	185	180	175
v_7	185	180	175	170	165
v_8	180	175	170	165	160
v_9	190	185	180	175	170
v_{10}	200	195	190	185	180
v_{out}	-	-	-	-	-

Table 7.3 Nodes $v_{[1,9]}$ power consumption (in watts) for different quality levels.

7.5.1.3 Machine Learning Setup

For our machine learning dataset, we generate 500 data samples using our SMT-based solution. each sample contains a randomly selected PWR value, and the optimized quality levels for the 10 nodes in Figure 7.2. We train our ANN with this dataset for 10 epochs (full iterations) using the k-fold cross validation method described in Section 7.4.

7.5.1.4 Experimental Results

We now run a variety of experiments on the setup described above and report our findings below.

Naive Model vs. Optimized Model. vs. Machine Learning Model. First we run the model to find the best possible reliability, given bounds on other resources in the producer-consumer network. We want to observe the improvement in run time between a model that performs regular constraint solving, and a model that performs constraint solving using the binary probing technique shown in Algorithm 7.1. To this end, we assign the PWR resource $b_{\mathbb{V}}^{\text{PWR}}$ to $\langle 0, 1835 \rangle$ and run our solvers. As shown in Figure 7.1a, using the naive (brute force) technique, we get a best reliability value of 0.855 within a run time of 18.855 seconds, whereas using the binary probing technique, we get a best reliability value of 0.857 within a run time of 3.185 seconds. Using our machine learning model, we get a best reliability value of 0.839 in under 0.036 seconds.

Reliability vs. Power. We now observe the tradeoff between reliability and power. We start off by assigning the PWR resource $b_{\mathbb{V}}^{\text{PWR}}$ to $\langle 0, 1900 \rangle$. We can observe in Figure 7.1b the burndown of reliability as we tighten the power bound by 5 watts on each iteration. We stop at $\langle 0, 1700 \rangle$ when the network is no longer functional due to the given power being lower than the minimal requirement. In this scenario, our machine learning-based model report a more uniform reliability drop than our SMT-based model. Figure 7.1c demonstrates the run time measurements for the same experiment. As the available power is reduced, the search space for the SMT solver gets smaller as well due to having to check for fewer valid configurations. Which is why a gradual decline in run time can be observed for the SMT-based model. However, the machine learning-based model reports the results through inference, and therefore show very little variation in run time.

Run time vs. Node availability. In this experiment, we observe the effect of node availability, and by extension overall reliability on run time. To this end, we assign the PWR resource $b_{\mathbb{V}}^{\text{PWR}}$ to $\langle 0, 1835 \rangle$ and reduce reliability of the nodes v_5, v_2, v_3, v_8, v_7 and v_9 to 0 one node at a time in the given order. Figure 7.1d shows the run time of this experiment. As expected, as more nodes become inactive, the overall run time for the solver decreases for the SMT-based model. However, similar to the previous observation, for the ML-based

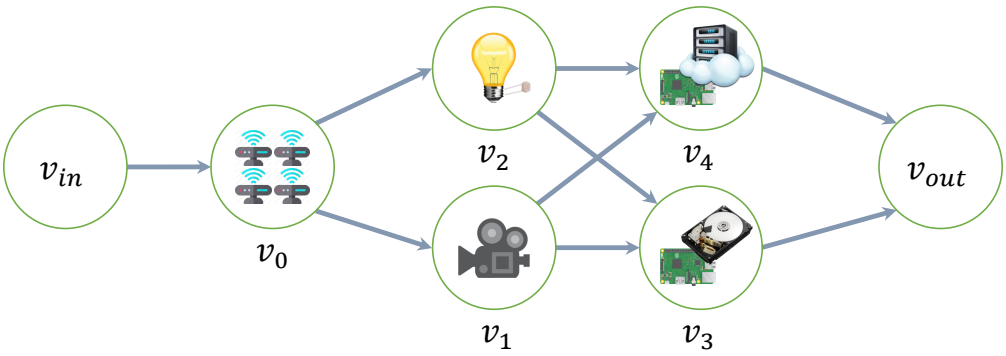


Figure 7.3 A Multi-Layer Network of Raspberry Pi Devices.

model, the run time remains steady. Note that removing any more nodes from the network will render the network inactive, as the solver will fail to find any valid path from v_{in} to v_{out}.

7.5.2 Case Study

In this subsection, we introduce our case study on a real world layered sensor network, where each sensor is tasked with a streaming or processing job, and is operated with a Raspberry Pi device.

7.5.2.1 Experimental Setup

We construct our layered sensor network with five nodes as shown in Figure 7.3, where v_0, v_1 and v_2 are producer nodes, and v_1, v_2, v_3 and v_4 are consumer nodes in $\mathbb{V}$. Just like before, we add two place holder nodes v_{in} and v_{out} such that, v_{in} has an outgoing edge to v_0, and v_{out} has two incoming edges from v_3 and v_4. Below we explain the streaming tasks of each device, along with the resources and quality levels.

Motion Sensor Node (v_0). This node is comprised of four motion sensors that are able to detect objects and movement. When any one of these motion sensors are activated, v_0 sends an activation signal to its subsequent nodes. Table 7.4a shows resource consumption for node v_0 under different quality levels. In this case, the quality levels simply indicate the number of active motion sensors. Motion sensors are fairly reliable under normal operational circumstances, which is why v_0 has high reliability as long as at least one sensor is active.

	Active Sensors	Reliability	Power Usage
q^5	0	0	450
q^4	1	0.94	455
q^3	2	0.96	460
q^2	3	0.98	465
q^1	4	1	470

(a) Quality levels and resource usage of v_0.

	Resolution	Bandwidth	Power Usage
q^6	256 x 144	15	446
q^5	426 x 240	35	476
q^4	640 x 360	50	488
q^3	854 x 480	125	500
q^2	1280 x 720	275	512
q^1	1920 x 1080	2500	566

(b) Quality level and resource usage of v_1.

	Illuminance	Brightness	Power Usage
q_{11}	120000	0	472
q_{10}	108000	10	479
q_9	96000	20	486
q_8	84000	30	493
q_7	72000	40	500
q_6	60000	50	507
q_5	48000	60	514
q_4	36000	70	521
q_3	24000	80	528
q_2	12000	90	535
q_1	0	100	542

(c) Quality level and resource usage of v_2.

	Resolution	Bandwidth	Power Usage
q^6	256 x 144	15	458
q^5	426 x 240	35	464
q^4	640 x 360	50	470
q^3	854 x 480	125	476
q^2	1280 x 720	275	482
q^1	1920 x 1080	2500	488

(d) Quality level and resource usage of v_3.

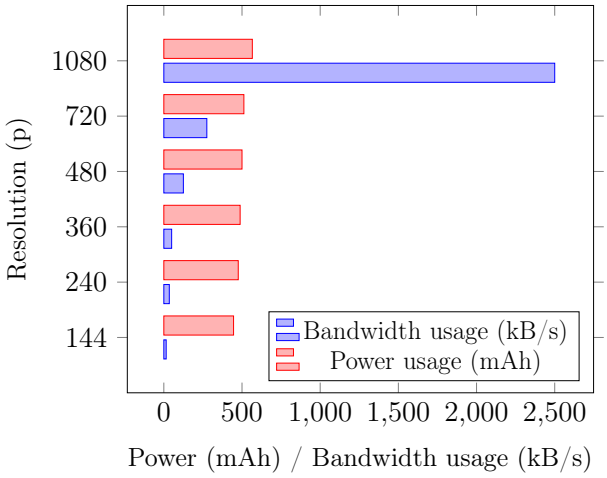
	Resolution	Bandwidth	Power Usage
q^6	256 x 144	30	452
q^5	426 x 240	70	452
q^4	640 x 360	100	452
q^3	854 x 480	250	452
q^2	1280 x 720	550	452
q^1	1920 x 1080	5000	452

(e) Quality level and resource usage of v_4.

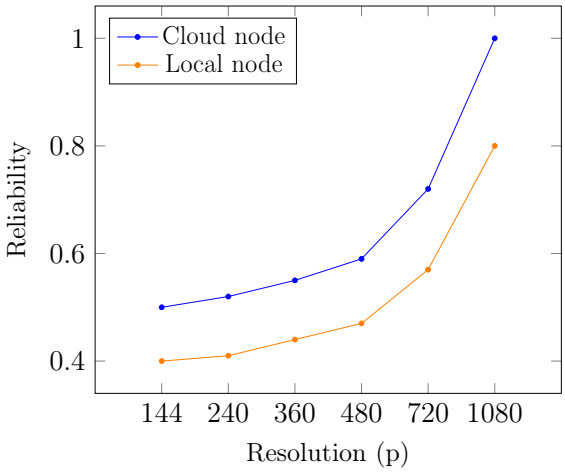
	MOTION	CAM	LIGHT	LOCAL	CLOUD
1410	q_1	q_1	q_1	q_1	q_1
1405	q_2	q_1	q_1	q_1	q_1
1400	q_3	q_1	q_1	q_1	q_1
1395	q_4	q_1	q_1	q_1	q_1
1390	q_4	q_1	q_1	q_2	q_1
1385	q_4	q_1	q_1	q_3	q_1
1380	q_4	q_1	q_1	q_4	q_1
1375	q_4	q_1	q_1	q_5	q_1
1370	q_4	q_1	q_1	q_6	q_1
1365	q_4	q_1	q_1	q_6	q_1
1360	q_4	q_1	q_1	q_6	q_3
1355	q_4	q_1	q_1	q_6	q_6

(f) Quality level change due to power.

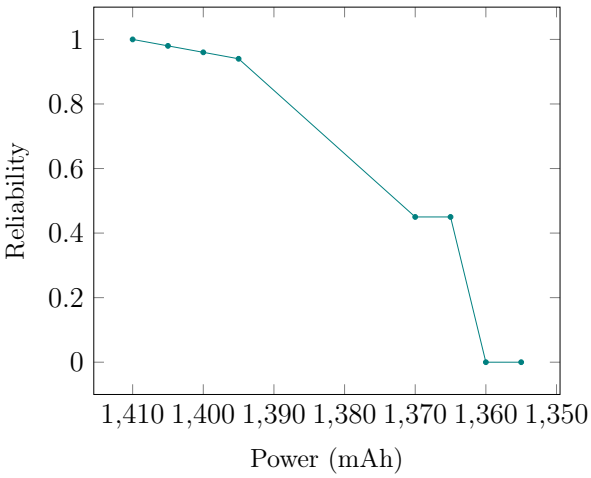
Table 7.4 Quality level tables for different nodes.



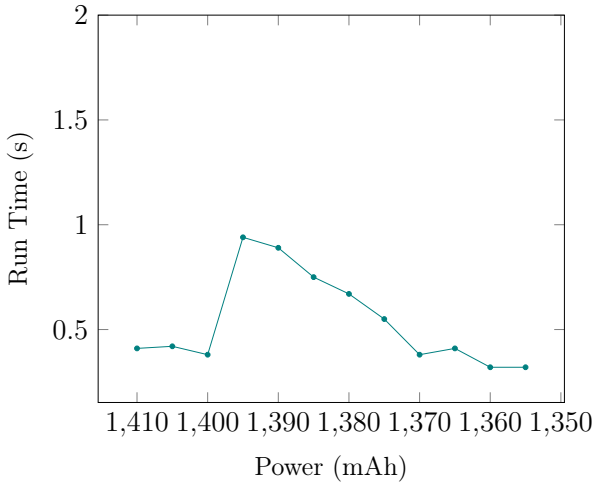
(a) Resolution vs. power.



(b) Local vs. cloud processing.



(c) Reliability vs. power.



(d) Run time vs. power.

Figure 7.4 Case study results.

Camera Node (v_1). This node is tasked with operating a 5MP camera at varying resolution/bitrate, and is activated upon receiving an activation signal from v_0. The resolution/bitrate of the captured stream is changed at runtime, which allows us to map the different resolution/bitrate modes to the quality level of node v_1. The node consumes two other resources: bandwidth and power. At the highest quality level, this node uses $\approx$ 2.5 Mbit/s and $\approx$ 566 mAh. Table 7.4b shows resource consumption for node v_1 under different quality levels, and Figure 7.4a shows the tradeoff between stream resolution vs. power consumption and bandwidth usage. Note that change stream resolution/bitrate is not always proportional to bandwidth. This is due to video compression standards (e.g H.264).

Illuminance Detection Node (v_2). This node is connected to an illuminance detector, and a smart light bulb with adjustable brightness. As both of these devices are powered and operated by node v_2 with minimal data transfer and delay, we consider them to be a part of node v_2 itself. Similar to v_1, this node is activated upon receiving an activation signal from v_0. Depending on how dark or bright the general area that is being streamed by the camera on node v_1 is, node v_2 adjusts the brightness of the smart bulb accordingly. We keep the illuminance detector some distance away from the network, in order to prevent light flickering due to feedback loop. Table 7.4c shows resource consumption for node v_1 under different quality levels.

Local Storage Node (v_3). This node compresses, checksum verifies, and stores the video stream received through edge (v_1, v_3) into a secure external hard disk drive. Table 7.4d shows resource consumption for v_1 under different quality levels.

Cloud Storage Node (v_4). Similar to node v_3, node v_4 is also tasked with storing the video stream received through edge (v_1, v_3). However, instead of storing it locally, the stream is uploaded directly to a cloud storage. Offloading the compression and checksum verification task to the cloud allows node v_4 to consume minimal power, at the cost of additional bandwidth. Table 7.4e shows resource consumption for node v_4 under different quality levels.

Note that the bandwidth requirement for node v_4 is double the amount in comparison to that of v_3 at the same quality level. This is due to the fact that when v_4 receives a video stream from node v_1, it uploads the same stream to the cloud, effectively doubling the required bandwidth. Furthermore, we assume that while maintaining similar qualities and stream, node v_4 generally more reliable than node v_3 for being able to store and backup data in the cloud as shown in Figure 7.4b.

7.5.2.2 Machine Learning Setup

For our machine learning dataset, similar to our synthetic experiments, we generate 500 data samples using our SMT-based solution. each sample contains a randomly selected PWR value, and the optimized quality levels for the 5 nodes in the Pi network shown in Figure 7.3. We train our ANN with this dataset for 10 epochs (full iterations) using the k-fold cross validation method described in Section 7.4.

7.5.2.3 Experimental Results

We run a variety of experiments on the setup described above and report our findings below.

Reliability vs. Power. We now observe the tradeoff between reliability and power in our multi-layer network of nodes. We start off by assigning the PWR resource $b_{\mathbb{V}}^{\text{PWR}}$ to $\langle 0, 1410 \rangle$ as shown in Figure 7.4c. From this point we measure the best obtainable reliability and tighten the bound by 5 mAh on each iteration in a similar manner as our synthetic experiment. Figure 7.4d shows the run time for the same experiment. The run time is low at the beginning due to the system having adequate power flow to operate all nodes at a near maximum reliability, and therefore having very few SMT constraints to solve. Observe that our machine learning-based model exhibits similar behavior as seen during our synthetic experiments. The reliability drop is a steady decline, whereas the run time does not very to a great degree. However, for bother models this changes when $b_{\mathbb{V}}^{\text{PWR}}$ is $\langle 0, 1410 \rangle$ and onward, as the available power is no longer sufficient for all nodes to operate at maximum quality

and reliability.

We now observe the changes in quality levels for which the burndown in reliability has occurred. From Table 7.4f, we can see that the model gradually lowered the quality levels of v_0 to q_4 first due to the small difference in reliability between each quality level. Afterwards, the quality level of v_3 (local node) is lowered instead of v_4 (cloud node) due to v_3 consuming more power than v_4. Once the quality level of v_3 has reached the lowest point, lowering the available power further finally caused the model to lower the quality level of v_4. Finally, below 900 mAh, the available power was not sufficient for keeping the nodes running, even at the lowest quality levels, and therefore, the network was shutdown.

We run the same experiment again with available bandwidth as the tightening resource. In this case, we observe the exact opposite behavior, where the quality level of v_4 (cloud node) was lowered first, and then v_3 (local node). This is due to the fact that v_4 requires double the bandwidth when compared to v_3, as shown in Table 7.4e.

7.6 Conclusion

In this chapter, we developed a generalized model of a streaming sensor network CPS as a network of producers and consumers. Our approach incorporates tradeoffs between output quality and resource utilization. These tradeoffs were articulated as a multi-objective optimization problem with the goal of minimizing resource consumption while maximizing the reliability (and quality) of devices (or tasks) in a network. To tackle the aforementioned optimization challenge, we provided an efficient technique based on constraint solving utilizing SMT-solvers to identify the ideal processing quality selection for each node in the network while respecting resource limitations and minimizing error. We further improve this work by incorporating machine learning and dramatically speed up the resource optimization speed. Since sensor network applications frequently require stream processing, which entails a complicated network of processing nodes where data is collected, analyzed, and then communicated to succeeding nodes, this is a significant problem. We put our work into practice, and provide the results of an experiment using an IoT device network.

CHAPTER 8

RELATED WORK

In this section, we summarize a portion of the vast quantity of work that has been done thus far in the field of distributed CPS that has influenced this dissertation, going as far back as the origins of distributed monitoring.

8.1 Lattice-based Distributed Monitoring

Early work on distributed monitoring involving predicate detection in distributed computing [53, 90, 115] are known to be NP-complete. To propose a more efficient solution, a *computation slicing* [91] technique is used to reduce the computation size, which in turns results in smaller search space in predicate detection, as far as state space is concerned. This work is later extended to an online distributed monitoring algorithm [26].

Lattice-based distributed monitoring solutions generally suffer from two shortcomings: (1) having to handle an enormous amount of concurrent states, and (2) lacking methods to handle temporal properties. A methodology for detecting Basic Temporal Logic [99] partially addresses the latter issue by providing methodologies for detecting a subset of temporal operators in distributed systems. A bound-based monitoring approach [130] addresses state space problem, and then later extended to a more efficient technique [116] that utilizes SAT solvers. In our work, we avoid using any lattice-based distributed monitoring approaches, and by extension, avoid needing to handle a unamanagable amount of concurrent states.

8.2 Runtime Monitoring in CPS

Accurate time-keeping for CPS was thoroughly investigated by the Roseline project [98]. Roseline team addresses the problem that local clocks have little, if any, knowledge of the quality of time needed by the software, nor any ability to adapt to it. They achieve this by rethinking and re-engineering how the knowledge of time is handled across a computing system's hardware and software and driving accurate timing information deep into the software system.

Assuming perfect time synchrony, an offline toolbox called S-TALIRO [5] is introduced by its developers, that searches for trajectories of robust MTL [67] semantics. S-TALIRO can analyze arbitrary Simulink models or user defined functions that model the system, and operate using randomized testing based on stochastic optimization techniques like Monte-Carlo methods [87] and Ant-Colony optimization [37].

An online monitoring technique [35] of STL [36] for continuous and hybrid systems employs an efficient algorithm for computing the robustness degree in which a piecewise-continuous signal satisfies or violates an STL formula.

An efficient monitoring solution [34] is proposed by its authors that utilizes Dynamic Programming algorithms for online monitoring of the state robustness of MTL [67] specifications with past time operators. The authors provide an approach for predictive monitoring by computing the robustness of MTL with unbounded past and bounded future temporal operators over sampled traces of CPS. However, in order to do so, prior knowledge the full dynamical model of the system is required.

Our work relating to predicate detection is closer to the work involving an online monitoring approach [33], where a robust online monitoring of partial traces is formalized. However, this work assumes worst-case a priori bounds on signal values, but without factoring system dynamics.

A logic called differential dynamic game logic [105] is a new logic that aims to demonstrate how the satisfaction of a temporal property is affected by imperfect implementations. This work is similar to a conformance testing framework [1], where the authors quantify the closeness between two systems via a distance measure between their outputs, and study of how the satisfaction of a temporal property is affected by timing inaccuracies.

The a control-theoretic software monitoring solution [83] is proposed by its authors for coordinating time predictability and memory utilization in runtime monitoring of systems that interact with the physical world. This method maximizes memory utilization being employing a minimally intrusive monitoring tactic.

A tool called Brace [133] is introduced by its developers, where using the tool, users can attempt to minimize false positives and false negatives, while trying to stay under a given threshold for computation overhead of CPS. The authors do not provide any guarantees of completely eliminating false positives or false negatives, only minimizing them. This aspect is different from our approach on monitoring distributed signals in CPS, due to the fact that our approach guarantees no false positives.

Another tool called ModelPlex [89] is introduced as a method for ensuring verification results for models. ModelPlex also allows the said models to account for the effect of environmental variances and disturbances on a CPS, while considering only the relevant part of surrounding physics.

In the medical field, a specification language DRTV [63] in order to specify vital real-time data sampled by medical devices. DRTV also allows for runtime monitoring temporal properties originated from clinical guidelines.

A hybrid approach to runtime monitoring in CPS called Extended Hidden Markov systems [112] is explored, where the systems under inspection are comprised of both integer-valued and real-valued variables.

While the above works propose various techniques and tools for monitoring CPS, they do not account for partial synchrony, nor system dynamics in their monitoring methodologies.

8.3 Asynchronous Distributed Monitoring

The notion of computational slide [91] introduces the ability to monitor distributed systems in an asynchronous setting. In this approach, the slice of a computation with respect to a predicate is a sub-computation with the least number of consistent cuts that contains all consistent cuts of the computation satisfying a given predicate. This work is later extended to a distributed setting [26], where a distributed algorithm is presented for computing the slice of a distributed computation with respect to a regular predicate.

The work on distributed monitoring of concurrent and asynchronous systems [14] investigates the problem of distributed monitoring under time asynchrony, with application to

distributed fault management in telecommunications networks. To this end, the authors combine compositional unfoldings to handle concurrency, and a variant of graphical algorithms and belief propagation, originating from statistics and information theory. This work is later further extended [44], where the authors study the diagnosis of distributed asynchronous systems with concurrency. In this work, diagnosis is performed by a peer-to-peer distributed architecture of supervisors. This approach relies on Petri net [103] unfoldings and event structures, as means to manipulate trajectories of systems with concurrency.

A tool called DIANA [109] is introduced by its developers in order to monitor temporal properties of distributed systems. The authors use past time distributed temporal logic (a variant of past time linear temporal logic) as the specification language. In this approach, the notion of knowledge vector is introduced where each process is kept aware of other processes' local states. This approach, however, suffers from producing false negatives.

A decentralized runtime verification technique for LTL specifications [96] demonstrates a method for runtime verification of asynchronous distributed programs for the 3-valued semantics of LTL specifications. This approach however, also suffers from false negatives results. On the other hand, in a temporal logic predicate detection approach [99] the authors introduce the concept of a compact representation of all global cuts that satisfy a predicate.

The approaches mentioned above all operate within fully asynchronous setting. To the contrary to these approaches, we leverage a practical assumption and employ an off-the-shelf clock synchronization algorithm to limit the time window of asynchrony.

A method for designing parallel algorithms [54] is proposed to solve constrained combinatorial optimization problems like marriage problem, shortest path problem, market clearing price problem, and so on. The authors achieve this by transforming these problems into a search problem, where an element that satisfies an appropriate predicate in a distributive lattice is obtained.

An approach for detecting latent bugs caused by concurrency and race conditions among concurrent processes [116] is by its authors. In this work, the authors propose a method

for detecting errors and monitoring system constraints in partially synchronous distributed systems using a monitoring framework with SMT as its foundation.

In the work involving runtime monitoring of LTL formulas for synchronous distributed systems in the absence of a central data collection point [9], the authors propose an approach where LTL formulas are decomposed into sub-formulas, such that satisfaction or violation of specifications can be detected by local monitors alone. This work is later expanded upon with the introduction of a synchronous global clock [28], in which monitors are organised as a tree across the distributed system, and each child feeds intermediate results to its parent. A similar approach using LTL, but for stream runtime verification of CPS [31] is later proposed in. However, these approaches assume perfectly synchronous clocks, which is rarely achieveable.

The four-valued logic Runtime Verification Linear Temporal Logic RV-LTL [11] introduces a logic, where the system behavior either (i) satisfies the monitored property, (ii) violates the property, (iii) will presumably violate the property, or (iv) will presumably conform to the property in the future, once the system has stabilized. This work is later improved upon, where a fault tolerant verification technique LTL_{2k+4} [16] is proposed for asynchronous systems. In our automata-based monitoring technique, we used LTL_3 over four-valued LTL or LTL_{2k+4}, because the unknown verdict in LTL_3 was sufficient for our monitoring purposes, and the distinction between 'will presumably violate the property' and 'will presumably satisfy the property' served no additional benefit.

8.4 Synchronous Distributed Monitoring

Two approaches for runtime monitoring of LTL formulas have been studied by the authors of the monitor framework THEMIS [41]. The first approach introduces a data structure that keeps track of the execution of an automaton, has predictable parameters and size, and guarantees strong eventual consistency. The second approach defines decentralized specifications wherein multiple specifications are provided for separate parts of the system. The framework THEMIS can be used to analyze systems using the two approaches.

An adaptive synchronous parallel method for distributed machine learning [131] is explored, where the performance monitoring model adaptively adjusts the synchronization method of each computing node with the parameter server by taking into account the whole performance of each node, ensuring improved accuracy. Furthermore, this technique guards against the machine learning model being influenced by irrelevant tasks in the same cluster.

A hybrid approach to monitoring is taken by the authors of the monitoring tool called SMEDL [132], where low-level properties are checked synchronously, while higher-level ones are checked asynchronously. SMEDL can be used to construct and deploy monitors based on an architecture specification.

The specification language intended for industrial use called LOLA [119] is proposed, where the authors provide a syntactic characterization of efficiently monitorable specifications, for which the space requirement of the online monitoring algorithm is independent of the size of the trace, and linear in the specification size. can express properties involving both the past and the future.

Both online and offline verification techniques using temporal logics [107] are studied by the authors of the specification language LOLA and present in detail the online and offline monitoring algorithms. To this end, the authors use temporal logic for Stream Runtime Verification [17].

A novel efficient two-layered monitoring technique [119] is proposed by its authors that aims to overcome the time and space constraints introduced by most synchronous monitoring approaches. The first layer is imperfect yet effective, whereas the second layer is exact but (relatively) ineffective. The two-layered monitor also supports the usage of O(1) sized Hybrid Logical Clocks. Another approach that aims to overcome the time and space constraints is a monitoring method that incorporates a control idea of synchronization on CPS [60], which include dividing a node's main loop program into several processes and adopting twice trigger signals to activate synchronization control.

The impact of synchronous and asynchronous monitoring instrumentation on runtime

overheads in the context of a runtime verification framework for actor-based systems [21] is thoroughly studied, and the authors show that, in such a context, asynchronous monitoring incurs substantially lower overhead costs. They also demonstrate how, for certain properties that require synchronous monitoring, a hybrid approach can be used that ensures timely violation detections for the important events while, at the same time, incurring lower overhead costs that are closer to those of an asynchronous instrumentation.

A solution to the decentralized monitoring problem for the more general setting of stream runtime verification [31] is provided by its authors, and a property on specification is also introduced here that guarantees that the online monitoring can be performed with bounded resources.

An algorithm for distributing and monitoring LTL formula [10] employs a technique where satisfaction or violation of specifications can be detected by local monitors alone, even when the system's implementation details are hidden to the user.

However, these approaches have the shortcoming of assuming a global clock across all distributed processes. Predicate detection for asynchronous system [114] has been studied extensively where 3 distinct detection modalities are achieved by introducing the notion of 'definitely occurred before' and 'possibly occurred before' event orderings. However, doing so causes the assumption needed to evaluate happen-before relationship to be too strong.

In this dissertation, we utilize HLC, which not only is more realistic but also decreases the level of concurrency. Finally, an automata-based fault tolerant verification technique [64] is proposed for synchronous systems with no clock skews across the distributed processes. A fault-tolerant distributed membership protocol for the determination of the set of active nodes in a synchronous distributed real-time systems [66] is presented. We use a clock synchronization algorithm which guarantees bounded clock skews. Our solution is also SMT based and to our knowledge this is the first SMT based distributed monitoring algorithm for LTL, which results in better scalability.

8.5 Partially Synchronous Distributed Monitoring

In the context of monitoring partially synchronous systems, the feasibility of monitoring partially synchronous distributed systems [116] in order to detect latent bugs was first investigated. The authors provide a monitoring framework, where both the system constraints, and the latent bugs are modeled as SMT formulas. The latent bugs are identified using SMT solvers. This technique was later generalized to full LTL [49], where the presence of latent bugs are detected using SMT solvers in a discrete setting. The authors introduce two monitoring techniques where the specification in the LTL is either represented by a deterministic finite automaton, or, a progression-based formula rewriting technique to reduce the distributed runtime verification problem to an SMT problem.

A tool for identifying data races in distributed system traces called SPIDER [102] is introduced for handling non-deterministic discrete event orderings. This is an automated tool that can be used to identify data races in distributed system traces. However, these approaches cannot fully capture the continuous-time and continuous-valued behavior of CPS.

There is extensive work in identifying a subclass of systems [22], for which convergence features may be confirmed using the proof of convergence for the related discrete-time shared state system. The method is extended to systems in which an agent's state develops continuously over time. The proof approach was formalized in the PVS interface for timed I/O automata and used to verify the convergence of a mobile agent pattern formation algorithm.

A failure detector for partially synchronous distributed systems [121] is proposed by its authors, where the authors present an alternative failure detector algorithm, which is based on a clock synchronization algorithm.

A solution to the processor group membership problem [29] is achieved by precisely specifying the processor problem in order to define the system model and failure assumptions. The author then provides two protocols for solving this problem.

A technique to monitor predicates on a partially synchronous distributed system by retiming continuous signals [95] is explored. While this approach improves monitoring efficiency

by levering knowledge about system dynamics, it is limited to only being able to monitor predicates, and cannot capture temporal behavior.

A method for runtime monitoring of blockchain executions for partially synchronous distributed computations [50] is proposed where the specification language is metric temporal logic [67].

The effects of the impedance mismatch between the monitor and the underlying program for the detection of conjunctive predicates [130] is analyzed. An interesting observation of this work is that the authors identify a small interval where the monitor assumptions are hypersensitive to the underlying program environment.

A domain specific language called PSync [38] based on the Heard-Of model [24, 25], is demonstrated, where asynchronous faulty systems are viewed as synchronous ones with an adversarial environment that simulates asynchrony and faults by dropping messages.

While the approaches above provide various techniques for monitoring partially synchronous discrete systems, they are unable to fully capture the continuous nature of CPS.

8.6 Decentralized Distributed Monitoring

There is a rich literature dealing with decentralized predicate detection in the discrete-time setting. These works range from detection of regular discrete-time predicates [26] to detecting lattice-linear predicates over discrete states [54]. There is recent work on performing detection on a regular subset of Computation Tree Logic [108] that aims to avoid the state explosion problem. Literature books by Garg [52] and Singhal [68] elaborate extensively on decentralized monitoring in discrete-time settings. By contrast, we are concerned with monitoring continuous-time signals in decentralised setting, which have uncountably many events and necessitate new techniques. For instance, one cannot iterate through events as done in the discrete setting.

The recent works [94, 95] do monitoring of temporal formulas over partially synchronous analog distributed systems, that is, they only find one satisfaction, not all. Moreover, their solution is centralized.

Generally, there is a plethora of work to be found on monitoring temporal logic properties, especially Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL). Notably, these works involve using a three-valued MTL for monitoring in the presence of failures and non-FIFO communication channels [7], monitoring satisfaction of an LTL formula [10], using a three-valued LTL for distributed systems with asynchronous properties [96], using a tableau technique for three-valued LTL [8], and finally, using a past-time distributed temporal logic which emphasizes distributed properties over time [109]. However, all these methods either focus on centralized monitoring or work in discrete settings. In our work, we provide methodologies for decentralized monitoring in continuous time settings.

8.7 Monitoring Reliability in CPS

Resource trade-off is a broadly studied with respect to monitoring reliability in CPS. Such as the work exploring the trade-offs between power and reliability of Wireless Sensor Networks [30]. This work proposes a model for evaluating the reliability of WSNs considering the battery level as a key factor.

The problem of modeling and evaluating the coverage-oriented reliability of CPS subject to common-cause failures is explored [110], where the proposed methodology takes advantage of reduced ordered binary decision diagrams, which is similar to our binary probing technique.

A methodology based on an automatic generation of a fault tree [111] is proposed in order to evaluate the reliability and availability of CPS, when permanent faults occur on network devices.

One stream of work in CPS is concerned with security related trade-offs, where security comes at a cost of energy or performance. A relevant literature in this regard provides a classification of existing security concerns and researches [3].

A method to determine when to inject cryptographic checks without interfering with control tasks [75] is demonstrated by its authors, where the general idea behind the methodology is maximizing security checks while maintaining a predefined level of control quality. Another similar work is available where the authors propose a feedback scheduling tech-

nique for maintaining network quality of service in wireless sensor networks [125]. Both of these works fall under soft real-time constraints, where essentially security is traded off with deadline adherence.

A notable literature review further studies and elaborates on the challenges in designing reliable CPS [74]. The work emphasizes on the necessity of raising the level of abstraction in terms of designing reliable CPS, as the current networking technologies often do not provide adequate foundation for CPS.

There is a line of work in the parallel and distributed processing domain on distributing power resources efficiently. For example, a method to bound the energy consumption of a message passing interface program [106] is proposed, where the authors use a linear programming model that knows the execution time of jobs on machines and the effect of changing the frequency on their speedup. This work has been then extended to propose a scalable method to determine individual task power bounds in a distributed setting given a global power bound [84].

Our work involving resource consumption in a producer consumer network draws inspiration from existing work that addresses the problem of energy consumption in a producer consumer network using learning mechanisms to reduce the energy consumption of the overall system [82].

Many researchers target the problem of finding optimal energy savings without impacting performance. One such notable study observes the trade-offs between energy and delay for a wide set of applications [48]. The work also studies metrics to use to predict memory or communication bottlenecks. There exist multiple works that attempt to tackle this bottleneck problem on a single processor [62, 78]. These works mainly proposes an alternative Dynamic voltage and frequency scaling strategy that maintains the same performance at reduced energy consumption.

The work on formal control techniques for power-performance management [124] discusses the effectiveness of using control theory in power management. Furthermore, the series of

works [126, 127, 128] construct an integer linear programming (ILP) model to determine the minimum energy consumption that a program can consume on a single processor.

Our work on multi-resource multi-node optimization problem is similar to the work on the management of energy security trade-off in a distributed cyber-physical system [120], in the sense that our approach is more generalized.

CHAPTER 9

CONCLUSION

In this chapter, we summarize our work and highlight our contributions for each methodology. We then discuss our current ongoing work and short term goals. Finally, we conclude by exploring potential future avenues of research that could be logical next steps of our work.

9.1 Summary

We begin with distributed runtime monitoring in this dissertation. Our proposed techniques take an LTL formula and a distributed computation as input and, assuming a bounded clock skew among all processes, chops the computation into multiple segments before applying the automata-based and progression-based monitoring algorithms implemented as an SMT decision problem to verify the correctness of the formula. We carried out rigorous synthetic experiments using LTL formulas of varying complexity. Although we attempted to keep our synthetic experiments as close to real world scenarios as possible, we acknowledge that in these synthetic experiments (as well as any synthetic experiments in our following works), there could be missing environmental variables (which would otherwise be present in real world scenarios) that could influence our run time. However, to partially account for this shortcoming, we carried out case studies on Cassandra consistency circumstances and NASA air traffic control dataset.

Following that work, we show an online predicate detection strategy for distributed signals that do not share a global clock. To make the problem tractable, we use causality analysis between real-valued signals, a reasonable assumption on maximum clock skew among local clocks, and rough knowledge of system dynamics. We also studied the influence of signal dynamics information on monitoring efficiency. By testing on a real network of autonomous cars, a simulated network of UAVs, and a simulated water distribution system, we discovered numerous noteworthy discoveries. Our method may be used to successfully monitor a distributed CPS in an online setting.

For distributed CPS, we presented an approach for monitoring specifications expressed in

signal temporal logic (STL), where continuous-time and valued signals from a group of agents do not share a global clock. Our method relies on an off-the-shelf clock synchronization solution, such as NTP, to ensure a maximum constrained clock skew across all agents in the system. Leveraging our work in predicate detection, we also presented a signal retiming approach that effectively aligns continuous signals in order to detect potential STL violations. To address the complexity, we simplify our runtime monitoring problem to a basic SMT solving problem and cut the distributed signals into a sequence of smaller segments. We also presented a formula progression approach similar to our work with distributed systems, which takes a distributed signal and an STL formula as input and outputs another STL formula that depicts the formula's progress through the signals. We also presented experimental results from the monitoring of an unmanned aerial vehicle (UAV) fleet and a water distribution system.

We then extend our work to decentralized monitoring, where we perform online conjunctive predicate detection for distributed signals. Our algorithm returns all possible violations of the predicate, which in turn allows us to identify and eliminate bugs from distributed systems regardless of actual clock drift.

Finally, we provided a generalized model of a streaming network CPS as a producer-consumer network. Our approach incorporates tradeoffs between output quality and resource utilization. These tradeoffs were articulated as a multi-objective optimization problem with the goal of lowering resource utilization while maximizing the reliability (and quality) of devices (or jobs) in a network. To tackle the aforementioned optimization challenge, we provided an efficient technique based on constraint solving utilizing SMT-solvers to identify the ideal processing quality selection for each node in the network while respecting resource limitations and minimizing error. This is a significant problem since network applications frequently require stream processing, which entails a complicated network of processing nodes where data is collected, analyzed, and then communicated to following nodes. We have fully implemented our approach and shown testing findings on an IoT device network.

9.2 Ongoing Work

We have thus far discussed methodologies for monitoring various formal specifications on partially synchronous distributed CPS under both centralized and decentralized monitoring settings. However, in every case, we assume that all the agents in these systems are *honest*, that is, the agents follow the intended behaviors and protocols without malicious intent. Our current work involves designing *secure* monitoring techniques for both centralized and decentralized distributed CPS, where ensuring data privacy is the primary objective.

We explain the necessity of data privacy in CPS with the following example. Alice uses health monitoring wearables to measure her heart rate, blood glucose level, etc. Alice's hospital has a server (monitor) that would like to monitor Alice's health data, and if a certain specification is met (e.g., Alice's heart rate is above a threshold and glucose level is below a threshold), send an alert to Alice's caregiver. However, Alice does not wish to reveal her personal health data to the monitor, and the monitor does not want to reveal specification to Alice. In other words, both Alice and the monitor wish runtime verification to be performed on Alice's data using the monitor's specification, while keeping each party's data private.

9.2.1 Monitoring with Secure Multi-Party Computation

Secure Multi-Party Computation or simply Multi-Party Computation (MPC) [57] is a cryptographic protocol that allows multiple parties to jointly compute a function over their individual private inputs without ever revealing the said inputs to each other.

An example of MPC would be, consider a scenario where three friends, Alice, Bob and Charlie wish to compute their average salary while never disclosing their actual salary to one another. Let S_a, S_b and S_c be the salaries of Alice, Bob and Charlie respectively. Only Alice knows the value of S_a, Bob knows the value of S_b, and Charlie knows the value of S_c. Now Alice privately splits her salary amount into three random pieces, such that, $S_a = a_1 + a_2 + a3$. Bob and Charlie do the same, that is, $S_b = b_1 + b_2 + b3$ and $S_c = c_1 + c_2 + c_3$. Now, Alice shares a_2 with Bob, and a_3 with Charlie. Bob shares b_1 with Alice, and b_3 with Charlie.

Charlie shares c_1 with Alice, and c_2 with Bob. Now Alice computes $S_1 = a_1 + b_1 + c_1$, Bob computes $S_2 = a_2 + b_2 + c_2$, and Charlie computes $S_3 = a_3 + b_3 + c_3$. It should be noted that, it is impossible to extract any salary amounts from S_1, S_2 or S_3. However, if Alice, Bob and Charlie now share S_1, S_2 or S_3 with each other, add compute $(S_1 + S_2 + S_3)/3$, then the desired average salary can be obtained with any party revealing their salary amount to others.

While the above example is fairly straightforward, MPC provides more complex protocols with which arithmetic operations can be carried out without any loss of precision [42]. In our work we are mostly interested in performing addition and multiplication operations with MPC protocols efficiently, as generally rely on these two operations for our retiming approach (Recall 4.4e).

However, MPC does come with its own set of challenges. While addition protocols can be executed locally (i.e., on agents), multiplication protocols require agents to share *partial* data with each other multiple times before being able to compute the solution. Naturally, this is an issue for runtime verification, as there are various factors (e.g., network latency, workload, agent availability) that can influence communication delay, and by extension, run time.

We have already made significant headway in addressing some of the challenges presented by runtime verification using MPC. We hope to continue our work in this direction, and make significant progress in the near future.

9.3 Future Work

The work done in this dissertation paves way for various intriguing directions for further investigation. In this section we discuss the possible avenues of future work that are currently in our consideration.

First of all, for the monitoring approaches proposed in Chapter 3, 4, and 5, a study of the trade-off between accuracy and scalability can be conducted. We can define accuracy of

verdicts as follows:

$$\frac{actual\ number\ of\ correct\ verdicts - number\ of\ missed\ verdicts}{actual\ number\ of\ correct\ verdicts}$$

An interesting scope of research would be to observe and report the relationship between the degradation of accuracy and the improvement of runtime for the aforementioned monitoring techniques.

While monitoring predicates on distributed signals, our approach finds the first global states that violate a predicate in a segment. A crucial step in debugging distributed CPS is to find all such states. Thus, it is important to investigate data structures that can efficiently represent a set of global states of distributed continuous signals that violate a predicate. In the discrete setting, computation *slices* [91] are an example of such a data structure. One way to achieve this is by using the long-known notion of regions in timed automata [4].

Because we are reducing the monitoring problem to an SMT solution problem, the problem may become undecidable in some cases. The inevitable next step is to identify the STL piece where the problem is undecidable.

Another conceivable aim is for our monitor of the framework to become fully distributed, as we assume a central monitor in all cases in this dissertation. Having a centralized monitor also exposes our techniques to a single point of failure.

Furthermore, we have every reason to suspect that individual monitors in the system may have faults, such as crashing or reporting false verdicts. This necessitates the development of distributed *fault-tolerant* monitoring techniques.

For our approach on monitoring reliability of CPS, one obvious use of our method is to represent networks that are not necessarily acyclic, that is, the network may include feedback loops. Another intriguing line of study is to watch and report on the trade-off between monitor reliability and runtime overhead, as well as network communication.

BIBLIOGRAPHY

[1] Abbas, H., Mittelmann, H., and Fainekos, G. (2014). Formal property verification in a conformance testing framework. In *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 155–164. IEEE.

[2] Abiodun, O. I., Jantan, A., Omolara, A. E., Dada, K. V., Mohamed, N. A., and Arshad, H. (2018). State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938.

[3] Alguliyev, R., Imamverdiyev, Y., and Sukhostat, L. (2018). Cyber-physical systems and their security issues. *Computers in Industry*, 100:212–223.

[4] Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical computer science*, 126(2):183–235.

[5] Annpureddy, Y., Liu, C., Fainekos, G., and Sankaranarayanan, S. (2011). S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer.

[6] Barrett, C. and Tinelli, C. (2018). *Satisfiability modulo theories*. Springer.

[7] Basin, D., Klaedtke, F., and Zălinescu, E. (2015). Failure-aware runtime verification of distributed systems. In *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015)*, volume 45, pages 590–603. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[8] Bataineh, O., Rosenblum, D. S., and Reynolds, M. (2019). Efficient decentralized ltl monitoring framework using tableau technique. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–21.

[9] Bauer, A. and Falcone, Y. (2012). Decentralised ltl monitoring. In *International Symposium on Formal Methods*, pages 85–100. Springer.

[10] Bauer, A. and Falcone, Y. (2016). Decentralised ltl monitoring. *Formal Methods in System Design*, 48(1):46–93.

[11] Bauer, A., Leucker, M., and Schallhart, C. (2010). Comparing ltl semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674.

[12] Bauer, A., Leucker, M., and Schallhart, C. (2011). Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):1–64.

[13] Benndorf, M. and Haenselmann, T. (2016). Time synchronization on android devices for mobile construction assessment. In *The Tenth International Conference on Sensor*

Technologies and Applications. Thinkmind.

[14] Benveniste, A., Haar, S., Fabre, E., and Jard, C. (2003). Distributed monitoring of concurrent and asynchronous systems. In *International Conference on Concurrency Theory*, pages 1–26. Springer.

[15] Bhuyan, B., Sarma, H. K. D., Sarma, N., Kar, A., Mall, R., et al. (2010). Quality of service (qos) provisions in wireless sensor networks and related challenges. *Wireless Sensor Network*, 2(11):861.

[16] Bonakdarpour, B., Fraigniaud, P., Rajsbaum, S., Rosenblueth, D. A., and Travers, C. (2016). Decentralized asynchronous crash-resilient runtime verification. In *27th International Conference on Concurrency Theory (CONCUR 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[17] Bozzelli, L. and Sánchez, C. (2014). Foundations of boolean stream runtime verification. In *International Conference on Runtime Verification*, pages 64–79. Springer.

[18] Brunelli, D. and Caione, C. (2015). Sparse recovery optimization in wireless sensor networks with a sub-nyquist sampling rate. *Sensors*, 15(7):16654–16673.

[19] Cassandra, A. (2014). Apache cassandra. *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra*, 13.

[20] Cassandras, C. G. and Lafortune, S. (2008). *Introduction to discrete event systems*. Springer.

[21] Cassar, I. and Francalanza, A. (2015). On synchronous and asynchronous monitor instrumentation for actor-based systems. *arXiv preprint arXiv:1502.03514*.

[22] Chandy, K. M., Mitra, S., and Pilotto, C. (2008). Convergence verification: From shared memory to partially synchronous systems. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 218–232. Springer.

[23] Charron-Bost, B. (1991). Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16.

[24] Charron-Bost, B. and Schiper, A. (2006). The heard-of model: Unifying all benign failures. *EPFL Scientific Publications*.

[25] Charron-Bost, B. and Schiper, A. (2009). The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71.

[26] Chauhan, H., Garg, V. K., Natarajan, A., and Mittal, N. (2013). A distributed abstraction algorithm for online predicate detection. In *2013 IEEE 32nd International Symposium*

on Reliable Distributed Systems, pages 101–110. IEEE.

[27] Chen, H. (2017). Applications of cyber-physical system: a literature review. *Journal of Industrial Integration and Management*, 2(03):1750012.

[28] Colombo, C. and Falcone, Y. (2016). Organising ltl monitors over distributed systems with a global clock. *Formal Methods in System Design*, 49(1):109–158.

[29] Cristian, F. (1988). Agreeing on who is present and who is absent in a synchronous distributed system. In *1988 The Eighteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 206–207. IEEE Computer Society.

[30] Dâmaso, A., Rosa, N., and Maciel, P. (2014). Reliability of wireless sensor networks. *Sensors*, 14(9):15760–15785.

[31] Danielsson, L. M. and Sánchez, C. (2019). Decentralized stream runtime verification. In *International Conference on Runtime Verification*, pages 185–201. Springer.

[32] De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

[33] Deshmukh, J. V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., and Seshia, S. A. (2017). Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51(1):5–30.

[34] Dokhanchi, A., Hoxha, B., and Fainekos, G. (2014). On-line monitoring for temporal logic robustness. In *International Conference on Runtime Verification*, pages 231–246. Springer.

[35] Donzé, A., Ferrere, T., and Maler, O. (2013). Efficient robust monitoring for stl. In *International Conference on Computer Aided Verification*, pages 264–279. Springer.

[36] Donzé, A. and Maler, O. (2010). Robust satisfaction of temporal logic over real-valued signals. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 92–106. Springer.

[37] Dorigo, M., Birattari, M., and Stutzle, T. (2006). Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39.

[38] Drăgoi, C., Henzinger, T. A., and Zufferey, D. (2016). Psync: a partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices*, 51(1):400–415.

[39] Drone Life (2019). FAA UTM project: Decentralized uas traffic management

demonstration. *https://dronelife.com/2019/09/09/decentralized-uas-traffic-management-demonstration.*

[40] Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323.

[41] El-Hokayem, A. and Falcone, Y. (2020). On the monitoring of decentralized specifications: semantics, properties, analysis, and simulation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(1):1–57.

[42] Evans, D., Kolesnikov, V., Rosulek, M., et al. (2018). A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3):70–246.

[43] FAA (2019). DOT UAS initiatives. *https://www.faa.gov/uas/programs_partnerships/DOT_initiatives.*

[44] Fabre, E., Benveniste, A., Haar, S., and Jard, C. (2005). Distributed monitoring of concurrent and asynchronous systems. *Discrete Event Dynamic Systems*, 15(1):33–84.

[45] Fainekos, G. E. and Pappas, G. J. (2007). Robust sampling for mitl specifications. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 147–162. Springer.

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

[47] Fraigniaud, P., Rajsbaum, S., and Travers, C. (2020). 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.

[48] Freeh, V. W., Lowenthal, D. K., Pan, F., Kappiah, N., Springer, R., Rountree, B. L., and Femal, M. E. (2007). Analyzing the energy-time trade-off in high-performance computing applications. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):835–848.

[49] Ganguly, R., Momtaz, A., and Bonakdarpour, B. (2021). Distributed runtime verification under partial synchrony. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[50] Ganguly, R., Xuey, Y., Jonckheere, A., Ljungy, P., Schornsteiny, B., Bonakdarpour, B., and Herlihy, M. (2022). Distributed runtime verification of metric temporal properties for cross-chain protocols. *arXiv preprint arXiv:2204.09796.*

[51] Gareth, J., Daniela, W., Trevor, H., and Robert, T. (2013). *An introduction to statistical learning: with applications in R.* Spinger.

[52] Garg, V. (2002a). *Elements of Distributed Computing*. John Wiley & Sons.

[53] Garg, V. K. (2002b). *Elements of distributed computing*. John Wiley & Sons.

[54] Garg, V. K. (2020). Predicate detection to solve combinatorial optimization problems. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–245.

[55] Garg, V. K. and Chase, C. M. (1995). Distributed algorithms for detecting conjunctive predicates. In *Proceedings of 15th International Conference on Distributed Computing Systems*, pages 423–430. IEEE.

[56] Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias/-variance dilemma. *Neural computation*, 4(1):1–58.

[57] Goldreich, O. (1998). Secure multi-party computation. *Manuscript. Preliminary version*, 78(110).

[58] Hasabelnaby, M. (2016). Decentralized runtime verification of ltl specifications in distributed systems. Master's thesis, University of Waterloo.

[59] Havelund, K. and Rosu, G. (2001). Monitoring programs using rewriting. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 135–143. IEEE.

[60] He, F. and Zhao, S. (2008). Research on synchronous control of nodes in distributed network system. In *2008 IEEE International Conference on Automation and Logistics*, pages 2999–3004. IEEE.

[61] Hendry-Brogan, M. (2019). Global unmanned aerial vehicle (uav) market report. Technical report, Technical report, May.

[62] Hsu, C.-H. and Kremer, U. (2003). The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 38–48.

[63] Jiang, Y., Song, H., Wang, R., Gu, M., Sun, J., and Sha, L. (2016). Data-centered runtime verification of wireless medical cyber-physical system. *IEEE transactions on industrial informatics*, 13(4):1900–1909.

[64] Kazemlou, S. and Bonakdarpour, B. (2018). Crash-resilient decentralized synchronous runtime verification. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 207–212. IEEE.

[65] Ketkar, N. (2017). Introduction to keras. In *Deep learning with Python*, pages 97–111.

Springer.

[66] Kopetz, H., Grünsteidl, G., and Reisinger, J. (1991). Fault-tolerant membership service in a synchronous distributed real-time system. In *Dependable Computing for Critical Applications*, pages 411–429. Springer.

[67] Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299.

[68] Kshemkalyani, A. and Singhal, M. (2011). *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press.

[69] Kuhn, M., Johnson, K., et al. (2013). *Applied predictive modeling*, volume 26. Springer.

[70] Kuila, P. and Jana, P. K. (2014). A novel differential evolution based clustering algorithm for wireless sensor networks. *Applied soft computing*, 25:414–425.

[71] Kulkarni, S. S., Demirbas, M., Madappa, D., Avva, B., and Leone, M. (2014). Logical physical clocks. In *International Conference on Principles of Distributed Systems*, pages 17–32. Springer.

[72] Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40.

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

[74] Lee, E. A. (2008). Cyber physical systems: Design challenges. In *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*, pages 363–369. IEEE.

[75] Lesi, V., Jovanov, I., and Pajic, M. (2017). Security-aware scheduling of embedded control tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–21.

[76] Lim, K. K., Park, J., and Shon, J. G. (2019). Differential data processing technique to improve the performance of wireless sensor networks. *The Journal of Supercomputing*, 75(8):4489–4504.

[77] Liu, L., Kong, W., Ando, T., Yatsu, H., and Fukuda, A. (2013). A survey of acceleration techniques for smt-based bounded model checking. In *2013 international conference on computer sciences and applications*, pages 554–559. IEEE.

[78] Lorch, J. R. and Smith, A. J. (2001). Improving dynamic voltage scaling algorithms with pace. *ACM SIGMETRICS Performance Evaluation Review*, 29(1):50–61.

[79] Maler, O. and Nickovic, D. (2004). Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer.

[80] Manna, Z. and Pnueli, A. (2012). *Temporal verification of reactive systems: safety*. Springer Science & Business Media.

[81] Mattern, F. et al. (1988). *Virtual time and global states of distributed systems*. Univ., Department of Computer Science.

[82] Medhat, R., Bonakdarpour, B., and Fischmeister, S. (2018). Energy-efficient multiple producer-consumer. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):560–574.

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

[84] Medhat, R., Funk, S., and Rountree, B. (2017). Scalable performance bounding under multiple constrained renewable resources. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, pages 1–8.

[85] Mehlitz, P., Giannakopoulou, D., and Shafiei, N. (2019). Analyzing airspace data with race. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pages 1–10. IEEE.

[86] Mehmood, I., Ullah, A., Muhammad, K., Deng, D.-J., Meng, W., Al-Turjman, F., Sajjad, M., and de Albuquerque, V. H. C. (2019). Efficient image recognition and retrieval on iot-assisted energy-constrained platforms from big data repositories. *IEEE Internet of Things Journal*, 6(6):9246–9255.

[87] Metropolis, N. and Ulam, S. (1949). The monte carlo method. *Journal of the American statistical association*, 44(247):335–341.

[88] Mills, D., Martin, J., Burbank, J., and Kasch, W. (2010). Network time protocol version 4: Protocol and algorithms specification. RFC 5905, RFC Editor.

[89] Mitsch, S. and Platzer, A. (2016). Modelplex: Verified runtime validation of verified cyber-physical system models. *Formal Methods in System Design*, 49(1):33–74.

[90] Mittal, N. and Garg, V. K. (2001). On detecting global predicates in distributed computations. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 3–10. IEEE.

[91] Mittal, N. and Garg, V. K. (2005). Techniques and applications of computation slicing.

Distributed Computing, 17(3):251–277.

[92] Mittal, V., Gupta, S., and Choudhury, T. (2018). Comparative analysis of authentication and access control protocols against malicious attacks in wireless sensor networks. In *Smart computing and informatics*, pages 255–262. Springer.

[93] Mogull, R. and Securosis, L. (2007). Understanding and selecting a data loss prevention solution. *Technicalreport, SANS Institute*, 27.

[94] Momtaz, A., Abbas, H., and Bonakdarpour, B. (2023). Monitoring signal temporal logic in distributed cyber-physical systems. In *Proceedings of the ACM/IEEE 14th International Conference on Cyber-Physical Systems (with CPS-IoT Week 2023)*, ICCPS '23, page 154–165, New York, NY, USA. Association for Computing Machinery.

[95] Momtaz, A., Basnet, N., Abbas, H., and Bonakdarpour, B. (2021). Predicate monitoring in distributed cyber-physical systems. In *International Conference on Runtime Verification*, pages 3–22. Springer.

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

[97] Moura, L. d. and Bjørner, N. (2008). Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

[98] National Science Foundations (2014). Revolutionizing how we keep track of time in cyber-physical systems. *https://nsf.gov/news/news_ summ.jsp?cntn$_i$d=131691*.

[99] Ogale, V. A. and Garg, V. K. (2007). Detecting temporal logic predicates on distributed computations. In *International Symposium on Distributed Computing*, pages 420–434. Springer.

[100] Pant, Y. V., Abbas, H., and Mangharam, R. (2017). Smooth operator: Control using the smooth robustness of temporal logic. In *2017 IEEE Conference on Control Technology and Applications (CCTA)*, pages 1235–1240. IEEE.

[101] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830.

[102] Pereira, J. C., Machado, N., and Sousa Pinto, J. (2020). Testing for race conditions in distributed systems via smt solving. In *International Conference on Tests and Proofs*, pages 122–140. Springer.

[103] Petri, C. A. and Reisig, W. (2008). Petri net. *Scholarpedia*, 3(4):6477.

[104] Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. ieee.

[105] Quesel, J.-D. (2013). *Similarity, logic, and games: bridging modeling layers of hybrid systems*. PhD thesis, Univ., Fak. II, Department für Informatik.

[106] Rountree, B., Lowenthal, D. K., Funk, S., Freeh, V. W., De Supinski, B. R., and Schulz, M. (2007). Bounding energy consumption in large-scale mpi programs. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–9. IEEE.

[107] Sánchez, C. (2018). Online and offline stream runtime verification of synchronous systems. In *International Conference on Runtime Verification*, pages 138–163. Springer.

[108] Sen, A. and Garg, V. K. (2004). Detecting temporal logic predicates in distributed programs using computation slicing. In *Principles of Distributed Systems: 7th International Conference, OPODIS 2003, La Martinique, French West Indies, December 10-13, 2003, Revised Selected Papers 7*, pages 171–183. Springer.

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

[110] Shrestha, A., Xing, L., and Liu, H. (2007). Modeling and evaluating the reliability of wireless sensor networks. In *2007 Annual Reliability and Maintainability Symposium*, pages 186–191. IEEE.

[111] Silva, I., Guedes, L. A., Portugal, P., and Vasques, F. (2012). Reliability and availability evaluation of wireless sensor networks for industrial applications. *Sensors*, 12(1):806–838.

[112] Sistla, A. P., Žefran, M., and Feng, Y. (2011). Runtime monitoring of stochastic cyber-physical systems with hybrid state. In *International Conference on Runtime Verification*, pages 276–293. Springer.

[113] Sodhro, A. H., Chen, L., Sekhari, A., Ouzrout, Y., and Wu, W. (2018). Energy efficiency comparison between data rate control and transmission power control algorithms for wireless body sensor networks. *International Journal of Distributed Sensor Networks*, 14(1):1550147717750030.

[114] Stoller, S. D. (1997). Detecting global predicates in distributed systems with clocks. In *International Workshop on Distributed Algorithms*, pages 185–199. Springer.

[115] Stoller, S. D. and Schneider, F. B. (1995). Verifying programs that use causally-ordered

message-passing. *Science of computer programming*, 24(2):105–128.

[116] Tekken Valapil, V., Yingchareonthawornchai, S., Kulkarni, S., Torng, E., and Demirbas, M. (2017). Monitoring partially synchronous distributed systems using smt solvers. In *International Conference on Runtime Verification*, pages 277–293. Springer.

[117] USNRC (2021a). Emergency core cooling systems. *https://www.nrc.gov/docs/ML1122/ML11223A220.pdf*.

[118] USNRC (2021b). Pressurized water reactor systems. *https://www.nrc.gov/reading-rm/basic-ref/students/for-educators/04.pdf*.

[119] Valapil, V. T., Kulkarni, S., Torng, E., and Appleton, G. (2020). Efficient two-layered monitor for partially synchronous distributed systems (technical report). *arXiv preprint arXiv:2007.13030*.

[120] Vu, A.-D., Medhat, R., and Bonakdarpour, B. (2019). Managing the security-energy tradeoff in distributed cyber-physical systems. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 118–128.

[121] Widder, J., Lann, G. L., and Schmid, U. (2005). Failure detection with booting in partially synchronous systems. In *European Dependable Computing Conference*, pages 20–37. Springer.

[122] Wolf, W. (2009). Cyber-physical systems. *Computer*, 42(03):88–89.

[123] Wong, T.-T. and Yeh, P.-Y. (2019). Reliable accuracy estimates from k-fold cross validation. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1586–1594.

[124] Wu, Q., Juang, P., Martonosi, M., Peh, L.-S., and Clark, D. W. (2005). Formal control techniques for power-performance management. *IEEE micro*, 25(5):52–62.

[125] Xia, F., Ma, L., Dong, J., and Sun, Y. (2008). Network qos management in cyber-physical systems. In *2008 International Conference on Embedded Software and Systems Symposia*, pages 302–307. IEEE.

[126] Xie, F., Martonosi, M., and Malik, S. (2003). Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 49–62.

[127] Xie, F., Martonosi, M., and Malik, S. (2004). Intraprogram dynamic voltage scaling: Bounding opportunities with analytic modeling. *ACM Transactions on Architecture and Code Optimization (TACO)*, 1(3):323–367.

[128] Xie, F., Martonosi, M., and Malik, S. (2005). Bounds on power savings using runtime

dynamic voltage scaling: an exact algorithm and a linear-time heuristic approximation. In *Proceedings of the 2005 international symposium on Low power electronics and design*, pages 287–292.

[129] Xu, S. and Chen, L. (2008). A novel approach for determining the optimal number of hidden layer neurons for fnn's and its application in data mining. *5th International Conference on Information Technology and Applications*.

[130] Yingchareonthawornchai, S., Nguyen, D. N., Valapil, V. T., Kulkarni, S. S., and Demirbas, M. (2016). Precision, recall, and sensitivity of monitoring partially synchronous distributed systems. In *International Conference on Runtime Verification*, pages 420–435. Springer.

[131] Zhang, J., Tu, H., Ren, Y., Wan, J., Zhou, L., Li, M., and Wang, J. (2018). An adaptive synchronous parallel strategy for distributed machine learning. *IEEE Access*, 6:19222–19230.

[132] Zhang, T., Gebhard, P., and Sokolsky, O. (2016). Smedl: combining synchronous and asynchronous monitoring. In *International Conference on Runtime Verification*, pages 482–490. Springer.

[133] Zheng, X., Julien, C., Podorozhny, R., Cassez, F., and Rakotoarivelo, T. (2016). Efficient and scalable runtime monitoring for cyber–physical system. *IEEE Systems Journal*, 12(2):1667–1678.

[134] Zhou, Y., Zhang, Y., and Fang, Y. (2007). Access control in wireless sensor networks. *Ad Hoc Networks*, 5(1):3–13.