Title
Modular and Safe Event-Driven Programming

Author
Desai, Ankush Pankaj

Publication Date
2019

Peer reviewed|Thesis/dissertation

Modular and Safe Event-Driven Programming

by

Ankush Pankaj Desai

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Dr. Shaz Qadeer
Professor Koushik Sen
Professor Raja Sengupta
Professor Claire Tomlin

Fall 2019

Modular and Safe Event-Driven Programming

Abstract

Modular and Safe Event-Driven Programming

by

Ankush Pankaj Desai

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

Asynchronous event-driven systems are ubiquitous across domains such as device drivers, distributed systems, and robotics. These systems are notoriously hard to get right as the programmer needs to reason about numerous control paths resulting from the complex interleaving of events (or messages) and failures. Unsurprisingly, it is easy to introduce subtle errors while attempting to fill in gaps between high-level system specifications and their concrete implementations. This dissertation proposes new methods for *programming safe event-driven asynchronous systems*.

In the first part of the thesis, we present MODP, a modular programming framework for compositional programming and testing of event-driven asynchronous systems. The MODP module system supports a novel theory of compositional refinement for assume-guarantee reasoning of dynamic event-driven asynchronous systems. We build a complex distributed systems software stack using MODP. Our results demonstrate that compositional reasoning can help scale model-checking (both explicit and symbolic) to large distributed systems. MODP is transforming the way asynchronous software is built at Microsoft and Amazon Web Services (AWS). Microsoft uses MODP for implementing safe device drivers and other software in the Windows kernel. AWS uses MODP for compositional model checking of complex distributed systems. While MODP simplifies analysis of such systems, the state space of industrial-scale systems remains extremely large.

In the second part of this thesis, we present scalable verification and systematic testing approaches to further mitigate this state-space explosion problem. First, we introduce the concept of a *delaying explorer* to perform prioritized exploration of the behaviors of an asynchronous reactive program. A delaying explorer stratifies the search space using a custom strategy (tailored towards finding bugs faster), and a delay operation that allows deviation from that strategy. We show that prioritized search with a delaying explorer performs significantly better than existing approaches for finding bugs in asynchronous programs.

Next, we consider the challenge of verifying time-synchronized systems; these are almost-synchronous systems as they are neither completely asynchronous nor synchronous. We introduce *approximate synchrony*, a sound and tunable abstraction for verification of almost-

synchronous systems. We show how approximate synchrony can be used for verification of both time-synchronization protocols and applications running on top of them. Moreover, we show how approximate synchrony also provides a useful strategy to guide state-space exploration during model-checking. Using approximate synchrony and implementing it as a delaying explorer, we were able to verify the correctness of the IEEE 1588 distributed time-synchronization protocol and, in the process, uncovered a bug in the protocol that was well appreciated by the standards committee.

In the final part of this thesis, we consider the challenge of programming a special class of event-driven asynchronous systems – safe autonomous robotics systems. Our approach towards achieving *assured autonomy* for robotics systems consists of two parts: (1) a *high-level programming language* for implementing and validating the reactive robotics software stack; and (2) an integrated *runtime assurance* system to ensure that the assumptions used during design-time validation of the high-level software hold at runtime. Combining high-level programming language and model-checking with runtime assurance helps us bridge the gap between design-time software validation that makes assumptions about the untrusted components (e.g., low-level controllers), and the physical world, and the actual execution of the software on a real robotic platform in the physical world. We implemented our approach as DRONA, a programming framework for building safe robotics systems. We used DRONA for building a distributed mobile robotics system and deployed it on real drone platforms. Our results demonstrate that DRONA (with the runtime-assurance capabilities) enables programmers to build an autonomous robotics software stack with *formal safety* guarantees.

To summarize, this thesis contributes new theory and tools to the areas of programming languages, verification, systematic testing, and runtime assurance for *programming safe asynchronous event-driven* across the domains of fault-tolerant distributed systems and safe autonomous robotics systems.

To my parents, Lata and Pankaj,
and my wife, Priyanka.

Guru Brahma Guru Vishnu
Guru Devo Maheshwara
Guru Sakshat Param Brahma
Tasmai Shri Gurave Namah

— Adi Shankaracharya
Guru strotam

ACKNOWLEDGMENTS

The above prayer best describes the dedication of this thesis to all my teachers, mentors, and family. It means: "*Guru is verily the supreme reality. Sublime prostrations to all my Gurus*". None of what I have achieved would have been possible without their constant guidance and support.

I want to begin by thanking my amazing advisors Sanjit Seshia and Shaz Qadeer.

I will always be indebted to Sanjit for the continuous support and guidance received over the past six years. He has taught me, trusted me, and has always shown an unwavering enthusiasm to all my nebulous ideas. His patience and positive attitude towards each rejection (which I had quite a few) have always amazed me. I hope that a small percentage of his brilliance has rubbed off on me through our collaboration over the last six years.

I am incredibly fortunate to have had the opportunity to work with Shaz for more than eight years now. His unceasing support and thoughtful advice over the years has bolstered the quality of my research. I idolize Shaz for his industriousness and the passion for doing impactful, practical research. He has not only been a great advisor to me but also a friend, a collaborator, a hacker on P, and a mentor for life!

I will always be grateful to Sriram Rajamani for taking me under his wings at Microsoft Research. It was under his and Shaz's guidance that I started working on P. Sriram introduced me to research, gave me the essential training, and encouraged me to pursue a doctorate degree.

I sincerely thank my thesis and quals committee: Claire Tomlin, Koushik Sen, and Raja Sengupta for their valuable feedback.

Special thanks to Amar Phanishayee for getting me interested in distributed systems and for the wonderful summer internship at Microsoft Research. I was fortunate to do a summer internship at SRI with Natarajan Shankar and Ashish Tiwari, their knowledge and humility will always be a source of inspiration for me.

I thank the people of Learn and Verify group for being a part of my journey and making the time spent in 545S memorable: Pramod Subramanyan, Daniel Fremont, Rohit Sinha, Eric Kim, Dorsa Sadigh, Markus Rabe, Tommaso Dreossi, Indranil Saha,

Ben Caulfield, Marcell Vazquez-Chanlatte, Shromona Ghosh, Hadi Ravanbakhsh, Yi-Chin Wu, Nishant Totla, Garvit Juniwal, Edward Kim, Wenchao Li, Alexander Donze, Daniel Bundala, and Jonathan Kotker. A big shout out to Daniel Fremont for his help reviewing many of my papers before submission.

Thanks to the staff within EECS and BIO, in particular, Shirley Salanio, and Tatiana Djordjevic.

I want to thank my family for giving me the strength to complete my graduate studies. I am, as ever, indebted to my parents, Lata and Pankaj, for all the love, support, and sacrifices they have made to give us the best. My sister, Nikita, I cannot thank her enough; she has been like my second mother in the US. My brother-in-law, Abhijit, I thank him for always being there for me and encouraging me to follow my dreams. I am so blessed to have two Angels, Mihika and Anay, spending time with them over the weekends energized me for the Ph.D. grind.

The final thanks go to my wonderful wife, Priyanka, for being that one person who held everything together and helped me keep my sanity amid chaos. Her belief in me; and her positivity has led me to look at the world differently. Words cannot express my thanks to her. Looking forward to our life together!

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1

INTRODUCTION

There is a race between the increasing complexity of the systems we build and our ability to develop intellectual tools for understanding their complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable for all but a relatively small set of common tasks. Given how hard thinking is, if those intellectual tools are to succeed, they will have to substitute calculation for thought.

— *Leslie Lamport*

Today applications across domains are implemented as event-driven asynchronous systems[1] — from the device-drivers in an operating system, to the fault-tolerant distributed systems running the cloud services, to the control software in a modern car or an autonomous robot. Programmers inevitably choose to develop these systems as event-driven asynchronous systems to exploit concurrency for better performance, responsiveness, fault tolerance, and autonomy. *Asynchrony* has, therefore, become a fundamental attribute of most main-stream software systems.

Unfortunately, asynchrony is at odds with correctness, programming asynchronous event-driven systems is notoriously hard as one needs to reason about numerous control paths resulting from the myriad interleaving of messages. It is easy to introduce subtle errors while improvising to fill in gaps between the high-level protocol descriptions and their concrete implementations. In practice, it is extremely difficult to test asynchronous systems; unlike sequential programs whose execution can be

1 Event-driven asynchronous systems are systems that are built on top of the actor [5] or communicating state-machines [93] model of computation where processes execute concurrently and communicate with each other by sending message asynchronously.

controlled via the input, controlling the execution of an asynchronous program requires fine-grained control over the timing of the execution of event handlers (or delivery of messages). In the absence of such control, most control paths remain untested, and serious bugs lie dormant for months or even years after deployment. Finally, bugs that occur during testing or after deployment tend to be Heisenbugs; they are notoriously difficult to reproduce because their manifestation requires timing requirements that might not hold from one execution to another. These problems are well-known and have been highlighted by creators of large-scale industrial systems [37]. Despite decades of research in verification and testing techniques oriented towards concurrent, asynchronous event-driven systems, the practice of programming such systems "in-the-wild" has not changed. However, this problem can no longer be overlooked, especially as most of the real-world systems increasingly have correctness requirements such as consistency or fault-tolerance guarantees for distributed services and safety guarantees for autonomous robots. In order to address these challenges:

> *This dissertation presents a new language, supported by novel testing, verification, and run-time assurance techniques, for programming* ***safe*** *asynchronous event-driven systems.*

We address these challenges by dividing it into three core problems and take the following approaches for solving them:

- First, to address the problem of programming complex event-driven system, we design a domain-specific programming language for implementing and specifying asynchronous event-driven systems. Thus, enabling the programmers to capture the protocol logic at a higher level of abstraction without worrying about the low-level implementation details.

- Second, to check the correctness of systems built using our language, we propose new verification and systematic testing techniques that enable scalable analysis of these systems.

- Finally, to demonstrate the efficacy of the proposed programming language and the accompanying formal analysis techniques for building real-world systems, we build applications across the domains of fault-tolerant distributed systems and safe autonomous robotics systems. When building these applications, we also solved several domain-specific challenges. For example, we had to extend the framework with the principles of runtime assurance to ensure the safety of robotics systems in the presence of machine learning components.

To summarize, this thesis combines ideas from and contributes new theory and tools to the areas of programming languages, verification, systematic testing, and runtime assurance for solving the challenging problem of *safe asynchronous event-driven programming*. The contributions are described in further detail in Section 1.2.

We strongly believe that formal methods can succeed in practice and become a part of the software development cycle if the process of modeling, specification, implementation, and verification (or systematic testing) is *unified* into a single programming framework. Hence, the solutions proposed in this thesis are all built on top of (and integrated into) the unified programming framework, P [53, 150].

In the rest of this chapter, we first provide a brief background on the P programming framework and then present the primary contributions of this thesis. We conclude by providing the thesis outline and list the previously published work included in this thesis.

1.1 BACKGROUND: THE P PROGRAMMING FRAMEWORK

Event-driven asynchronous systems are ubiquitous and developers in the industry use frameworks based on popular asynchronous programming paradigms of actors [5, 7, 13, 32, 161] and communicating state machines [53, 93, 112] for building these systems of significant commercial interest [7, 24, 47, 53]. Event-driven asynchronous programs typically have layers of design, where the higher layers reason with how the various components (or machines) interact and the protocol they follow, and where lower layers manage more data-intensive computations, controlling local devices, etc. However, the programs often get written in traditional languages that offer no mechanisms to capture these abstractions, and hence over time leads to code where the individual layers are no longer discernible. High-level protocols, though often first designed on paper using clean graphical state-machine abstractions, eventually get lost in code, and hence verification tools for such programs face the daunting task of extracting these models from the programs.

The natural solution to the above problem is to build a programming language for asynchronous event-driven programs that preserves the protocol abstractions in code. Apart from the difficulty in designing such a language, this task is plagued by the reluctance of programmers to adopt a new language of programming and the discipline that it brings. However, this precise solution was pioneered by the P [52] programming framework, where, during the development of Windows 8, the team building the USB driver stack used P for modeling, implementing, and model-checking of the USB 3.0 device drivers. Programs written in P capture the high-level protocol using a collection of interacting state machines that communicate with each other by exchanging messages. P supports the actor [5] model of computation with the additional syntactic sugar of actors being replaced by state-machines as it is easier to capture the protocol design as state-machines

Figure 1.1 presents the architectural overview of the P framework, consisting of three main building blocks — (1) the P programming language for implementing and specifying the event-driven programs, (2) the P-Explorer (model-checker) for verification and systematic testing of P programs, and (3) the P-Runtime that efficiently

executes the generated code from the high-level P programs. Not only can a P program be compiled into executable code, but it can also be validated using model-checking. This aspect of the P language, of being a unified framework, effectively blurs the distinction between modeling, writing specification, and programming; and hence, makes formal methods more accessible to programmers.

1.2 PRIMARY CONTRIBUTIONS

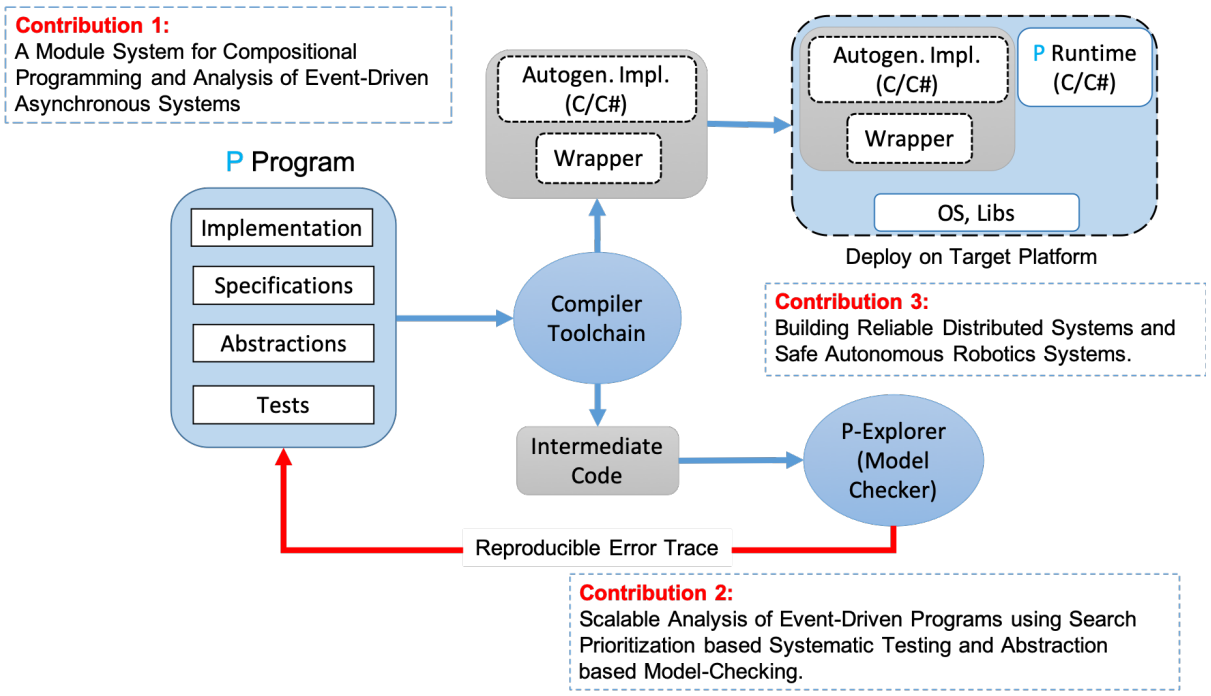


Figure 1.1: The Primary Contributions of this Thesis

The P programming framework laid the foundation for this thesis and was the first step towards enabling safe programming of asynchronous event-driven systems. However, when building more complex applications like fault-tolerant distributed system and safe robotics systems using P, we had to overcome several challenges and propose new methods which led to the contributions of this thesis. Figure 1.1 provides an overview of the primary contributions of this thesis: (1) *a module system for compositional programming and testing* of event-driven asynchronous systems (Section 1.2.1), (2) new approaches for *scalable analysis* (verification and systematic testing) of event-driven asynchronous systems (Section 1.2.2), and (3) applying this unified framework for *building reliable distributed systems and safe robotics systems* (Section 1.2.2).

1.2.1 *A Language for Modular Programming of Event-Driven Systems*

P is based on the actor model of computation similar to other popular languages frameworks used for implementing high-performance asynchronous distributed systems [7, 13, 32, 161]. However, these languages do not support compositional programming and testing of distributed systems. A real-world system is rarely implemented as a standalone monolithic system. Instead, it is composed of multiple independent interacting components that together ensure the desired system-level specification (e.g., our case study in Figure 3.1). One can scale systematic testing to large, industrial-scale implementations by decomposing the system-level testing problem into a collection of simpler component-level testing problems. Moreover, the results of component-level testing can be lifted to the whole system level by leveraging the theory of assume-guarantee (AG) reasoning [3, 9, 132].

We propose a module system based on a novel theory of compositional trace refinement for dynamic event-driven systems consisting of asynchronously-communicating state machines, where state machines can be dynamically created, and communication topology of the existing state machines can change at runtime. We present MODP (Modular P), an extension of the P language that implements the module system for compositional programming and testing (based on assume-guarantee reasoning) of *asynchronous event-driven systems*. To the best of our knowledge, MODP is the first system that supports assume-guarantee reasoning in a practical programming language with these dynamic features for implementing asynchronous event-driven systems.

Research Impact

Positive experience with P and MODP in the Windows kernel and Microsoft Azure led to the development of P# [47], a framework that provides state machines and systematic testing via an extension to C#. The programming model of P# (e.g., state machines and monitors for writing specifications) is inspired from the MODP language described in this thesis. P# is used by several teams in Azure to design, implement and automatically test production distributed systems and services (`https://github.com/p-org/PSharp`). More recently, MODP is being used inside Amazon Web Services (AWS) for compositional model checking of distributed protocols.

1.2.2 *Approaches for Scalable Analysis of Event-Driven Systems*

The MODP module system enables compositional verification (or systematic testing) of P programs. Analysis (model-checking) of the decomposed system does simplify the overall monolithic problem but still suffers from scalability issues when applied to industrial-scale systems. Each component in a real-world system software stack

implements a complex protocol, and hence, even testing a component in isolation can lead to the state-space explosion problem. To further scale the analysis and mitigate the state-space explosion problem, we complement compositional testing with two techniques: **(1)** *Search prioritization-based falsification* (or bug-finding): Extending the model-checker with guided or directed search geared towards falsification of the property to be verified; and **(2)** *Abstraction-based verification*: Using a sound abstraction (*superset*) of the program behaviors to simplify the overall verification problem.

We introduce the concept of a *delaying explorer* to perform prioritized exploration of the behaviors of an asynchronous reactive program. A delaying explorer stratifies the search space using a custom strategy (tailored towards finding bugs faster), and a delay operation that allows deviation from that strategy. We show that prioritized search with a delaying explorer performs significantly better than existing approaches for finding bugs in asynchronous programs. Our results also demonstrated that there is no unique winning strategy for finding bugs in concurrent systems.

Research Impact

The P# [47] framework used by Microsoft Azure to build several distributed services implements a *portfolio* approach for systematic testing, where they run a collection of different search prioritization strategies in parallel, each targeting a different part of the search space.
The portfolio approach beats other state-of-the-art search heuristics for finding bugs in asynchronous programs and is inspired from the results and observations of our work on delaying explorers.

The next challenge problem we considered was verification of the IEEE 1588 [68] distributed time synchronization protocol using P. For time-synchronized systems, at any time point, clocks of different nodes can have different values, but time synchronization ensures that those values are within a specified offset of each other, i.e., they are *almost synchronized*, neither completely asynchronous or synchronous. We present an abstraction-based model-checking approach for verification of almost-synchronous event-driven systems. We introduce *approximate synchrony*, a sound and tunable abstraction for verification of almost-synchronous systems. We show how approximate synchrony can be used for verification of both time-synchronization protocols and applications running on top of them. Moreover, we show how approximate-synchrony also provides a useful strategy to guide state-space exploration during model-checking.

Research Impact

Using approximate synchrony and implementing it as a delaying explorer, we were able to verify the correctness of IEEE1588 protocol and also in the process uncovered a bug in the protocol that was well appreciated by the standards committee [30].

1.2.3 *Building Reliable Distributed Systems and Robotics Applications*

To demonstrate the efficacy of the modular P (ModP) language and its backend scalable analysis framework described above, we built applications across two domains: fault-tolerant distributed systems and autonomous robotics systems.

Reliable distributed systems. We used ModP to build a fault-tolerant distributed services software stack consisting of 7 complex protocols. Our results demonstrate that the theory of compositional refinement can be used in practice to build systems compositionally and scale systematic testing to large systems. We also compared the performance of services built using ModP with its open-source equivalent to show that distributed systems can be implemented in a principled way using formal analysis without sacrificing performance.

Assured Autonomy. The recent drive towards achieving greater autonomy and intelligence in robotics has led to increasing levels of complexity in the robotics software stack. This trend has resulted in a widening gap between the complexity of systems being deployed and those that can be certified for safety and correctness of operation. *Assured autonomy* requires a robot to make correct and timely decisions, where the robotics software stack is implemented as a concurrent, reactive, event-driven system that may also use advanced machine learning-based components.

Our approach towards achieving *assured autonomy* for robotics systems consists of two parts: **(1)** a *high-level programming language* based on P for implementing and validating the reactive robotics software stack; and **(2)** an integrated *runtime assurance* system to ensure that the assumptions used during design-time validation of the high-level software hold at runtime. Combining high-level programming language and model-checking with runtime assurance helps us bridge the gap between design-time software validation that makes assumptions about the untrusted components (e. g., low-level controllers), and the physical world, and the actual execution of the software on a real robotic platform in the physical world. We implemented the above approach in Drona, a framework for building safe robotics systems. We advocate the use of principles of runtime assurance to ensure the safety of the robotics systems in the presence of untrusted components like third-party libraries or machine learning-based components. We present the runtime assurance framework integrated into Drona and

demonstrate how it enables guaranteeing the safety of the robotics system, including when untrusted components have bugs or deviate from the desired behavior.

Research Impact

We used DRONA for building distributed robotics systems and deployed them on autonomous drone platforms for several DARPA demos (videos available on https://drona-org.github.io/Drona/). To the best of our knowledge, we are the first to integrate the principles of runtime assurance into a practical programming language and use it to build real-world robotics software stack that can be deployed on real drones. We conducted rigorous software in the loop simulations for **104** hours. We found that there were 109 disengagements; these were cases where one of the safety systems took control and avoided a potential crash. These results demonstrated that runtime assurance can help guarantee safety of the system in the presence of machine-learning components that are hard to verify.

1.3 THESIS OUTLINE

The remainder of this dissertation proceeds in three parts, describing each of the contributions as follows:

1. Part i presents the MODP module system for compositional programming and testing of event-driven asynchronous systems. Chapter 2 presents the novel theory of compositional refinement supported by the MODP module system that enables assume-guarantee reasoning of P programs. Chapter 3 demonstrates the efficacy of the theory of compositional refinement in practice by building real-world fault-tolerance distributed services using MODP.

2. Part ii discusses the systematic testing and verification approaches for scalable analysis of complex systems implemented using P. Chapter 4 presents a programmable search-prioritization technique for systematic testing of asynchronous reactive programs. Chapter 5 introduces approximate synchrony, an abstraction for scalable verification of almost-synchronous systems.

3. Part iii presents the application of novel programming languages and runtime assurance techniques for building safe autonomous robotics systems. Chapter 6 describes our robotics case study of autonomous drones and highlights the challenges in guaranteeing assured autonomy. Chapter 7 presents the DRONA framework that extends P to enable programming reliable distributed mobile robotics software stack. Chapter 8 presents the SOTER framework that extends

DRONA with runtime assurance capabilities for guaranteeing assured autonomy in the presence of untrusted components.

Finally, Chapter 9 concludes and provides future work for this dissertation.

1.4 PREVIOUSLY PUBLISHED WORK AND FORMAL ACKNOWLEDGMENT

This thesis includes and revises content from several of my previously published papers. I gratefully acknowledge and thank my advisors, Sanjit Seshia and Shaz Qadeer, who have played an important role in shaping the contributions in all these papers. Chapter 2 and Chapter 3 revises our paper on MODP [57]. I thank Amar Phanishayee for introducing us to the challenges in building fault-tolerant distributed systems and explaining the complex protocols used as case-studies in the MODP paper. Chapter 4 revises material from [50]. Chapter 5 revises our paper on approximate synchrony [55]. I sincerely thank John Eidson for introducing us to the problem of verifying the IEEE 1588 protocol that led to the development of approximate synchrony and the follow-up paper [30] that describes the bug we found in the IEEE 1588 protocol. Chapter 6 summarizes material from [51] and [49]. Chapter 7 includes content from our paper on DRONA [56], which is joint work with Indranil Saha and Cambridge Yang. Chapter 8 revises our publication on SOTER [58]. I thank Natarajan Shankar and Ashish Tiwari for proposing the idea of exploring Simplex assurance for building safe robotics systems. Shromona Ghosh helped with the experiments in SOTER [58] involving the FastTrack framework for computing the safe-controllers. Chapter 8 also includes some results from our paper on runtime verification for safe robotics systems [48], which is joint work with Tommaso Dreossi.

Part I

MODULAR PROGRAMMING OF EVENT-DRIVEN SYSTEMS

2

A MODULE SYSTEM FOR COMPOSITIONAL REASONING OF EVENT-DRIVEN SYSTEMS

To keep large programs well structured and modular, you either need superhuman will power, or proper language support.

— *Greg Nelson*

Existing validation methods for asynchronous even-driven systems fall into two categories: *proof-based verification* and *systematic testing*. Researchers have used theorem provers to construct correctness proofs of both single-node systems [38, 95, 113, 199] and distributed systems [96, 152, 200]. To prove a safety property on a distributed system, one typically needs to formulate an inductive invariant. Moreover, the inductive invariant often uses quantifiers, leading to unpredictable verification time and requiring significant manual assistance. While invariant synthesis techniques show promise, the synthesis of quantified invariants for large-scale real-world systems remains difficult. In contrast to proof-based verification, systematic testing explores behaviors of the system in order to find violations of safety specifications [90, 111, 205]. Systematic testing is attractive to programmers as it is mostly automatic and needs less expert guidance. Unfortunately, even state-of-the-art systematic testing techniques scale poorly with increasing system complexity.

A real-world system is rarely implemented as a standalone monolithic system. Instead, it is composed of multiple independent interacting components that together ensure the desired system-level specification (e.g., our case study in Figure 3.1). One can scale systematic testing to large, industrial-scale implementations by decomposing the system-level testing problem into a collection of simpler component-level testing problems. Moreover, the results of component-level testing can be lifted to the whole system level by leveraging the theory of assume-guarantee (AG) reasoning [3, 9, 132].

In this chapter, we present MODP (Modular P)[1], an extension of the P language for compositional programming and testing (based on AG reasoning) of *asynchronous*

1 MODP stands for Modular P and is available as part of the P programming framework [150].

event-driven systems. MODP occupies a spot between proofs and black-box monolithic testing in terms of the trade-off between validation coverage and programmer effort.

Actors [5, 7, 13, 32, 161] and state machines [53, 93, 112] are popular paradigms for programming asynchronous systems. These programming models support features like dynamic creation of machines (processes), directed messaging using machine references (as opposed to broadcast), and dynamic communication topology as references to machines can flow through the system (essential for modeling non-determinism like failures). These *dynamic* features have an important impact on assume-guarantee (AG) reasoning, which typically relies on having explicit component interfaces – e.g., wires between circuits or shared variables between programs [9, 128]. In dynamic distributed systems, *interfaces between modules can change* as new state machines instances are created, or communication topology changes and this dynamic behavior depends on the context of a module. While some formalisms for AG reasoning [18, 77] support such dynamic features, they do not provide a programming framework for building practical dynamic distributed systems. To the best of our knowledge, MODP is the first system that supports assume-guarantee reasoning in a practical programming language with these dynamic features.

We have implemented MODP on top of P [53] and used it for building reliable distributed systems (Chapter 3) and for programming safe robotics systems (Part iii).. The MODP compiler generates code for compositional testing, which involves both safety and refinement testing of the decomposed system. We empirically demonstrate (in Chapter 3) that MODP's abstraction-based decomposition helps the existing P systematic testing (both explicit and symbolic execution) back-ends to scale to large distributed systems.

In the rest of this chapter, we first provide an overview of the MODP framework (Section 2.1), and then present our novel theory of compositional refinement and a module system for the assume-guarantee reasoning of dynamic distributed systems (Section 2.2.1-Section 2.4.1). We conclude the chapter with the related work (Section 2.5).

2.1 MOTIVATION AND OVERVIEW

We illustrate the MODP framework for compositionally implementing, specifying, and testing distributed systems by developing a simple client-server application.

2.1.1 *Basic Programming Constructs in* MODP

A MODP program comprises P state machines communicating asynchronously with each other using events accompanied by typed data values. Each machine has an input

buffer, event handlers, and a local store. The machines run concurrently, receiving and sending events, creating new machines, and updating the local store.

We introduce the key constructs of MODP through a simple client-server application implemented as a collection of MODP state machines. In this example, the client sends a request to the server and waits for a response; on receiving a response from the server, it computes the next request to send and repeats this in a loop. The server waits for a request from the client; on receiving a request, it interacts with a helper protocol to compute the response for the client.

Events. An event declaration has a name and a payload type associated with it. Listing 2.1 (line 2) declares an event `eRequest` that must be accompanied by a tuple of type `RequestType`. Listing 2.1 (line 6) declares the named tuple type `RequestType`. MODP supports primitive types like int, bool, float, and complex types like tuples, sequences and maps.

Interfaces. Each interface declaration has an interface name and a set of events that the interface can receive. For example, the interface `ClientIT` declared at Listing 2.2 (line 3) is willing to receive only event `eResponse`. Interfaces are like symbolic names for machines. In MODP, unlike in the actor model where an instance of an actor is created using its name, an instance of a machine is created indirectly by performing **new** of an interface and linking the interface to the machine separately. For example, execution of the statement `server = new ServerToClientIT` at Listing 2.1 (line 16) creates a fresh instance of machine `ServerImpl` and stores a unique reference to the new machine instance in `server`. The link between `ServerToClientIT` and `ServerImpl` is provided separately by the programmer using the `bind` operation.

```
1  /* Events */
2  event eRequest : RequestType;
3  event eResponse: ResponseType;
4  ...
5  /* Types */
6  type RequestType = (source: ClientIT,reqId:int,val: int);
7  type ResponseType = (resId: int, success: bool);
8
9  machine ClientImpl receives eResponse;
10 sends eRequest; creates ServerToClientIT;
11 {
12   var server : ServerToClientIT;
13   var nextId, nextVal : int;
14   start state Init {
15     entry {
16       server = new ServerToClientIT;
17       goto StartPumpingRequests;
18     }
19   }
```

```
20    state StartPumpingRequests {
21      entry {
22        if(nextId < 5) //send 5 requests
23        {
24          send server, eRequest, (source = this,reqId = nextId,
                val = nextVal);
25          nextId++;
26        }
27      }
28      on eResponse do (payload: ResponseType) {
29        /* compute nextVal */
30        goto StartPumpingRequests;
31      }
32    }
33  }
```

Listing 2.1: Client State Machine in MODP

```
1  /* Interfaces */
2  interface ServerToClientIT receives eRequest;
3  interface ClientIT receives eResponse;
4  interface HelperIT receives eProcessReq;
5
6  machine ServerImpl
7  sends eResponse, eProcessReq;
8  receives eRequest, eReqSuccess, eReqFail;
9  creates HelperIT;
10 {
11   var helper: HelperIT;
12   start state Init {
13     entry {
14       helper = new HelperIT;
15       goto WaitForRequests;
16     }
17   }
18
19   state WaitForRequests {
20     on eRequest do (payload: RequestType) {
21       var client: ClientIT;
22       var result: bool;
23       client = payload.source;
24       /* interacts with the helper machine */
25       send helper, eProcessReq, (payload.reqId,payload.val);
26       ...
```

```
27          /* outcome: result = true or false*/
28          send client, eResponse, (resId = payload.reqId,success
                = result);
29        }
30      }
31  }
32  machine HelperImpl receives eProcessReq;
33  sends eReqSuccess, eReqFail, ..; creates .. ;
34  { /* body */ }
```

Listing 2.2: Server State Machine in MODP

Machines. Listing 2.1 (line 9) declares a machine `ClientImpl` that is willing to receive event `eResponse`, guarantees to send no event other than `eRequest`, and guarantees to create (by executing `new`) no interface other than `ServerToClientIT`. The body of a state machine contains variables and states. Each state can have an entry function and a set of event handlers. The entry function of a state is executed each time the machine transitions into that state. After executing the entry function, the machine tries to dequeue an event from the input buffer or blocks if the buffer is empty. Upon dequeuing an event from the input queue of the machine, the attached handler is executed. Listing 2.1 (line 28) declares an event-handler in the `StartPumpingRequests` state for the `eResponse` event, the `payload` argument stores the payload value associated with the dequeued `eResponse` event. The machine transitions from one state to another on executing the `goto` statement. Executing the statement `send t,e,v` adds event `e` with payload value `v` into the buffer of the target machine `t`. Sends are buffered, non-blocking, and directed. For example, the send statement Listing 2.1 (line 24) sends `eRequest` event to the machine referenced by the `server` identifier. In MODP, the type of a machine-reference variable is the name of an interface (Section 2.2.1.2).

Next, we walk through the implementation of the client (`ClientImpl`) and the server (`ServerImpl`) machines. Let us assume that the interfaces `ServerToClientIT`, `ClientIT`, and `HelperIT` are programmatically linked to the machines `ServerImpl`, `ClientImpl`, and `HelperImpl` respectively (we explain these bindings in Section 2.1.2). A fresh instance of a `ClientImpl` machine starts in the `Init` state and executes its entry function; it first creates the interface `ServerToClientIT` that leads to the creation of an instance of the `ServerImpl` machine, and then transitions to the `StartPumpingRequests` state. In the `StartPumpingRequests` state, it sends a `eRequest` event to the server with a payload value and then blocks for a `eResponse` event. On receiving the `eResponse` event, it computes the next value to be sent to the server and transitions back to the `StartPumpingRequests` state. The `this` keyword is the "self" identifier that references the machine itself. The `ServerImpl` machine starts by creating the `HelperImpl` machine and moves to the `WaitForRequests` state. On receiving a `eResponse` event, the server interacts with the helper machine to compute the `result` that it sends back to the client.

Dynamism. Two key features lead to dynamism in this model of computation, making compositional reasoning challenging: **(1)** *Machines can be created dynamically* during the execution of the program using the `new` operation that returns a reference to the newly-created machine. **(2)** References to machines are first-class values, and the payload in the sent event can contain references to other machines. Hence, the *communication topology can change dynamically* during the execution of the program.

2.1.2 *Compositional Programming using* MODP *Modules*

MODP allows the programmer to decompose a complex system into simple components where each component is a MODP module.

Listing 2.3 presents a modular implementation of the client-server application. A primitive module in MODP is a set of bindings from interfaces to state machines. `ServerModule` is a primitive module consisting of machines `ServerImpl` and `HelperImpl` where the `ServerImpl` machine is bound to the `ServerToClientIT` interface and the `HelperImpl` machine is bound to the `HelperIT` interface. The compiler ensures that the creation of an interface leads to the creation of a machine to which it binds. For example, creation of the `ServerToClientIT` interface (executing `new ServerToClientIT`) by any machine inside the module or by any machine in the environment (i.e., outside `ServerModule`) would lead to the creation of an instance of `ServerImpl`.

The client-server application (Listing 2.3) can be implemented modularly as two separate modules `ClientModule` and `ServerModule`; these modules can be implemented and tested in isolation. Modules in MODP are *open systems*, i.e., machines inside the module may create interfaces that are not bound in the module. Similarly, machines may send events to or receive events from machines that are not in the module. For example, the `ClientImpl` machine in `ClientModule` creates an interface `ServerToClientIT` that is not bound to any machine in `ClientModule`, it sends `eRequest` and receives `eResponse` from machines that are not in `ClientModule`.

```
1 module ClientModule = { ClientIT → ClientImpl };
2 module ServerModule = { ServerToClientIT → ServerImpl,
     HelperIT → HelperImpl };
3
4 //C code generation for the implementation.
5 implementation app: ClientModule ‖ ServerModule;
6
7 module ServerModule′ = { ServerToClientIT → ServerImpl′,
     HelperIT → HelperImpl };
8
9 implementation app′: ClientModule ‖ ServerModule′;
```

Listing 2.3: Modular Client-Server Implementation

Composition in MODP (denoted ||) is supported by type checking. If the composition type checks (Section 2.2.2) then the composition of modules behaves like language intersection over the traces of the modules. The compiler ensures that the joint actions in the composed module (`ClientModule || ServerModule`) are linked appropriately, e.g., the creation of the interface `ServerToClientIT` (Listing 2.1 line 16) in `ClientModule` is linked to `ServerImpl` in `ServerModule` and all the sends of `eRequest` events are enqueued in the corresponding `ServerImpl` machine. Note that the indirection enabled by the use of interfaces is critical for implementing the key feature of *substitution* required for modular programming, i.e., the ability to seamlessly replace one implementation module with another. For example, `ServerModule'` (Listing 2.3 line 7) represents a module where the server protocol is implemented by a different machine `ServerImpl'`. In module `ClientModule || ServerModule'`, the creation of an interface `ServerToClientIT` in the client machine is linked to machine `ServerImpl'`. The *substitution* feature is also critical for compositional reasoning, in which case, an implementation module is replaced by its abstraction. The compiler generates C code for the module in the `implementation` declaration.

2.1.3 *Compositional Testing using* MODP *Modules*

Monolithic testing of large distributed systems is prohibitively expensive due to an explosion of behaviors caused by concurrency and failures. The MODP approach to this problem is to use the principle of assume-guarantee reasoning for decomposing the monolithic system-level testing problem into simpler component-level testing problems; testing each component in isolation using abstractions of the other components.

```
1  machine AbstractServerImpl receives eRequest;
2  sends eResponse;
3  {
4    start state Init {
5      on eRequest do (payload: RequestType) {
6      send payload.source,eResponse,(resId = payload.reqId,
           success = choose());
7      }
8    }
9  }
10 spec ReqIdsAreMonoInc observes eRequest {
11   var prevId : int;
12   start state Init {
13     on eRequest do (payload: RequestType) {
14       assert(payload.reqId == prevId + 1);
15       prevId = payload.reqId;
```

```
16      }
17    }
18  }
19  spec ResIdsAreMonoInc observes eResponse
20  {
21    var prevId : int;
22    start state Init {
23      on eResponse do (payload: ResponseType) {
24        assert(payload.resId == prevId + 1);
25        prevId = payload.resId;
26      }
27    }
28  }
```

Listing 2.4: Abstraction and Specifications in MODP

In MODP, a programmer can specify temporal properties via specification machines (monitors). `spec s observes E1, E2 { .. }` declares a specification machine `s` that observes events `E1` and `E2`. If the programmer chooses to attach s to a module `M`, the code in `M` is instrumented automatically to forward any event-payload pair (e, v) to s if e is in the observes list of s; the handler for event e inside s executes synchronously with the delivery of e. The specification machines observe only the output events of a module. Thus, specification machines introduce a publish-subscribe mechanism for monitoring events to check temporal specifications while testing a MODP module. The module constructor `assert s in P` attaches specification machine `s` to module `P`. In Listing 2.4, `ReqIdsAreMonoInc` and `ResIdsAreMonoInc` are specification machines observing events `eRequest` and `eResponse` to assert the safety property that the `reqId` and `resId` in the payload of these events are always monotonically increasing. Note that `ReqIdsAreMonoInc` is a property of the client machine and `ResIdsAreMonoInc` is a property of the server machine.

In MODP, abstractions used for assume-guarantee reasoning are also implemented as modules. For example, `AbstractServerModule` is an abstraction of the `ServerModule` where the `AbstractServerImpl` machine implements an abstraction of the interaction between `ServerImpl` and `HelperImpl` machine. The `AbstractServerImpl` machine on receiving a request sends back a random response.

```
1 module AbsServerModule = { ServerToClientIT →
     AbstractServerImpl };
2
3 module AbsClientModule = { ClientIT → AbstractClientImpl };
4
5 /* Compositional Safety Checking */
6 //Test: ClientModule.
```

```
7  test test0: (assert ReqIdsAreMonoInc in ClientModule) ||
       AbsServerModule;
8  //Test: ServerModule.
9  test test1: (assert ResIdsAreMonoInc in ServerModule) ||
       AbsClientModule;
10
11 /* Circular Assume-Guarantee */
12 //Check that client abstraction is correct.
13 test test2: ClientModule || AbsServerModule
14             refines
15             AbsClientModule || AbsServerModule;
16
17 //Check that server abstraction is correct.
18 test test3: AbsClientModule || ServerModule
19             refines
20             AbsClientModule || AbsServerModule;
21
22 // Create abstract module using Hide
23 module hideModule = hide X in AbsServerModule;
24
25 test test4: ClientModule || ServerModule
26             refines
27             AbsClientModule || hideModule;
```

Listing 2.5: Test Declarations for Compositional Testing of the Client-Server Application

MODP enables decomposing the monolithic problem of checking: `(assert ReqIdsAreMonoInc, ResIdsAreMonoInc in ClientModule || ServerModule)` into four simple proof obligations. MODP allows the programmer to write each obligation as a test-declaration. The declaration `test tname: P` introduces a safety test obligation that the executions of module `P` do not result in a failure/error. The declaration `test tname: P refines Q` introduces a test obligation that module `P` refines module `Q`. The notion of refinement in MODP is trace-containment based only on externally visible actions, i.e., `P` refines `Q`, if every trace of `P` projected onto the visible actions of `Q` is also a trace of `Q`. MODP automatically discharges these test obligations using systematic testing. Using the theory of compositional safety (Theorem 2.4.3), we decompose the monolithic safety checking problem into two obligations (tests) `test0` and `test1` (Listing 2.5). These tests use abstractions to check that each module satisfies its safety specification. Note that interfaces and the programmable bindings together enable substitution during compositional reasoning. For example, `ServerToClientIT` gets linked to `ServerImpl` in implementation but to its abstraction `AbstractServerImpl` during testing.

Meaningful testing requires that these abstractions used for decomposition be sound. To this end, MODP module system supports circular assume-guarantee reasoning (Theorem 2.4.4) to validate the abstractions. Tests `test2` and `test3` perform the necessary

refinement checking to ensure the soundness of the decomposition (`test0`,`test1`). The challenge addressed by our module system is to provide the theorems of compositional safety and circular assume-guarantee for a dynamic programming model of MODP state machines. MODP module system also provides module constructors like `hide` for hiding events (interfaces) and `rename` for renaming of conflicting actions for more flexible composition. Hide operation introduces private events (interfaces) into a module, it can be used to convert some of the visible actions of a module into private actions that are no longer part of its visible trace. For example, assume that modules `AbstractServerModule` and `ServerModule` use event `X` internally for completely different purposes. In that case, the refinement check between them is more likely to hold if `X` is not part of the visible trace of the abstract module. Listing 2.5 (line 23-27) show how hide can be used in such cases. Ensuring compositional refinement for a dynamic language like MODP is particularly challenging in the presence of private events (Section 2.2.2.2)

2.1.4 *Roadmap*

MODP's module system supports two key theorems for the compositional reasoning of distributed systems: *Compositional Safety* (Theorem 2.4.3) and *Circular Assume-Guarantee* (Theorem 2.4.4). We use Section 2.2.1 through Section 2.3.1 to build up to these theorems. The module system formalized in this paper can be adapted to any actor-oriented programming language provided certain extensions can be applied. We describe these extensions that MODP state machines make to the P state machines in Section 2.2.1. When defining the operational semantics of a module and to ensure that *composition is intersection*, it is essential that constructed modules be well-formed. Section 2.2.2 presents the type-checking rules to ensure well-formedness for a module. Section 2.3.1 presents the operational semantics of a well-formed module. Finally, we describe how we apply the theory of compositional refinement to test distributed systems (Section 3.1) and present our empirical results (Section 3.3).

2.2 MODP MODULE SYSTEM

2.2.1 MODP *State Machines*

A module in MODP is a collection of the dynamic instances of MODP state machines. In this section, we describe the extensions MODP state machines makes to P state machines in terms of syntactic constructs and semantics. These extensions to P state machines are required for defining the operational semantics of MODP modules and making them amenable to compositional reasoning.

(**Extension 1**): we add interfaces that are symbolic names for machines. In MODP, as described in Section 2.1.1, an instance of a machine is created indirectly by performing `new` of an interface (instead of `new` of a machine in P).
(**Extension 2**): we extend P machines with annotations declaring the set of receive, send and create actions the dynamic instance of that machine can perform. These annotations are used to statically infer the actions a module can perform based on the actions of its comprising machines.
(**Extension 3**): we extend the semantics of `send` in P to provide the guarantee that a MODP state machine can never receive an event (from any other machine) that is not listed in its receive set. This is achieved by extending machine identifiers with permissions (more details in Section 2.2.1.2).

2.2.1.1 *Semantics of* MODP *State Machines*

Let $\mathcal{E}$ represent the set of names of all the events. *Permissions* is a nonempty subset of $\mathcal{E}$; Let $\mathcal{K}$ represent the set of all permissions ($2^{\mathcal{E}} \setminus \{\emptyset\}$). Let $\mathcal{I}$ and $\mathcal{M}$ represent the sets of names of all interfaces and machines, respectively; these sets are disjoint from each other. Let $\mathcal{S}$ represent the set of all possible values the local state of a machine could have during execution. The local state of a machine represents everything that can influence the execution of the machine, including control stack and data structures. The buffer associated with a machine is modeled separately. Let $\mathcal{B}$ represent the set of all possible buffer values. The input buffer of a machine is a sequence of $(e, v) \in \mathcal{E} \times \texttt{Vals}()$ pairs, where `Vals()` represent the set of all possible payloads that may accompany any event in a send action. Let $\mathcal{Z}$ be the set of all the machine identifiers.

A MODP state machine is a tuple $(MRecvs, MSends, MCreates, Rem, Enq, New, Local)$ where:

1. $MRecvs \subseteq \mathcal{E}$ is the nonempty set of events received by the machine.
2. $MSends \subseteq \mathcal{E}$ is the set of all events sent by the machine.
3. $MCreates \subseteq \mathcal{I}$ is the set of interfaces created by the machine.
4. $Rem \subseteq \mathcal{S} \times \mathcal{B} \times \mathbb{N} \times \mathcal{S}$ is the transition relation for removing a message from the input buffer. If $(s, b, n, s') \in Rem$, then the n-th entry in the input buffer b is removed and the local state moves from s to s'.
5. $Enq \subseteq \mathcal{S} \times \mathcal{Z} \times \mathcal{E} \times \texttt{Vals}(\times)\mathcal{S}$ is the transition relation for sending a message to a machine. If $(s, id, e, v, s') \in Enq$, then event e with payload v is sent to machine id and the local state of the sender moves from s to s'.
6. $New \subseteq \mathcal{S} \times \mathcal{I} \times \mathcal{S}$ is the transition relation for creating an interface. If $(s, i, s') \in New$, then the machine linked against interface i is created and the machine moves from s to s'.

7. *Local* $\subseteq \mathcal{S} \times \mathcal{Z} \times \mathcal{S} \times \mathcal{Z}$ is the transition relation for local computation in the machine. The state of a machine is a pair $(s, id) \in \mathcal{S} \times \mathcal{Z}$. The first component s is the machine local-state. The second component *id* is a placeholder used to store the identifier of a freshly-created machine or to indicate the target of a send operation. If $(s, id, s', id') \in \mathit{Local}$, then the state can move from (s, id) to (s', id'), which allows us to model the movement of machine identifiers from s to *id* and vice-versa. The role of *id* will become clearer when we use it to define the operational semantics of the module (Section 2.3.1).

We refer to components of machine $m \in \mathcal{M}$ as $\mathit{MRecvs}(m)$, $\mathit{MSends}(m)$, $\mathit{MCreates}(m)$, $\mathit{Rem}(m)$, $\mathit{Enq}(m)$, $\mathit{New}(m)$, and $\mathit{Local}(m)$ respectively. We use $\mathit{IRecvs}(i)$ to refer to the receive set corresponding to an interface $i \in \mathcal{I}$.

2.2.1.2 *Machine Identifiers with Permissions*

A machine can send an event to another machine only if it has access to the receiver's machine identifier. The capability of a machine to send an event to another machine can change dynamically as machine identifiers can be passed from one machine to another.

Machine identifiers cannot be created out of thin air. A state machine can get access to a machine identifier either through a *remove* transition (*Rem*) where some other machine sent the identifier as a payload or through *create* transition (*New*) where it creates an instance of a machine. The assumption that machine identifiers cannot appear "out-of-thin-air" is formalized as follows. For all $m \in \mathcal{M}$, $s, s' \in \mathcal{S}$, $id, id' \in \mathcal{Z}$, $e \in \mathcal{E}$, $v \in \mathtt{Vals}()$, $i \in \mathcal{I}$, $b \in \mathcal{B}$, and $n \in \mathbb{N}$:

1. $(s, id, s', id') \in \mathit{Local}(m) \Rightarrow \mathit{ids}(s') \cup \{id'\} \subseteq \mathit{ids}(s) \cup \{id\}$.
2. $(s, b, n, s') \in \mathit{Rem}(m) \Rightarrow \mathit{ids}(s') \subseteq \mathit{ids}(s) \cup \{\mathit{ids}(v) \mid \exists e.\ b[n] = (e, v)\})$.
3. $(s, id, e, v, s') \in \mathit{Enq}(m) \Rightarrow \mathit{ids}(v) \cup \mathit{ids}(s') \subseteq \mathit{ids}(s)$.
4. $(s, i, s') \in \mathit{New}(m) \Rightarrow \mathit{ids}(s') \subseteq \mathit{ids}(s)$.

There are two key properties required for the compositional reasoning of communicating state machines using our module system: **(1)** *a machine never receives an event that is not in its receive set*, this property is required when formalizing the open module semantics of MODP modules and its receptiveness to input events (Section 2.3.1); **(2)** *the capability to send a private (internal) event of a module does not leak outside the module*, this property is required to ensure that compositional refinement in the presence of private events (Section 2.2.2.2). These properties are particularly challenging in the presence of machine-identifier that can flow freely. Our solution is similar in spirit to permissions based capability control for π-calculus [97, 159] where permissions are associated with channels or locations and enforced using type-systems.

We concretize the set of machine identifiers $\mathcal{Z}$ as $\mathcal{I} \times \mathbb{N} \times \mathcal{K}$. For our formalization, we are interested in machine identifiers that are embedded inside the data structures in a machine local-state $s \in \mathcal{S}$ or value $v \in \texttt{Vals}()$. Instead of formalizing all datatypes in MODP, we assume the existence of a function *ids* such that $ids(s)$ is the set containing all machine identifiers embedded in s and $\texttt{ids}(v)$ is the set containing all machine identifiers embedded in v. An identifier $(i, n, \alpha) \in \mathcal{Z}$ refers to the n-th instance of an interface represented by $i \in \mathcal{I}$. We refer to the final component α of a machine identifier as its *permissions*. The set α represents all the events that may be sent via this machine identifier using the `send` operation. The creation of an interface `I` returns a machine identifier $(\texttt{I}, n, \beta) \in \mathcal{Z}$ referencing to the n-th instance of interface `I` where β represents the receive set associated with the interface `I` ($\beta = IRecvs(\texttt{I})$). The MODP compiler checks that if an interface `I` is bound to `M` in a module, then the received events of `I` are contained in the received events of `M` ($IRecvs(\texttt{I}) \subseteq MRecvs(\texttt{M})$). Hence, the events that can be sent using an identifier is a subset of the events that the machine can receive. This mechanism ensures that a machine never receives an event that it has not declared in its receive set. Note that the permissions embedded in a machine identifier control the capabilities associated with that identifier.

In order to control the flow of these capabilities, MODP requires the programmer to annotate each event with a set $\mathcal{A} \in 2^{\mathcal{K}}$ of allowed permissions. For an event e, the set $\mathcal{A}(e)$ represents any permission that the programmer can put inside the payload accompanying e i.e., if v represents any legal payload value with e then $\forall(\_, \_, \alpha) \in ids(v), \alpha \in \mathcal{A}(e)$. In other words, $\mathcal{A}(e)$ represents the set of permissions that can be transferred from one machine to another using event e.

Finally, the modified send operation `send t,e,v` succeeds only if: **(1)** `e` is in the permissions of machine identifier `t`, to ensure `t` receives only those events that are in its receives set, and **(2)** all permissions embedded in `v` are in $\mathcal{A}(\texttt{e})$, the send fails otherwise (captured as the **(SendOk)** condition when defining the semantics of send in Section 2.3.1). This changed semantics of `send` based on permission-based capability control plays a crucial role in ensuring well-formedness of the hide operation that adds private events to a module (Section 2.2.2.2).

To statically check the permission that is passed using an event, we need to reflect the permission of a machine-reference stored in a variable in the variable's type. Recollect that, the type of a machine-reference variable is the name of an interface (Listing 2.1). An interface type represents the set of machine-identifiers whose permission is the receives events set of the interface. In other words, the type of a machine-identifier represents the permission stored in it. Thus, the static type of the payload associated with an event can be used to infer the permissions embedded in it and the check **(2)** above for the correctness of the send operation can be performed statically. We do not present the state-machine level typing rules for performing these checks statically because of space constraints; instead, they are described as dynamic checks when presenting the operational semantics in Section 2.3.1.

$$
\begin{array}{rcl}
\alpha \in 2^{\mathcal{E}} \quad \beta \in 2^{\mathcal{I}} \quad i, i', i_1, .., i_k \in \mathcal{I} \quad m_1, .., m_k \in \mathcal{M} & & \\
P, Q \in \mathit{ModuleExpr} & ::= & \textbf{bind } i_1 \to m_1, .., i_k \to m_k \\
& | & P \parallel Q \\
& | & \textbf{hide } \alpha \textbf{ in } P \\
& | & \textbf{hide } \beta \textbf{ in } P \\
& | & \textbf{rename } i \to i' \textbf{ in } P
\end{array}
$$

Figure 2.1: Module constructors

The module system formalized in this paper can be adopted to any actor-oriented programming language whose semantics is as described in Section 2.2.1.1 and can be extended with the three features (**Extension 1**) – (**Extension 3**).

2.2.2 MODP *Modules*

MODP seeks to manage the complexity of a distributed system by designing it in a structured way, at different levels of abstractions and modularly as the composition of interacting modules. Figure 2.1 presents the expression language supported by MODP module system for module construction.

The **bind** constructor creates a primitive module as a collection of machines $m_1, \ldots, m_k$ bound to interfaces $i_1, \ldots, i_k$ respectively (syntax is a bit different from the examples in Section 2.1). The composition ($\parallel$) constructor builds a complex module from simpler ones. The **hide** constructor creates an abstraction of a module, by converting some of its visible actions to private actions. The **rename** operation enables reuse of modules (and resolution of conflicting actions) when composing modules to create larger ones. The module language enables programmatic construction of modules, reuse of module expressions and ease of assembling modules for compositional reasoning (Section 2.4.1).

Well-formed module. In the MODP module system, a module P is a syntactic expression and its well-formedness is checked using the judgment $P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P$. If module P satisfies the judgment then we read it as: *Module P is well-formed with private events EP_P, private interfaces IP_P, interface definition map I_P, interface link map L_P, events received ER_P, events sent ES_P, and interfaces created IC_P*. The judgment derives the components on the right-hand side which are used for defining the operational semantics of a well-formed module (Section 2.3.1). We use $\mathtt{dom}(x)$ and $\mathit{codom}(x)$ to refer to the domain and codomain of any map x.

We next describe the components on the right-hand side of the judgment:

1. ***Private events.*** $EP_P \in 2^{\mathcal{E}}$ represents the private events for module P, these events must not cross the boundary of module P i.e. if a machine in P sends event $e \in EP_P$, then the target must be some machine in P and, if a machine in P

receives $e \in EP_P$, the sender must be some machine in P. The send of a private event is an internal (invisible) action of a module.

2. ***Private interfaces.*** $IP_P \in 2^{\mathcal{I}}$ represents the interfaces that are declared private in P; the creation of any interface in IP_P is an internal (invisible) action of P.

3. ***Interface definition map.*** $I_P \in \mathcal{I} \to \mathcal{M}$ interface definition map that binds an interface name i to a machine name $I_P[i]$. Recollect that in the MODP model of computation, dynamic instances of machines are created indirectly using interfaces. An interface definition map (I_P) is a collection of bindings from interface names to machine names. These bindings are initialized using the **bind** operation, so that if $(i, m) \in I_P$ then the creation of an interface i in module P leads to the creation of an instance of m.

4. ***Interface link map.*** $L_P \in \mathcal{I} \to \mathcal{I} \to \mathcal{I}$ is the *interface link map* that maps each interface $i \in \mathtt{dom}(I_P)$ to a machine link map that binds interfaces created by the code of machine $I_P[i]$ to an interface name. If the statement `new x` is executed by an instance of machine $I_P[i]$, an interface actually created in lieu of the interface name x is provided by the machine specific link map $L_P[i]$. If $(x, x') \in L_P[i]$, then the compiler interprets x in statement `new x` in the code of machine $I_P[i]$ as creation of interface x', creating an instance of machine $I_P[x']$.

The last three components of the judgment can be inferred using the first four components:

5. ***Events received.*** $ER_P \in 2^{\mathcal{E}}$ represent the set of events received by module P. It is inferred as the set of non-private events received by machines in P, $ER_P = \bigcup_{m \in codom(I_P)} MRecvs(m) \setminus EP_P$.

6. ***Events sent.*** $ES_P \in 2^{\mathcal{E}}$ represent the set of events sent by module P. It is inferred as the set of non-private events sent by machines in P, $ES_P = \bigcup_{m \in codom(I_P)} MSends(m) \setminus EP_P$.

7. ***Interfaces created.*** $IC_P \in 2^{\mathcal{I}}$ represent the set of interfaces created by module P. It is inferred as the set of interfaces created by machines in P (interpreted based on its link map), $IC_P = \bigcup_{(i,m) \in I_P, x \in MCreates(m)} \{L_P[i][x]\}$.

Exported interfaces. The domain of the interface definition map after removing the private interfaces is the set of exported interfaces for module P; these interfaces can be created either by P or its environment.

Input and output actions. The *input events* of module P are the events that are received but not sent by P i.e. $ER_P \setminus ES_P$. The *input interfaces* of P are the set of interfaces that are exported but not created by P i.e. $\mathtt{dom}(I_P) \setminus (IP_P \cup IC_P)$. The *output events* of P are

the sent events i.e. ES_P and the *output interfaces* are the created non-private interfaces of P i.e. $IC_P \setminus IP_P$. Informally, the *input actions* of a module is the union of its input events and input interfaces, the *output actions* of a module is the union of its output events and output interfaces (formally defined in Definition 2.3.1).

In the rest of this section, we describe the various module constructors and present the static rules to ensure that the constructed module satisfies: (1) well-formedness conditions (**WF1** – **WF3**) required for defining the semantics of a module, and (2) the compositionality Theorems 2.4.1- 2.4.2.

Note. For simplicity, when describing the static rules we do not provide the derivation for the last three components of the judgment as they can be inferred, but we use them above the line.

2.2.2.1 *Primitive Module*

In MODP, a primitive module is constructed using the **bind** operation. Programmatically initializing I_P using **bind** operation enables linking the creation of an interface `I` to either a concrete machine `Impl` for execution or an abstract machine `Abs` for testing, a key feature required for substitution during compositional reasoning.

(**Bind**)

$$\frac{f = \{(i_1, m_1), \ldots, (i_n, m_n)\} \quad f \subseteq \mathcal{I} \to \mathcal{M}^{(b1)} \qquad \forall (i, m) \in f.\ \mathit{IRecvs}(i) \subseteq \mathit{MRecvs}(m)^{(b2)}}{\textbf{bind}\ i_1 \to m_1, \ldots, i_n \to m_n \vdash \{\}, \{\}, f, \{(i, x, x) \mid (i, m) \in f \wedge x \in \mathit{MCreates}(m)\}}$$

Rule BIND presents the rule for **bind** $i_1 \to m_1, \ldots, i_k \to m_k$ that constructs a primitive module by binding each interface i_k to machine m_k for $k \in [1, n]$. These bindings are captured in f; condition **(b1)** checks that f is a function. Condition **(b2)** checks that the received events of an interface are contained in the received events of the machine bound to it (ensures **(WF1)** below). The resulting module does not have any private events and interfaces. The function f is the interface definition map and the interface link map for interface $i \in \mathtt{dom}(f)$ contains the identity binding for each interface created by $f(i)$ (ensures **(WF2)** below). The first entry for name x ever added to $L_P[i]$ is the identity map (x, x); subsequently, if interface x is renamed to x' (using **rename** constructor), this entry is updated to (x, x').

Well-formedness condition **(WF1)** helps ensure that a machine-identifier obtained by creating an interface can be used to send only those events that are in the receives set of the target machine (**(SendOk)** in Section 2.2.1.2).

(WF1) ***Interface definition map is consistent:*** *For each* $(i, m) \in I_P$, *we have* $\mathit{IRecvs}(i) \subseteq \mathit{MRecvs}(m)$.

Well-formedness condition **(WF2)** ensures that the link map lookups used during the create action always succeed.

(WF2) ***Interface link map is consistent:*** *The domains of I_P and L_P must be identical and for each $(i, m) \in I_P$ and $x \in MCreates(m)$, we have $x \in \texttt{dom}(L_P[i])$.*

2.2.2.2 *Hiding Events and Interfaces*

Hiding events and interfaces in a module allow us to construct a more abstract module [18]. There are two reasons to construct a more abstract version of a module P by hiding events or interfaces. First, suppose we want to check that another module `ServerModule` refines `AbstractServerModule`. But the event `X` is used for internal interaction among machines, for completely different purposes, in both `ServerModule` and `AbstractServerModule`. Then, the check that `ServerModule` refines `AbstractServerModule` is more likely to hold since sending of `X` is not a visible action of `AbstractServerModule`. Second, hiding helps make a module more composable with other modules. To compose two modules, the sent events and created interfaces of one module must be disjoint from the sent events and created interfaces of the other (Section 2.2.2.3). This restriction is onerous for large systems consisting of many modules, each of which may have been written independently by a different programmer. To address this problem, we relax disjointness for private events and interfaces, thus allowing incompatible modules to become composable after hiding conflicting events and interfaces.

To illustrate hiding of an event and an interface, we revisit the `ServerModule` in Listing 2.3. To legally hide an event in a module, it must be both a sent and received event of the module.

```
1 module HE_Server =
2 hide eProcessReq,eReqSuccess,eReqFail in ServerModule
```

Module `HE_Server` is well-formed and `eProcessReq`, `eReqSuccess`, `eReqFail` become private events in it. A send of event `eProcessReq` is a visible action in `ServerModule` but a private action in `HE_Server`. To hide an interface, it must be both an exported and created interface of that module.

```
1     module HI_Server = hide HelperIT in HE_Server
```

Module `HI_Server` is well-formed and interface `HelperIT` becomes a private interface in it. Creation of interface `HelperIT` is a visible action in `HE_Server` but a private action in `HI_Server`. *Hiding makes events and interfaces private to a module and converts output actions into internal actions.* All interactions between the server and the helper machine in `HI_Server` are private actions of the module.

Avoiding private permission leakage. Not requiring disjointness of private events creates a possibility for programmer error and a challenge for compositional refinement. When reasoning about a module P in isolation, only its input events (that are

disjoint from private events) would be considered as input actions. This is based on the assumption that private events of a module are exchanged only within a module, in other words, a private event of a module can never be sent by any machine outside the module to any machine inside the module.

Recollect that a machine can send only those events to a target machine that are in the permission set of the reference to the target machine (Section 2.2.1.2). Suppose a machine `M` in module `P` has a private event e in its set of received events. Any machine that possesses a reference to an instance of `M` could send e to this instance. If such a reference were to leak outside the module `P` to a machine in a different module, it would create an obstacle to reasoning about `P` separately (and proving the compositionality theorems for a module with private events), since the environment may now send private events targeted at a machine inside `P`. MODP ensures that such leakage of a machine reference with permissions containing a private event cannot happen.

In MODP, there are two ways for permissions to become available to a machine: (1) by creating an interface, or (2) by sending permissions to the machine in the payload accompanying some event. To tackle private permission leakage through (1), MODP requires that an input interface not have a private event in its set of received events so that an interface with private permissions cannot be created from outside the module. This is ensured by the condition **(he2)** below. To tackle private permission leakage through (2), MODP enforces that (a) each send of event e adheres to the specification **(SendOk)** in Section 2.2.1, and (b) the set of private events is disjoint from any permission in $\mathcal{A}(e)$ for any non-private event e (ensure **(WF3)** below). Together, these two checks ensure that permission containing a private event does not leak outside the module through sends.

(WF3) ***Permissions to send private events does not leak:*** *For all* $e \in ER_P \cup ES_P$ *and* $\alpha \in \mathcal{A}(e)$, *we have* $\alpha \cap EP_P = \emptyset$. This is a static check asserting the capabilities that can leak outside the module.

$$\textbf{(HideEvent)}\ \ (A \Delta B = (A \setminus B) \cup (B \setminus A))$$
$$\frac{\begin{array}{c} P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P \qquad \alpha \subseteq ER_P \cap ES_P^{(\textbf{he1})} \\ \forall x \in IC_P \Delta \mathtt{dom}(I_P).\ IRecvs(x) \cap \alpha = \emptyset^{(\textbf{he2})} \\ \forall e \in (ER_P \cup ES_P) \setminus \alpha.\ \forall \alpha' \in \mathcal{A}(e).\ \alpha \cap \alpha' = \emptyset^{(\textbf{he3})} \end{array}}{\textbf{hide}\ \alpha\ \textbf{in}\ P \vdash EP_P \cup \alpha, IP_P, I_P, L_P}$$

$$\textbf{(HideInterface)}$$
$$\frac{P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P \quad \beta \subseteq \mathtt{dom}(I_P) \cap IC_P^{(\textbf{hi1})}}{\textbf{hide}\ \beta\ \textbf{in}\ P \vdash EP_P, IP_P \cup \beta, I_P, L_P}$$

Rule HIDEEVENT handles the hiding of a set of events α in module `P`. This rule adds α to EP_P. Condition **(he1)** checks all events in β are both sent and received by module

P; this condition is required to ensure that the resulting module is an abstraction of P. Conditions **(he2)** and **(he3)** together ensure that once an event e becomes private, any permission containing e cannot cross the boundary of the resulting module (ensure **(WF3)**). Rule HIDEINTERFACE handles the hiding of a set of interfaces β in module P. This rule adds β to IP_P. Condition **(hi1)** is similar to the condition **(he1)** of rule HIDEEVENT; this condition ensures that the resulting module is an abstraction of P.

2.2.2.3 *Module Composition*

Module composition in MODP enforces an extra constraint that the output actions of the modules being composed are disjoint. The requirement of output disjointness i.e. output actions of P and Q be disjoint in order to compose them is important for compositional reasoning, especially to ensure that *composition is intersection* (Theorem 2.4.1). For defining the open system semantics of a module P (Section 2.3.1), we require P to be *receptive only* to its input actions (sent by its environment). In other words, for the input actions, P assumes that its environment will not send it any event sent by P itself. Similarly, P assumes that its environment will not create an interface that is created by P itself. Any input action of P that is an output action of Q is an output action of $P \parallel Q$ and hence not an input action of $P \parallel Q$. This property ensures that by composing P with a module Q (that outputs some input action of P), we achieve the effect of constraining the behaviors of P. Thus, the composition is a mechanism used to introduce details about the environment of a component, which constrains its behaviors (*composition is intersection*), and ultimately allows us to establish the safety properties of the component.

However, composition inevitably makes the size of the system larger thus making the testing problem harder. Hence, we need abstractions of components to allow precise yet compact modeling of the environment. If one component is replaced by another whose traces are a subset of the former, then the set of traces of the system only reduces, and not increases, i.e., no new behaviors are added (*trace containment is monotonic with respect to composition*: Theorem 2.4.2). This permits refinement of components in isolation.

$$\textbf{(Composition)}\ (A \Delta B = (A \setminus B) \cup (B \setminus A))$$
$$\frac{\begin{array}{c} P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P \quad Q \vdash EP_Q, IP_Q, I_Q, L_Q, ER_Q, ES_Q, IC_Q \\ \mathtt{dom}(I_P) \cap \mathtt{dom}(I_Q) = \emptyset^{(c1)} \qquad (ER_P \cup ER_Q \cup ES_P \cup ES_Q) \cap (EP_P \cup EP_Q) = \emptyset^{(c2)} \\ \forall x \in (\mathtt{dom}(I_P) \Delta IC_P) \cup (\mathtt{dom}(I_Q) \Delta IC_Q).\ IRecvs(x) \cap (EP_P \cup EP_Q) = \emptyset^{(c3)} \\ \forall e \in ER_P \cup ER_Q \cup ES_P \cup ES_Q.\ \forall \alpha \in \mathcal{A}(e).\ \alpha \cap (EP_P \cup EP_Q) = \emptyset^{(c4)} \\ IC_P \cap IC_Q = \emptyset^{(c5)} \qquad ES_P \cap ES_Q = \emptyset^{(c6)} \end{array}}{P \parallel Q \vdash EP_P \cup EP_Q, IP_P \cup IP_Q, I_P \cup I_Q, L_P \cup L_Q}$$

Rule COMPOSITION handles the composition of P and Q. Condition **(c1)** enforces that the domains of I_P and I_Q are disjoint, thus preventing conflicting interface bindings.

Conditions **(c2)** ensures that the input and output actions of P are not hidden by private events of Q and vice-versa. Conditions **(c3)** and **(c4)** together check that private permissions of P ‖ Q do not leak out. Condition **(c3)** checks that creation of an input interface of P does not leak permission containing a private event of Q and vice-versa. Condition **(c4)** checks that non-private events sent or received by P do not leak a permission containing a private event of Q and vice-versa (ensure **(WF3)**). Condition **(c5)** checks that created interfaces are disjoint; condition **(c6)** checks that sent events are disjoint. Composition is associative and commutative.

Example. If the conditions **(c1)** to **(c6)** hold then the composition of two modules is a union of its components. The composition operation acts as a language intersection. Consider the example of `ClientModule || ServerModule` from Listing 2.3. The interface `ServerToClientIT` is an input interface of `ServerModule` but becomes an output (no longer input) interface of `ClientModule || ServerModule`. Similarly, `eResponse` is an input event of `ClientModule` but becomes an output event of the composed module. Also, the union of the link-map and the interface definition maps ensures that the previously unbounded interfaces in link-map are appropriately bound after composition.

2.2.2.4 *Renaming Interfaces*

The `rename` module constructor allows us to rename conflicting interfaces before composition. The example in Listing 2.6 builds on top of the Client-Server example in Section 2.1. In module `ServerModule'`, the interface `ServerToClientIT'` is bound to machine `ServerImpl`. The creation of `HelperIT` interface (Listing 2.2) in `ServerImpl` machine is bound to `HelperImpl` machine in both `ServerModule` and `ServerModule'`. But, it is not possible to compose modules `ServerModule` and `ServerModule'` because of the conflicting bindings of interface `HelperIT` (rule COMPOSITION **(c1)**).

```
1  interface ServerToClientIT' receives eRequest,eReqFail;
2  interface HelperIT' receives eProcessReq;
3
4  machine HelperImpl' receives eProcessReq; sends ..; creates
       ..;
5  { /* body */ }
6
7  module ServerModule' =
8  {ServerToClientIT' → ServerImpl, HelperIT → HelperImpl};
9
10 module allServers =
11 ServerModule ‖ rename HelperIT → HelperIT' in ServerModule';
```

Listing 2.6: Renaming Interfaces Module Constructor

In Listing 2.6, the interface name `HelperIT` is renamed to `HelperIT'`. The rename module constructor updates the interface binding (`HelperIT` → `HelperImpl`) to (`HelperIT`

′ → HelperImpl)and the interface link map entry of (ServerToClientIT′ → HelperIT → HelperIT) to ()ServerToClientIT′ → HelperIT → HelperIT′). As a result, the composition of modules ServerModule and ServerModule′ is now possible.

Recollect that each module has an *interface link map* (Section 2.2.2) that maintains a machine specific mapping from the interface created by the code of a machine to the actual interface to be created in lieu of the new operation. The *interface link map* plays a critical role enable renaming of interfaces without changing the code of the involved machines. The execution of new HelperIT (Listing 2.2) in ServerImpl still leads to the creation of HelperImpl machine because of the indirection in the interface link map, and the corresponding visible action is creation of interface HelperIT′.

$$\textbf{ITE}(a, b, c)\text{: if a then b else c}$$

(Rename)

$$\frac{\begin{array}{c} P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P \qquad i \in \mathtt{dom}(I_P) \cup IC_P\,(\mathbf{r1}) \qquad IRecvs(i) = IRecvs(i')\,(\mathbf{r2}) \\ i' \in \mathcal{I} \setminus (\mathtt{dom}(I_P) \cup IC_P)\,(\mathbf{r3}) \qquad A = \{x \mid x' \in IP_P \wedge x = \textbf{ITE}(x' = i, i', x')\}\,(\mathbf{r4}) \\ B = \{(x, y) \mid (x', y) \in I_P \wedge x = \textbf{ITE}(x' = i, i', x')\}\,(\mathbf{r5}) \\ C = \{(x, y, z) \mid (x', y, z') \in L_P \wedge x = \textbf{ITE}(x' = i, i', x') \wedge z = \textbf{ITE}(z' = i, i', z')\}\,(\mathbf{r6}) \end{array}}{\textbf{rename } i \to i' \textbf{ in } P \vdash EP_P, A, B, C}$$

Rule RENAME handles the renaming of interface i to i' in module P. Condition **(r1)** checks that i is well-scoped; the set of $\mathtt{dom}(I_P) \cup IC_P$ is the universe of all interfaces relevant to P. Condition **(r2)** checks that the set of received events of i and i' are the same. Condition **(r3)** checks that i' is a new name different from the current set of interfaces relevant to P. Together with condition **(b2)** in rule BIND, this condition ensures that the set of received events of an interface is always a subset of the set of received events of the machine bound to it. Condition **(r4)** calculates in A the renamed set of private interfaces. Condition **(r5)** calculates in B the renamed interface definition map. Condition **(r6)** calculates in C the renamed interface link map.

2.3 OPERATIONAL SEMANTICS OF MODP MODULES

The MODP module system allows compositional reasoning of a module based on the principles of assume-guarantee reasoning. For assume-guarantee reasoning, the module system must guarantee that *composition is intersection* (Theorem 2.4.1), i.e., traces of a composed module are entirely determined by the traces of the component modules. We achieve this by first ensuring that a module is well-formed (Section 2.2.2), and then defining the operational semantics (as a set of traces) of a well-formed module such that its trace behavior (observable traces) satisfies the compositional trace semantics required for assume-guarantee reasoning.

In Section 2.2.2, a MODP module is described as a syntactic expression comprising of the module constructors listed in Figure 2.1. If the static rules are satisfied then

any constructed module P is well-formed and can be represented by its components $(EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P)$. In this section, we present the operational semantics of a well-formed module (Section 2.3.1) that help guarantee the key compositionality theorems described in Section 2.4.1.

2.3.1 *Operational Semantics of* MODP *Modules*

A key requirement for assume-guarantee reasoning [11, 128] is to consider each component as an *open system* that continuously reacts to input that arrives from its environment and generates outputs. The transitions (executions) of a module include non-deterministic interleaving of possible environment actions. Each component must be modeled as a labeled state-transition system so that traces of the component can be defined based only on the externally visible transitions of the system.

We refer to components on the right hand side of the judgment $P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P$ (Section 2.2.2) when defining the operational semantics of a well-formed module P. We present the *open system* semantics of a *well-formed* module P as a labeled transition system.

Configuration. A configuration of a module is a tuple (S, B, C): **(1)** The first component S is a partial map from $\mathcal{I} \times \mathbb{N}$ to $\mathcal{S} \times \mathcal{Z}$. If $(i, n) \in \mathtt{dom}(S)$, then $S[i, n]$ is the state of the n-th instance of machine $I_P[i]$. The state $S[i, n]$ has two components, local state $s \in \mathcal{S}$ and a machine identifier $id \in \mathcal{Z}$ (as described in Section 2.2.1.1). **(2)** The second component B is a partial map from $\mathcal{I} \times \mathbb{N}$ to $\mathcal{B}$. If $(i, n) \in \mathtt{dom}(B)$, then $B[i, n]$ is the input buffer of the n-th instance of the machine $I_P[i]$. **(3)** The third component C is a map from $\mathcal{I}$ to $\mathbb{N}$. $C[i] = n$ means that there are n dynamically created instances of interface i.

We present the operational semantics of a well-formed module P as a transition relation over its configurations. Let (S_P, B_P, C_P) represent the configuration for a module P. A transition is represented as $(S_P, B_P, C_P) \xrightarrow{a} (S'_P, B'_P, C'_P) \cup \{error\}$ where a is the label on a transition indicating the type of step being taken. The initial configuration of any module P is defined as (S^0_P, B^0_P, C^0_P) where S^0_P and B^0_P are empty maps, and C^0_P maps each element in its domain ($\mathcal{I}$) to 0.

Rules for local computation: Rules **(R1)**-**(R2)** present the rules for local computation of a machine. Rule INTERNAL picks an interface i and instance number n and updates $S[i, n]$ according to the transition relation *Local*, leaving B and C unchanged. The map I_P is used to obtain the concrete machine corresponding to the interface i. Rule REMOVE-EVENT updates $S[i, n]$ and $B[i, n]$ according to the transition relation $(s, b, pos, s') \in Rem(I_P[i])$, the entry in pos-th position of $B[i, n]$ is removed and the local state in $S[i, n]$ is updated to s' leaving the machine identifier (*id*) unchanged. The transition for both these rules is labeled with ϵ to indicate that the computation is local and is an internal transition of the module P.

$$\textbf{(Internal)}(\textbf{R1})\quad \frac{S_P[i,n] = (s, id) \qquad (s, id, s', id') \in Local(I_P[i])}{(S_P, B_P, C_P) \xrightarrow{\epsilon} (S_P[(i,n) \mapsto (s', id')], B_P, C_P)}$$

$$\textbf{(Remove-Event)}(\textbf{R2})\quad \frac{\begin{array}{c} S_P[i,n] = (s, id) \qquad B_P[i,n] = b \\ (s, b, pos, s') \in Rem(I_P[i])) \qquad b' = rem(b, pos) \end{array}}{(S_P, B_P, C_P) \xrightarrow{\epsilon} (S_P[(i,n) \mapsto (s', id)], B_P[(i,n) \mapsto b'], C_P)}$$

Figure 2.2: Operational Semantics Rules for Local Computation

Rules for creating interfaces: Let $s_0 \in \mathcal{S}$ represent a state such that $ids(s_0) = \emptyset$. Let $b_0 \in \mathcal{B}$ be the empty sequence over $\mathcal{E} \times \texttt{Vals}()$. Rules **(R3)-(R8)** present the rules for interface creation. In all the rules, I_P is used to look-up the machine name corresponding to an interface bound in module P. The environment of P triggers the first two rules, and the last four are triggered by P itself. The rule ENVIRONMENT-CREATE creates an interface that is neither created nor exported by P; consequently, it updates C by incrementing the number of instances of i but leaves S and B unchanged. The rule INPUT-CREATE creates an interface i exported by P that is not created by P. The instance number of the new interface is $C[i]$; its local-store is initialized to (s_0, id) where id in this case stores the "self" identifier that references the machine itself. Note that the environment cannot create an interface that is also created by P, which is based on the key assumption of *output disjointness* required for compositional reasoning (Section 2.2.2.3). The rule CREATE-BAD creates a transition into *error* if the interface i' being created by machine (i, n) violates the predicate $CreateOk(m, x) = x \in MCreates(m)$. Thus, machine (i, n) may only create machines in $MCreates(I_P[i])$.

We use machine (i, n) to refer to the n-th instance of the machine $I_P[i]$. OUTPUT-CREATE-OUTSIDE allows machine (i, n) to create an interface i'' that is not implemented inside P, indicated by $i'' \notin \texttt{dom}(I_P)$. Create of interface i'' will get bound to an appropriate machine when P is composed with another module Q that has binding for i'' i.e. $i'' \in \texttt{dom}(I_Q)$. The predicate $CreateOk(m, x) = x \in MCreates(m)$ checks that if a machine m performs `new x` then x belongs to its creates set. Thus, machine (i, n) may only create machines in $MCreates(I_P[i])$. A well-formed module satisfies the condition **(WF1)** together with the property that machines cannot create identifiers out of thin air to guarantee that the set of permissions in any machine identifier is a subset of the received events of the machine referenced by that identifier.

The rule OUTPUT-CREATE-INSIDE allows the creation of an interface that is exported by P. An interesting aspect of this rule is that the machine identifier made available to the creator machine has permission $IRecvs(i'')$ but the "self" identifier of the created machine is the entire receive set which may contain some private events in addition to

(Environment-Create)(R3)

$$\frac{i \in \mathcal{I} \setminus (IC_P \cup \mathtt{dom}(I_P)) \qquad n = C_P[i]}{(S_P, B_P, C_P) \xrightarrow{i} (S_P, B_P, C_P[i \mapsto n+1])}$$

(Input-Create)(R4)

$$\frac{i \in \mathtt{dom}(I_P) \setminus IC_P \qquad n = C_P[i] \qquad id = (i, n, \mathit{IRecvs}(i))}{(S_P, B_P, C_P) \xrightarrow{i} (S_P[(i,n) \mapsto (s_0, id)], B_P[(i,n) \mapsto b_0], C_P[i \mapsto n+1])}$$

(Create-Bad)(R5)

$$\frac{S_P[i,n] = (s,\_) \qquad (s,i',\_) \in \mathit{New}(I_P[i]) \qquad \neg \mathit{CreateOk}(I_P[i], i')}{(S_P, B_P, C_P) \xrightarrow{\epsilon} \mathit{error}}$$

(Output-Create-Outside)(R6)

$$\frac{\begin{array}{c} S_P[i,n] = (s,\_) \qquad (s,i',s') \in \mathit{New}(I_P[i]) \qquad \mathit{CreateOk}(I_P[i], i') \\ i'' = L_P[i][i'] \qquad n' = C_P[i''] \\ i'' \notin \mathtt{dom}(I_P) \qquad id' = (i'', n', \mathit{IRecvs}(i'')) \end{array}}{(S_P, B_P, C_P) \xrightarrow{i''} (S_P[(i,n) \mapsto (s', id')], B_P, C_P[i'' \mapsto n'+1])}$$

(Output-Create-Inside)(R7)

$$\frac{\begin{array}{c} S_P[i,n] = (s,\_) \\ (s,i',s') \in \mathit{New}(I_P[i]) \qquad \mathit{CreateOk}(I_P[i], i') \qquad i'' = L_P[i][i'] \qquad i'' \in \mathtt{dom}(I_P) \setminus IP_P \\ n' = C_P[i''] \qquad id' = (i'', n', \mathit{IRecvs}(i'')) \qquad id'' = (i'', n', \mathit{MRecvs}(I_P[i''])) \end{array}}{\begin{array}{c} (S_P, B_P, C_P) \xrightarrow{i''} (S_P[(i,n) \mapsto (s', id'), (i'', n') \mapsto (s_0, id'')], \\ B_P[(i'', n') \mapsto b_0], C_P[i'' \mapsto n'+1]) \end{array}}$$

(Create-Private)(R8)

$$\frac{\begin{array}{c} S_P[i,n] = (s,\_) \qquad (s,i',s') \in \mathit{New}(I_P[i]) \qquad \mathit{CreateOk}(I_P[i], i') \qquad i'' = L_P[i][i'] \\ i'' \in IP_P \qquad n' = C_P[i''] \qquad id' = (i'', n', \mathit{IRecvs}(i'')) \qquad id'' = (i'', n', \mathit{MRecvs}(I_P[i''])) \end{array}}{(S_P, B_P, C_P) \xrightarrow{\epsilon} (S_P[(i,n) \mapsto (s', id'), (i'', n') \mapsto (s_0, id'')], B_P[(i'', n') \mapsto b_0], C_P[i'' \mapsto n'+1])}$$

Figure 2.3: Operational Semantics Rules for Creating Interfaces

(Input-Send)(R9)

$$\frac{B_P[i,n] = b \qquad e \in MRecvs(I_P[i]) \setminus (EP_P \cup ES_P) \qquad v \in \texttt{Vals}() \qquad \forall (i',n',\alpha') \in ids(v).\ \alpha' \in \mathcal{A}(e) \wedge n' < C_P[i']}{(S_P, B_P, C_P) \xrightarrow{((i,n),e,v)} (S_P, B_P[(i,n) \mapsto b \odot (e,v)], C_P)}$$

(Send-Bad)(R10)

$$\frac{S_P[i,n] = (s, id_t) \qquad id_t = (\_,\_,\alpha_t) \qquad (s, id_t, e, v, \_) \in Enq(I_P[i]) \qquad \neg SendOk(I_P[i], \alpha_t, e, v)}{(S_P, B_P, C_P) \xrightarrow{e} error}$$

(Output-Send-Outside)(R11)

$$\frac{S_P[i,n] = (s, id_t) \qquad id_t = (i_t, n_t, \alpha_t) \qquad i_t \notin \texttt{dom}(I_P) \qquad (s, id_t, e, v, s') \in Enq(I_P[i]) \qquad SendOk(I_P[i], \alpha_t, e, v)}{(S_P, B_P, C_P) \xrightarrow{((i_t,n_t),e,v)} (S_P[(i,n) \mapsto (s', id_t)], B_P, C_P)}$$

(Output-Send-Inside)(R12)

$$\frac{S_P[i,n] = (s, id_t) \qquad id_t = (i_t, n_t, \alpha_t) \qquad i_t \in \texttt{dom}(I_P) \qquad e \in ES_P \qquad b_t = B_P[i_t, n_t] \qquad (s, id_t, e, v, s') \in Enq(I_P[i]) \qquad SendOk(I_P[i], \alpha_t, e, v)}{(S_P, B_P, C_P) \xrightarrow{((i_t,n_t),e,v)} (S_P[(i,n) \mapsto (s', id_t)], B_P[(i_t,n_t) \mapsto b_t \odot (e,v)], C_P)}$$

(Send-Private)(R13)

$$\frac{S_P[i,n] = (s, id_t) \qquad id_t = (i_t, n_t, \alpha_t) \qquad i_t \in \texttt{dom}(I_P) \qquad e \in EP_P \qquad b_t = B_P[i_t, n_t] \qquad (s, id_t, e, v, s') \in Enq(I_P[i]) \qquad SendOk(I_P[i], \alpha_t, e, v)}{(S_P, B_P, C_P) \xrightarrow{e} (S_P[(i,n) \mapsto (s', id_t)], B_P[(i_t,n_t) \mapsto b_t \odot (e,v)], C_P)}$$

Figure 2.4: Operational Semantics Rules for Sending Events

all events in $IRecvs(i'')$. Allowing extra private events in the permission of the "self" identifier is useful if the machine wants to send permissions to send private events to a sibling machine inside P. In all these rules, the link map (L_P) is used to look up the interface i'' to be created corresponding to `new` i'. The condition **(WF2)** holds for any well-formed module and guarantees that this lookup always succeeds.

Rules for sending events: Rules **(R9)**-**(R13)** present the rules for sending events. The environment of P triggers the first rule, and the last two are triggered by P itself. The rule INPUT-SEND enqueues a pair (e, v) into machine (i, n) if $e \in MRecvs(I_P[i])$ and e is neither private in P nor sent by P and v does not contain any machine identifiers with private events in its permissions. First, an event that is sent by P is not considered as an input event, which is safe since rules of *output-disjointness* (Section 2.2.2.3) forbid composing P with another module that sends an event in common with P. Second, only an event in the receives set of a machine is considered as an input event, because

any machine can send only those events that are in the permission of an identifier and the permission set of an identifier is guaranteed to be a subset of the receives set of the machine referenced by it (based on **(WF1)**). Finally, private events or payload values with private events in its permissions are not considered as input because permission to send a private event cannot leak out of a well-formed module (based on **(WF3)**).

Before executing a send statement the target machine identifier is loaded into the local store represented by id_t using an internal transition. The predicate $SendOk(\hat{m}, \alpha, e, v) = e \in MSends(\hat{m}) \wedge e \in \alpha \wedge \forall(\_, \_, \beta) \in ids(v).\ \beta \in \mathcal{A}(e)$) captures the **(SendOk)** specification described in Section 2.2.1.2. Thus, machine (i, n) may only send events declared by it in $MSends(I_P[i])$ and allowed by the permission α_t of the target machine and should not embed machine identifiers with private permissions in the payload v. Note that the dynamic check **(SendOk)** helps guarantee the well-formedness condition **(WF3)** and also ensures that a module receives only those events from other modules that are its input events (and is expected to be receptive against).

The rule OUTPUT-SEND-OUTSIDE sends an event to machine outside P whereas rules OUTPUT-SEND-INSIDE and SEND-PRIVATE send an event to some machine inside P. In the former, the target machine m_t is not in the domain of I_P, whereas in the latter cases the target machine is inside the module and hence present in the domain of I_P. For SEND-PRIVATE, the label on the transition is ϵ as a private event is sent. For brevity, we refer to a configuration (S^k, B^k, C^k) as G^k.

Definition 2.3.1: Execution

An execution *of* P *is a finite sequence* $G^0 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} G^n$ *for some* $n \in \mathbb{N}$ *such that* $G^i \xrightarrow{a_i} G^{i+1}$ *for each* $i \in [0, n)$.

Let $execs(P)$ represent the set of all possible executions of the module P.

Invariants for Executions of a Module

For any execution $\tau \in execs(P)$ where τ is a sequence of global configurations $G_0 \xrightarrow{a_0} G_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} G_n$, all global configurations G_i satisfy the invariants:

I1 $\mathtt{dom}(S_P) = \mathtt{dom}(B_P)$

I2 $\forall(i, n) \in \mathtt{dom}(B_P).\ i \in \mathtt{dom}(I_P) \wedge n < C[i]$

I3 $\forall i \in \mathtt{dom}(I_P).\ C[i] = card(\{n \mid (i, n) \in \mathtt{dom}(B_P)\})$

I4 $\forall(x, n, \alpha) \in ids(S_P) \cup ids(B_P).\ x \in \mathtt{dom}(I_P) \Rightarrow (x, n) \in \mathtt{dom}(B_P)$

I5 $\forall(x, n, \alpha) \in ids(S_P) \cup ids(B_P).\ n < C_P[x]$

An execution is *unsafe* if $G^n \xrightarrow{\epsilon} \mathit{error}$; otherwise, it is *safe*. The module P is *safe*, if for all $\tau \in \mathit{execs}(P)$, τ is a safe execution. The signature of a module P is the set of labels corresponding to all externally visible transitions in executions of P.

Definition 2.3.2: Module Signature

The signature of a module P *is the set* $\Sigma_P = (\mathcal{I} \setminus IP_P) \cup ((\mathcal{I} \times \mathbb{N}) \times (ES_P \cup ER_P) \times \texttt{Vals}())$. *The signature is partitioned into the* output signature $(IC_P \setminus IP_P) \cup ((\mathcal{I} \times \mathbb{N}) \times ES_P \times \texttt{Vals}())$ *and the* input signature $(\mathcal{I} \setminus IC_P) \cup ((\mathcal{I} \times \mathbb{N}) \times (ER_P \setminus ES_P) \times \texttt{Vals}())$.

The transitions in an execution labeled by elements of the output signature are the **output actions** whereas transitions labeled by elements of the input signature are the **input actions**.

Definition 2.3.3: Trace

Given an execution $\tau = G^0 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} G^n$ *of* P, *the trace of* τ *is the sequence* σ *obtained by removing occurrences of* ϵ *from the sequence* $a_1, \ldots, a_{n-1}$.

Let $\mathit{traces}(P)$ represents the set of all possible traces of P. Our definition of a trace captures externally visible operations that add dynamism in the system like machine creation and sends with a payload that can have machine-references. If $\sigma \in \mathit{traces}(P)$ then $\sigma[\Sigma_P]$ represents the projection of trace σ over the set Σ_P where if $\sigma = a_0, \ldots, a_n$, then $\sigma[\Sigma_P]$ is the sequence obtained after removing all a_i such that $a_i \notin \Sigma_P$.

Definition 2.3.4: Refinement

The module P refines *the module* Q, *written* $P \preceq Q$, *if the following conditions hold: (1)* $IC_Q \setminus IP_Q \subseteq IC_P \setminus IP_P$, *(2)* $\texttt{dom}(I_Q) \setminus IP_Q \subseteq (\texttt{dom}(I_P) \cup IC_P) \setminus IP_P$, *(3)* $ES_Q \subseteq ES_P$, *(4)* $ER_Q \subseteq ER_P \cup ES_P$ *(note that (1)-(4) together imply* $\Sigma_Q \subseteq \Sigma_P$*), (5) and for every trace* σ *of* P *the projection* $\sigma[\Sigma_Q]$ *is a trace of* Q.

2.4 COMPOSITIONAL REASONING USING MODP MODULES

2.4.1 *Principles of Assume-Guarantee Reasoning*

The two fundamental compositionality results required for assume-guarantee reasoning are:

Theorem 2.4.1: Composition Is Intersection

Let P, Q *and* $\mathsf{P}\|\mathsf{Q}$ *be well-formed modules. For any* $\pi \in \Sigma^*_{\mathsf{P}\|\mathsf{Q}}$, $\pi \in \mathit{traces}(\mathsf{P}\|\mathsf{Q})$ *iff* $\pi[\Sigma_{\mathsf{P}}] \in \mathit{traces}(\mathsf{P})$ *and* $\pi[\Sigma_{\mathsf{Q}}] \in \mathit{traces}(\mathsf{Q})$.

Theorem 2.4.1 states that composition of modules behaves like language intersection, the traces of the component modules completely determine traces of a composed module.

Theorem 2.4.2: Composition Preserves Refinement

Let P, Q, *and* R *be well-formed modules such that* $\mathsf{P}\|\mathsf{Q}$ *and* $\mathsf{P}\|\mathsf{R}$ *are well-formed. Then* $\mathsf{R} \preceq \mathsf{Q}$ *implies that* $\mathsf{P}\|\mathsf{R} \preceq \mathsf{P}\|\mathsf{Q}$.

Theorem 2.4.2 states that parallel composition is monotonic with respect to trace inclusion i.e. if one module is replaced by another whose traces are a subset of the former, then the set of traces of the resultant composite module can only be reduced.

Theorems 2.4.1 and 2.4.2 form the basis of our theory of compositional refinement and are used for proving the principles of circular assume-guarantee reasoning underlying our compositional testing methodology (Theorems 2.4.3-2.4.4). We introduce a generalized composition operation $\|\,\mathcal{P}$, where $\mathcal{P}$ is a non-empty set of modules. This operator represents the composition of all modules in $\mathcal{P}$. The binary parallel composition operator is both commutative and associative. Thus, $\|\,\mathcal{P}$ is a module obtained by composing modules in $\mathcal{P}$ in some arbitrary order. Let $\mathcal{P}$ and $\mathcal{Q}$ be set of modules. We say that $\mathcal{P}$ is a subset of $\mathcal{Q}$ if $\mathcal{P}$ can be obtained by dropping modules in $\mathcal{Q}$.

Theorem 2.4.3: Compositional Safety

Let $\|\,\mathcal{P}$ *and* $\|\,\mathcal{Q}$ *be well-formed. Let* $\|\,\mathcal{P}$ *refine each module* $\mathsf{Q} \in \mathcal{Q}$. *Suppose for each* $\mathsf{P} \in \mathcal{P}$, *there is a subset* $\mathcal{X}$ *of* $\mathcal{P} \cup \mathcal{Q}$ *such that* $\mathsf{P} \in \mathcal{X}$, $\|\,\mathcal{X}$ *is well-formed, and* $\|\,\mathcal{X}$ *is safe. Then* $\|\,\mathcal{P}$ *is safe.*

When using Theorem 2.4.3 in practice, modules in $\mathcal{P}$ and $\mathcal{Q}$ typically consists of the implementation and abstraction modules respectively. When proving the safety of any module $\mathsf{P} \in \mathcal{P}$, it is allowed to pick any modules in $\mathcal{Q}$ for constraining the environment of P. To use Theorem 2.4.3, we need to show that $\|\,\mathcal{P}$ refines each module $\mathsf{Q} \in \mathcal{Q}$ which requires reasoning about all modules in $\mathcal{P}$ together. The following theorem shows that the refinement between $\|\,\mathcal{P}$ and $\mathcal{Q}$ can also be checked compositionally.

Theorem 2.4.4: Circular Assume-Guarantee

Let $\|\mathcal{P}$ and $\|\mathcal{Q}$ be well-formed. Suppose for each module $Q \in \mathcal{Q}$ there is a subset $\mathcal{X}$ of $\mathcal{P} \cup \mathcal{Q}$ such that $Q \notin \mathcal{X}$, $\|\mathcal{X}$ is well-formed, and $\|\mathcal{X}$ refines Q. Then $\|\mathcal{P}$ refines each module $Q \in \mathcal{Q}$.

Theorem 2.4.4 states that to show that $\|\mathcal{P}$ refines $Q \in \mathcal{Q}$, any subset of modules in $\mathcal{P}$ and $\mathcal{Q}$ can be picked as long as Q is not picked. Therefore, it is possible to perform sound *circular* reasoning, i.e., use Q_1 to prove refinement of Q_2 and Q_2 to prove refinement of Q_1. This capability of circular reasoning is essential for compositional testing of the distributed systems we have implemented.

Note that $\|\mathcal{P}$ refines every submodule of $\mathcal{Q}$ is implied by $\|\mathcal{P}$ refines module $\|\mathcal{Q}$. If $\|\mathcal{P}$ refines $\|\mathcal{Q}$, then using Theorem 2.4.1, $\|\mathcal{P}$ would refine each individual submodule in $\mathcal{Q}$ as well. Similarly, if $\|\mathcal{P}$ refines every submodule of $\mathcal{Q}$ and $\|\mathcal{Q}$ is a well-formed module, then $\|\mathcal{P}$ refines module $\|\mathcal{Q}$.

2.4.2 *Proofs and Lemmas for the* MODP *Module System*

Summary

The MODP module system provide the following important top-level theorems and lemmas:

1. **Composition Is Intersection:** Composition behaves like language intersection. This is captured by the Theorem 2.4.1, which asserts that traces of a composed module are completely determined by the traces of the component modules. This Lemma forms the basis and used by the rest of the lemmas.

2. **Composition Preserves Refinement:** The traces of a composed module is a subset of the traces of each component module. Hence, the composition of two modules creates a new module which is equally or more detailed than its components. This is captured by the Lemma 2.4.3.

3. **Circular Assume-Guarantee:** Theorem 2.4.4 states that to show $\| \mathcal{P}$ refines $Q \in \mathcal{Q}$, any subset of modules in $\mathcal{P}$ and $\mathcal{Q}$ can be picked as long as Q is not picked. Therefore, it is possible to perform sound *circular* reasoning, i.e., use Q_1 to prove Q_2 and Q_2 to prove Q_1.

4. **Compositional Safety Analysis:** Theorem 2.4.3 talks about implementation modules in $\mathcal{P}$ and abstraction modules in $\mathcal{Q}$. When proving safety of any module $P \in \mathcal{P}$, it is allowed to pick any modules in $\mathcal{Q}$ for constraining the environment of P.

5. **Hide Event Preserves Refinement:** Lemma 2.4.6 states that the hide event operation preserves refinement, is compositional and create a sound abstraction of the module.

6. **Hide Interface Preserves Refinement:** Lemma 2.4.7 states that the hide interface operation preserves refinement, is compositional and create a sound abstraction of the module.

In the rest of this section, we present the proofs for the theorems introduced in Section 2.4.1 and the also the lemmas supported by the MODP Module system. We first present the definitions needed for the formalism and proofs in this section.

1. Let $\mathcal{G}$ be the set of all possible configurations. For a configuration $G = (S, B, C)$, we refer to its elements as G_S, G_B, and G_C respectively.

2. Let $\texttt{last}$ be a function that given an execution which is a sequence of alternating global configuration and transition labels returns the last global configuration state. If $\tau = G^0 \xrightarrow{a_0} G^1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} G^n$ then $\texttt{last}(\tau) = G^n$

3. Let $\texttt{trace}(\tau_P)$ represent the trace corresponding to execution $\tau_P \in \mathit{execs}(P)$.

4. Two configurations $G, G' \in \mathcal{G}$ are compatible, if the following conditions hold:
   a) $\forall (i,n) \in (\texttt{dom}(G_S) \cap \texttt{dom}(G'_S))$, $G_S[i,n] = G'_S[i,n]$,
   b) $\forall (i,n) \in (\texttt{dom}(G_B) \cap \texttt{dom}(G'_B))$, $G_B[i,n] = G'_B[i,n]$,
   c) $\forall i \in (\texttt{dom}(G_C) \cap \texttt{dom}(G'_C))$, $G_C[i] = G'_C[i]$

   Informally, two configurations are compatible, if each element in the configurations agree on the common values in their domain.

5. Let $\texttt{union}$ be a partial function from $(\mathcal{G} \times \mathcal{G})$ to $\mathcal{G}$ satisfying the following properties:
   a) $(G, G') \in \texttt{dom}(\texttt{union})$ iff G and G' are compatible.
   b) $(G^p, G^q, G^c) \in \texttt{union}$ iff $G^c = (G^p_S \cup G^q_S, G^p_B \cup G^q_B, G^p_C \cup G^q_C)$.

We prove the Theorem 2.4.1 by proving two simpler lemmas, Lemma 2.4.1 and Lemma 2.4.2. The proof is decomposed into the following two implications:
Forward Implication for traces:
If $\sigma \in \mathit{traces}(P||Q)$ then the projection $\sigma[\Sigma_P] \in \mathit{traces}(P)$ and the projection $\sigma[\Sigma_Q] \in \mathit{traces}(Q)$. This follows from the Lemma 2.4.1.
Backward Implication for traces:
If there exists a sequence $\sigma \in \Sigma^*_{P||Q}$ such that $\sigma[\Sigma_P] \in \mathit{traces}(P)$ and $\sigma[\Sigma_Q] \in \mathit{traces}(Q)$, then $\sigma \in \mathit{traces}(P||Q)$. This follows from the Lemma 2.4.2.

Lemma 2.4.1

For any execution $\tau_c \in \mathit{execs}(P||Q)$, there exists an execution $\tau_p \in \mathit{execs}(P)$ such that $\texttt{trace}(\tau_p)[\Sigma_P] = \texttt{trace}(\tau_c)[\Sigma_P]$ and there exists an execution $\tau_q \in \mathit{execs}(Q)$ such that $\texttt{trace}(\tau_q)[\Sigma_Q] = \texttt{trace}(\tau_c)[\Sigma_Q]$.

Proof. We perform induction over the length of execution τ_c of the composed module $P||Q$.

Inductive Hypothesis: For every execution $\tau_c \in \mathit{execs}(P||Q)$, there exists an execution $\tau_p \in \mathit{execs}(P)$ such that $\texttt{trace}(\tau_p)[\Sigma_P] = \texttt{trace}(\tau_c)[\Sigma_P]$, there exists an execution $\tau_q \in \mathit{execs}(Q)$ such that $\texttt{trace}(\tau_q)[\Sigma_Q] = \texttt{trace}(\tau_c)[\Sigma_Q]$, and $\texttt{last}(\tau_c) = \texttt{union}(\texttt{last}(\tau_p), \texttt{last}(\tau_q))$.

We refer to the elements of the global configuration $\texttt{last}(\tau_c)$ as $\texttt{last}(\tau_c)_S$, $\texttt{last}(\tau_c)_B$, $\texttt{last}(\tau_c)_C$.

Base case: The base case for the inductive proof is for an execution τ_c of length 0, $\tau_c \in \mathit{execs}(P\|Q)$. The projection of the execution τ_c over the alphabet of the individual modules results in a execution of length zero which belongs to the set of executions of all the modules. We know that, for the base case there exists an execution $\tau_p \in \mathit{execs}(P)$ and $\tau_q \in \mathit{execs}(Q)$ of length zero such that $\texttt{last}(\tau_c) = \texttt{union}(\texttt{last}(\tau_p), \texttt{last}(\tau_q))$. Hence, the inductive hypothesis holds for the base case.

Inductive case: Let us assume that the hypothesis holds for any execution $\tau_c \in \mathit{execs}(P\|Q)$. Let τ_p and τ_q be the corresponding executions for module P and Q such that $\texttt{trace}(\tau_c)[\Sigma_P] = \texttt{trace}(\tau_p)[\Sigma_P]$, $\texttt{trace}(\tau_c)[\Sigma_Q] = \texttt{trace}(\tau_q)[\Sigma_Q]$ and $\texttt{last}(\tau_c) = \texttt{union}(\texttt{last}(\tau_p), \texttt{last}(\tau_q))$.

To prove that the hypothesis is inductive we show that it also holds for the execution $\tau_c' \in \mathit{execs}(P\|Q)$ where $\tau_c' = \tau_c \xrightarrow{a} G$ and τ_p', τ_q' be the corresponding executions of P and Q.

We perform case analysis for all possible transitions labels a.

1. $a = \epsilon$
   This is the case when the composed module $P\|Q$ takes an invisible transition. Lets say n-th instance of an interface i identified by $(i, n) \in \texttt{dom}(\texttt{last}(\tau_c)_S)$ made an invisible transition. This could be because the machine took any of the following transitions: INTERNAL, REMOVE-EVENT, CREATE-BAD,Output-Create-3, SEND-BAD, and Output-Send-3.

   Consider the case when $i \in \texttt{dom}(I_P)$ i.e. machine corresponding to interface i is implemented in module P.

   Based on the assumption that $\texttt{last}(\tau_c) = \texttt{union}(\texttt{last}(\tau_p), \texttt{last}(\tau_q))$, we know that $\texttt{last}(\tau_c)_S[i, n] = \texttt{last}(\tau_p)_S[i, n]$ and $\texttt{last}(\tau_c)_B[i, n] = \texttt{last}(\tau_p)_B[i, n]$. Hence, if machine instance (i, n) in $P\|Q$ can make an invisible transition a when in global configuration $\texttt{last}(\tau_c)$, then the same invisible transition can be taken by module P in configuration $\texttt{last}(\tau_p)$. Hence, $\texttt{trace}(\tau_p')[\Sigma_P] = \texttt{trace}(\tau_c')[\Sigma_P]$ (where $\tau_p' = \tau_p \xrightarrow{a} G'$). Since $a = \epsilon$, $\texttt{trace}(\tau_q)[\Sigma_Q] = \texttt{trace}(\tau_c')[\Sigma_Q]$

   Note that the invisible transitions do not change the map C. Since, the module $P\|Q$ and P took the same transition a and configuration of module Q has not changed, the resultant configurations satisfy the property $\texttt{last}(\tau_c') = \texttt{union}(\texttt{last}(\tau_p'), \texttt{last}(\tau_q))$.

   The same analysis can be applied to the case when $m \in \texttt{dom}(M_Q)$.

2. $a = i$ where $i \in \mathcal{I}$
   This is the case when the composed module or the environment takes the visible transition of creating an interface i. We perform case analysis for all such possible transitions:

a) ENVIRONMENT-CREATE
Consider the case when the environment of module P||Q takes a transition to create an interface i. If i is created by P||Q using the ENVIRONMENT-CREATE, then it can be created by P and Q only using the ENVIRONMENT-CREATE rule. This comes from the fact that i does not belong to $IC_{P||Q}$ and $\texttt{dom}(I_{P||Q})$.

Hence the environment of both P and Q can take the transition and the resultant executions τ'_p, τ'_q will satisfy the condition $\texttt{last}(\tau'_c) = \texttt{union}(\texttt{last}(\tau'_p), \texttt{last}(\tau'_q)$, $\texttt{trace}(\tau'_c)[\Sigma_P] = \texttt{trace}(\tau'_p)[\Sigma_P]$, $\texttt{trace}(\tau'_c)[\Sigma_Q] = \texttt{trace}(\tau'_q)[\Sigma_Q]$.

b) INPUT-CREATE
Our definition of composition and compatibility guarantees that if P||Q is well-formed then:

i. $\texttt{dom}(I_{P||Q}) = \texttt{dom}(I_P) \cup \texttt{dom}(I_Q)$

ii. $\texttt{dom}(I_P) \cap \texttt{dom}(I_Q) = \emptyset$

Hence, if the composed module P||Q receives an input create request for $i \in \texttt{dom}(I_P) \subset IC_{I_p}$ from the environment, then either $i \in \texttt{dom}(I_P)$, or $i \in \texttt{dom}(I_Q)$. Also, since $i \notin IC_{P||Q}$, it implies that $i \notin IC_Q$ and $i \notin IC_P$.

Consider the case when $i \in \texttt{dom}(I_P)$. Based on the assumption that $\texttt{last}(\tau_c) = \texttt{union}(\texttt{last}(\tau_p), \texttt{last}(\tau_q))$, we know that $\texttt{last}(\tau_c)_S[i, n] = \texttt{last}(\tau_p)_S[i, n]$ and $\texttt{last}(\tau_c)_B[i, n] = \texttt{last}(\tau_p)_B[i, n]$. Hence, if P||Q takes the visible INPUT-CREATE transition i, when in global configuration $\texttt{last}(\tau_c)$, then the same transition can be taken by module P in configuration $\texttt{last}(\tau_p)$. $i \in \Sigma_Q$ (*we know that* $i \notin (\texttt{dom}(I_Q) \cup IC_Q)$), hence Q takes the ENVIRONMENT-CREATE transition. The resultant executions τ'_c, τ'_p and τ'_q satisfy the condition that $\texttt{last}(\tau'_c) = \texttt{union}(\texttt{last}(\tau'_p), \texttt{last}(\tau'_q))$. Also, $\texttt{trace}(\tau'_c)[\Sigma_P] = \texttt{trace}(\tau'_p)[\Sigma_P]$ and $\texttt{trace}(\tau'_c)[\Sigma_Q] = \texttt{trace}(\tau'_q)[\Sigma_Q]$ since all modules took the same labeled transition.

The same analysis can be applied to the case when $i \in \texttt{dom}(I_Q)$.

c) OUTPUT-CREATE-1
This is the case when a machine instance $(i', n) \in \texttt{dom}(\texttt{last}(\tau_c)_S)$ creates an interface i and $i \notin \texttt{dom}(I_{P||Q})$ which means that interface i is implemented by some machine in the environment of P||Q.

Consider the case when $i' \in \texttt{dom}(I_P)$ (which implies that $i' \notin \texttt{dom}(I_Q)$), since $i \notin \texttt{dom}(I_{P||Q})$ we know that $i \notin \texttt{dom}(I_P)$ and $i \notin \texttt{dom}(I_Q)$.

Based on the assumption that $\texttt{last}(\tau_c) = \texttt{union}(\texttt{last}(\tau_p), \texttt{last}(\tau_q))$ we know that $S[\tau_c][i, n] = S[\tau_p][i, n]$ and $B[\tau_c][i, n] = B[\tau_p][i, n]$ and hence if P||Q takes the visible OUTPUT-CREATE-1 transition when in global con-

figuration $\texttt{last}(\tau_c k)$, the same transition can be taken by module P in configuration $\texttt{last}(\tau_p k')$.

$i \in \Sigma_Q$ and hence the environment of module Q creates an interface i (ENVIRONMENT-CREATE) and the resultant executions satisfy the condition that $\texttt{last}(\tau'_c) = \texttt{union}(\texttt{last}(\tau'_p), \texttt{last}(\tau'_q))$.

d) OUTPUT-CREATE-2
Similar analysis can be applied to prove that our inductive hypothesis holds when the composed module P||Q takes an OUTPUT-CREATE-2 transition.

■

Lemma 2.4.2

For every pair of executions $\tau_p \in \mathit{execs}(P)$ and $\tau_q \in \mathit{execs}(Q)$, if there exists $\sigma \in \Sigma^*_{P||Q}$ such that $\sigma[\Sigma_P] = \texttt{trace}(\tau_p)[\Sigma_P]$ and $\sigma[\Sigma_Q] = \texttt{trace}(\tau_q)[\Sigma_Q]$, then there exists an execution $\tau_c \in \mathit{execs}(P||Q)$ such that $\texttt{trace}(\tau_c)[\Sigma_{P||Q}] = \sigma$.

Proof.

Given a pair of executions (p, q) and (p', q'), we define a partial order over pair of executions as $(p, q) \preceq (p', q')$ iff p is a prefix of p' and q is a prefix of q'. We perform induction over the pair of executions of module P and Q using the partial order.

Inductive Hypothesis: For any pair of executions (τ_p, τ_q) of modules P and Q respectively, if there exists $\sigma \in \Sigma^*_{P||Q}$ such that $\sigma[\Sigma_P] = \texttt{trace}(\tau_p)[\Sigma_P]$ and $\sigma[\Sigma_Q] = \texttt{trace}(\tau_q)[\Sigma_Q]$ then there exists an execution $\tau_c \in \mathit{execs}(P||Q)$ such that $\texttt{trace}(\tau_c)[\Sigma_{P||Q}] = \sigma$ and $\texttt{last}(\tau_c) = \texttt{union}(\texttt{last}(\tau_p), \texttt{last}(\tau_q))$.

Base case: The inductive hypothesis hold trivially for the base case when the length of the executions τ_p, τ_q of modules P, Q is zero.

$\texttt{trace}(\tau_p)[\Sigma_P] = \texttt{trace}(\tau_q)[\Sigma_P] = \epsilon$ ($\epsilon \in \Sigma^*_{P||Q}$).

we know that: there exists $\tau_p = (S^p_0, B^p_0, C^p_0) \in \mathit{execs}(P)$, there exists $\tau_q = (S^q_0, B^q_0, C^q_0) \in \mathit{execs}(Q)$ and there exists $\tau_c = (S^c_0, B^c_0, C^c_0) \in \mathit{execs}(P||Q)$.

Hence, there exists an execution $\tau_c \in \mathit{execs}(P||Q)$ such that $\texttt{trace}(\tau_c)[\Sigma_{P||Q}] = \epsilon$

Finally, we have $\texttt{last}(\tau_c) = \texttt{union}(\texttt{last}(\tau_p), \texttt{last}(\tau_q))$ as:

- $S^c_0 = S^p_0 = S^q_0 = S_0$ (empty map)
- $B^c_0 = B^p_0 = B^q_0 = B_0$ (empty map)
- $C^c_0 = C^p_0 = C^q_0 = C_0$ (all elements map to 0)

Inductive case: Let us assume that the hypothesis holds for any pair of executions (τ_p, τ_q) and any σ. To prove that the hypothesis is inductive, we show that the hypothesis holds for the next pair of executions in the partial order $((\tau'_p, \tau_q), (\tau_p, \tau'_q)$ and (τ'_p, τ'_q) where $\tau'_p = \tau_p \xrightarrow{a} G'$, $\tau'_q = \tau_q \xrightarrow{a} G''$ and $\tau'_c = \tau_c \xrightarrow{a} G'''$). Just to provide an intuition, (τ'_p, τ_q) represents the case when module P takes a transition with label a and $a \notin \Sigma_Q$, similarly (τ_p, τ'_q) represents the case when module Q takes a transition with label a and $a \notin \Sigma_P$. (τ'_p, τ'_q) represents the case when module P and Q both take transition with label a, as $a \in \Sigma_P, a \in \Sigma_Q$.

We perform case analysis for all possible transitions taken by module P and module Q. We provide a proof for one such case:

1. Let us consider the case when module P takes a transition OUTPUT-SEND-1 with label $a = ((i_t, n_t), e, v)$. Let $(i, n) \in \mathtt{dom}(\mathtt{last}(\tau_P)_S)$ be the machine that takes this transition. Hence, $\sigma' = \sigma.a$ and $\mathtt{trace}(\tau'_p)[\Sigma_P] = \sigma'[\Sigma_P]$.

   Let us consider the case when $i_t \in \mathtt{dom}(I_Q)$, and $e \in MRecvs(i_t) \setminus (EP_Q \cup ES_Q)$ (input event of Q). Based on the assumption that $\mathtt{last}(\tau_c) = \mathtt{union}(\mathtt{last}(\tau_p), \mathtt{last}(\tau_q))$ and the invariants I1-I6 about the state configurations, we know that $(i_t, n_t) \in \mathtt{dom}(\mathtt{last}(\tau_q)_B)$.
   Hence, Q can take a INPUT-SEND transition with label $a = ((i_t, n_t), e, v)$ and therefore $\mathtt{trace}(\tau'_q)[\Sigma_Q] = \sigma'[\Sigma_Q]$.

   Finally, using same assumption $\mathtt{last}(\tau_c) = \mathtt{union}(\mathtt{last}(\tau_p), \mathtt{last}(\tau_q))$ and the invariants I1-I6, the composed module P||Q can take the transition OUTPUT-SEND-2 with the same label $a = ((i_t, n_t), e, v)$. Hence, $\mathtt{trace}(\tau'_c)[\Sigma_{P||Q}] = \sigma'$. The resultant executions still satisfy the condition that $\mathtt{last}(\tau'_c) = \mathtt{union}(\mathtt{last}(\tau'_p), \mathtt{last}(\tau'_q))$.

   Note that proving that executions of modules satisfy the property $\mathtt{last}(\tau_c) = \mathtt{union}(\mathtt{last}(\tau_p), \mathtt{last}(\tau_q))$ helps us prove a stronger property than what is needed for the lemma.

■

Lemma 2.4.3: Composition preserves refinement

Let P, Q, and R be three modules such that P,Q and R are composable. Then the following holds: (1) P||R $\preceq$ P and (2) P $\preceq$ Q implies that P||R $\preceq$ Q||R

Proof. (1) follows directly from the Theorem 2.4.1. For (2), let σ be a trace of P||R, then we know that $\sigma[\Sigma_P]$ is a trace of P and $\sigma[\Sigma_R]$ is a trace of R. We know that, P $\preceq$ Q therefore $\sigma[Q]$ is a trace of Q and using the Theorem 2.4.1 $\sigma[\Sigma_{Q||R}]$ is a trace of Q||R.
■

Lemma 2.4.4: Circular Assume-Guarantee

Let $\|\mathcal{P}$ and $\|\mathcal{Q}$ be well-formed. Suppose for each module $Q \in \mathcal{Q}$ there is a subset $\mathcal{X}$ of $\mathcal{P} \oplus \mathcal{Q}$ such that $Q \notin \mathcal{X}$, $\|\mathcal{X}$ is well-formed, and $\|\mathcal{X}$ refines Q. Then $\|\mathcal{P}$ refines each module $Q \in \mathcal{Q}$.

Proof. Definitions:

- Let $\mathcal{Q}$ be a collection of ($n > 1$) composable modules represented by the set $\{Q_1, Q_2, ...Q_n\}$.
- Let $\mathcal{P}$ be a collection of ($n' > 1$) composable modules represented by the set $\{P_1, P_2, ...P_{n'}\}$. In this proof, we refer to $\|\mathcal{P}$ (composition of all modules in $\mathcal{P}$) as module $\mathcal{P}$
- Let $\forall k.\mathcal{X}_k$ be a subset of $\mathcal{P} \oplus \mathcal{Q}$.

Let us assume that $\forall Q_k \in \mathcal{Q}$ there exists a $\mathcal{X}_k$ such that $\mathcal{X}_k \preceq Q_k$.

Inductive Hypothesis: Our inductive hypothesis is that for every execution $\tau_{\mathcal{P}} \in \mathit{execs}(\mathcal{P})$ and for all $Q_k \in \mathcal{Q}$, there exists an execution $\tau_{Q_k} \in \mathit{execs}(Q_k)$ such that $\mathtt{trace}(\tau_{\mathcal{P}})[\Sigma_{Q_k}] = \mathtt{trace}(\tau_{Q_k})[\Sigma_{Q_k}]$.
Note that the inductive hypothesis is over the executions of $\mathcal{P}$ but it implies that, if for all $Q_k \in \mathcal{Q}$, there exists a $\mathcal{X}_k$ such that $\mathcal{X}_k \preceq Q_k$ then for all traces $\sigma_{\mathcal{P}} \in \mathit{traces}(\mathcal{P})$ and for all $Q_k \in \mathcal{Q}$ we have $\sigma_{\mathcal{P}}[\Sigma_{Q_k}] \in \mathit{traces}(Q_k)$.

We prove our inductive hypothesis by performing induction over the length of execution $\tau_{\mathcal{P}}$.

1. **Base case:** The base case is one where the length of execution $\tau_{\mathcal{P}}$ is 0. The inductive hypothesis trivially holds for the base case.
2. **Inductive case:** Let us assume that the inductive hypothesis holds for any execution $\tau_p \in \mathit{execs}(\mathcal{P})$ of length k. To prove that the hypothesis is inductive, we show that the hypothesis also holds for any execution τ'_p where $\tau'_p = \tau_p \xrightarrow{a} G$.

   We have to perform the case analysis for all possible transition labels a. We provide a proof for some of these cases:

   a) $a = \epsilon$ (Invisible transition)

      It can be easily seen that the inductive hypothesis holds for the case when the module $\mathcal{P}$ takes an invisible transition.

   b) $a = i$ where $i \in \mathcal{I}$ (creation of an interface)
      a can be equal to i because of any of the following cases: (1) module $\mathcal{P}$ creates an interface using the transitions: OUTPUT-CREATE-1, OUTPUT-CREATE-2 or

(2) the environment creates it using the transitions: ENVIRONMENT-CREATE, INPUT-CREATE.

Let us consider the case when $a = i$ because $\mathcal{P}$ executes the OUTPUT-CREATE-1 transition.

Recollect that $\mathcal{P}$ is a composition of modules $P_1, P_2, ..P_{n'}$. Using Lemma 2.4.1, we can decompose the execution $\tau_{\mathcal{P}}$ of module $\mathcal{P}$ ($\tau_{\mathcal{P}} \in \mathit{execs}(\mathcal{P})$) into the executions $\tau_{P_1}, \tau_{P_2}, ...$ of the component modules such that for all $P_k \in \mathcal{P}$, $\mathtt{trace}(\tau_{\mathcal{P}})[\Sigma_{P_k}] = \mathtt{trace}(\tau_{P_k})[\Sigma_{P_k}]$.

From the operational semantics of OUTPUT-CREATE-1, we know that $i \in \mathit{IC}_{\mathcal{P}}$ and $i \notin \mathtt{dom}(I_{\mathcal{P}})$. Let us consider the case when there exists a module $P_k \in \mathcal{P}$ such that $i \in \mathit{IC}_{P_k}$, and from the definition of composition we know that $\forall j, j \neq k, i \notin \mathit{IC}_{P_j}$.

If $\exists j$, s.t. $j \neq k \wedge i \in \Sigma_{P_j}$ then P_j can take the ENVIRONMENT-CREATE transition to match the visible action $a = i$.

If $i \in \mathit{IC}_{\mathcal{Q}}$, then for some $Q_k \in \mathcal{Q}$, $i \in \mathit{IC}_{Q_k}$ – (1).

If $\forall Q_k \in \mathcal{Q}_k, i \notin \mathit{IC}_{\mathcal{Q}}$, then all Q_k can take the ENVIRONMENT-CREATE transition to match the visible action $a = i$.

Let us consider the case when only (1) is true. Since $i \in \mathit{IC}_{Q_k}$ and $\mathcal{X}_k \preceq Q_k$ we have $i \in \mathit{IC}_{\mathcal{X}_k}$.

Note that $\mathcal{P}$ and $\mathcal{Q}$ are well formed modules. Since (1) $Q_k \notin \mathcal{X}_k$ (2) $\forall j, j \neq k.i \notin \mathit{IC}_{P_j} \wedge i \notin \mathit{IC}_{Q_j}$, we know that $P_k \in \mathcal{X}_k$.

Using Lemma 2.4.2, and the fact that $\mathcal{X}_k \preceq Q_k$, we know that for any given $\tau'_{P_k} \in \mathit{execs}(P_k)$ there exist τ'_{Q_k} such that $\mathtt{trace}(\tau'_{P_k})[\Sigma_{Q_k}] = \mathtt{trace}(\tau'_{Q_k})[\Sigma_{Q_k}]$.

Finally, we know that:

i. Inductive hypothesis holds for any execution $\tau_{\mathcal{P}}$ and $\tau'_{\mathcal{P}} = \tau_{\mathcal{P}} \xrightarrow{i} G$ (Output-Create-1)

ii. $i \in \mathit{IC}_{Q_k}$ and $i \in \mathit{IC}_{P_k}$.

iii. $\forall j, j \neq k.i \notin \mathit{IC}_{P_j}$ and $\forall j, j \neq k.i \notin \mathit{IC}_{Q_j}$.

iv. there exists an execution $\tau'_{P_k} \in \mathit{execs}(P_k)$ such that $\mathtt{trace}(\tau'_{\mathcal{P}})[\Sigma_{P_k}] = \mathtt{trace}(\tau'_{P_k})[\Sigma_{P_k}]$.

v. there exists an execution $\tau'_{Q_k} \in \mathit{execs}(Q_k)$ such that $\mathtt{trace}(\tau'_{P_k})[\Sigma_{Q_k}] = \mathtt{trace}(\tau'_{Q_k})[\Sigma_{Q_k}]$

Hence, we can conclude that for the execution $\tau'_{\mathcal{P}}$ there exists an execution τ'_{Q_k} such that $\mathtt{trace}(\tau'_{\mathcal{P}})[\Sigma_{Q_k}] = \mathtt{trace}(\tau'_{Q_k})[\Sigma_{Q_k}]$.

And using (3), we also know that for all $Q_j \in \mathcal{Q}_k$, $\texttt{trace}(\tau'_{\mathcal{P}})[\Sigma_{Q_j}] = \texttt{trace}(\tau_{Q_k})[\Sigma_{Q_j}]$

Hence, the inductive hypothesis holds for the execution $\tau'_{\mathcal{P}}$.

We do similar analysis to prove the other cases.

■

Lemma 2.4.5: Compositional Safety Analysis

Let $\|\mathcal{P}$ and $\|\mathcal{Q}$ be well-formed. Let $\|\mathcal{P}$ refine each module $Q \in \mathcal{Q}$. Suppose for each $P \in \mathcal{P}$, there is a subset $\mathcal{X}$ of $\mathcal{P} \oplus \mathcal{Q}$ such that $P \in \mathcal{X}$, $\|\mathcal{X}$ is well-formed, and $\|\mathcal{X}$ is safe. Then $\|\mathcal{P}$ is safe.

Proof.

We describe a proof strategy using contradiction for a simplified system consisting of two implementation modules P_1, P_2 and two abstraction modules Q_1, Q_2. For such a system, the theorem states that if $P_1 \parallel P_2 \preceq Q_1$, $P_1 \parallel P_2 \preceq Q_2$ and $P_1 \parallel Q_2$, $Q_1 \parallel P_2$ are safe then $P_1 \parallel P_2$ is safe.

Lets say that there exists an error execution in τ^e in $P_1 \parallel P_2$. Using the compositional refinement Lemma, we can decompose the execution τ^e into τ^e_1 of P_1 and τ^e_2 of P_2. Lets say the error was because of module P_1 taking a transition and hence τ^e_1 is an error trace.

We know that $P_1 \parallel Q_2$ is safe which means that for all executions of module $P_1 \parallel Q_2$ there is no execution of P_1 that is equal to τ^e_1 after decomposition.

The above condition also implies that in the composed module $P_1 \parallel P_2$, module P_2 using an output action is triggering an execution in P_1 which results in execution τ^e_1. And this output action is not triggered by Q_2 in the composition $P_1 \parallel Q_2$.

The above condition implies that $P_1 \parallel P_2 \preceq Q_2$ does not hold which is a contradiction.

We generalized this proof strategy for proving the given lemma.

■

Lemma 2.4.6: Hide Event Preserves Refinement

For all well-formed modules P and Q and a set of events α, if $(\textbf{hide}_{\alpha}\textbf{ in }P)$ and $(\textbf{hide}_{\alpha}\textbf{ in }Q)$ are well-formed, then (1) $P \preceq (\textbf{hide}_{\alpha}\textbf{ in }P)$ and (2) if $P \preceq Q$, then $(\textbf{hide}_{\alpha}\textbf{ in }P) \preceq (\textbf{hide}_{\alpha}\textbf{ in }Q)$.

Proof. Let $hP = (\textbf{hide}_{\alpha}\textbf{ in }P)$ and $hQ = (\textbf{hide}_{\alpha}\textbf{ in }Q)$.

We perform induction over the length of execution τ_{hP} of module hP

Inductive Hypothesis: For every execution $\tau_p \in \mathit{execs}(\hbar P)$, there exists an execution $\tau_p \in \mathit{execs}(\hbar Q)$ such that $\mathtt{trace}(\tau_{\hbar P})[\Sigma_{\hbar Q}] = \mathtt{trace}(\tau_{\hbar Q})[\Sigma_{\hbar Q}]$

We prove our inductive hypothesis by performing induction over the length of execution $\tau_{\mathcal{P}}$.

- **Base case:** The base case is trivially satisfied by an execution of length zero.
- **Inductive case:** Let us assume that the hypothesis holds for any execution $\tau_{\hbar P} \in \mathit{execs}(\hbar P)$ and the corresponding execution of module $\hbar Q$ be $\tau_{\hbar Q} \in \mathit{execs}(\hbar Q)$.

  To prove that the hypothesis is inductive we show that it also holds for the execution $\tau'_{\hbar P} \in \mathit{execs}(\hbar P)$ where $\tau'_{\hbar P} = \tau_{\hbar P} \xrightarrow{a} G$ and $\tau'_{\hbar Q}$ be the resultant executions of $\hbar Q$.

  Hide operation only converts visible actions into internal actions. Hence, it can be easily shown that any execution of $\hbar P$ is also an execution of P, similarly for module $\hbar Q$ and Q, every execution of $\hbar Q$ is an execution of Q.

  The above property, along with the fact that $P \preceq Q$ helps us conclude that the inductive hypothesis always holds.

■

Lemma 2.4.7: Hide Interface Preserves Refinement

For all well-formed modules P and Q and a set of interfaces α, if $(\mathbf{hide}_{\,,}\ \mathbf{in}\ P)$ and $(\mathbf{hide}_{\,,}\ \mathbf{in}\ Q)$ are well-formed, then (1) $P \preceq (\mathbf{hide}_{\,,}\ \mathbf{in}\ P)$ and (2) if $P \preceq Q$, then $(\mathbf{hide}_{\,,}\ \mathbf{in}\ P) \preceq (\mathbf{hide}_{\,,}\ \mathbf{in}\ Q)$.

Proof. The proof is similar to the proof for Lemma 2.4.6. ■

2.5 RELATED WORK

Assume-Guarantee reasoning has been implemented in model checkers [11, 133, 134] and successfully used for hardware verification [69, 99, 132] and software testing [26]. However, the present paper is the first to apply it to distributed systems of considerable complexity and dynamic behavior. We next situate MODP with related techniques for modeling and analysis of distributed systems.

Formalisms and programming models. We categorize the formalisms for the modeling and compositional analysis of dynamic systems into three foundational approaches: process algebras, reactive modules [9], and I/O automata [128].

(1) Process algebra. In the process algebra approach deriving from Hoare's CSP [102] and Milner's CCS [137], the π-calculus [138, 158] has become the de facto standard in modeling mobility and reconfigurability for applications with message-based communication. The popular approach to reasoning about behavior in these formalisms is the notions of equivalence and congruence: weak and strong bisimulation, which involves examining the state transition structure of the two systems. There's also extensive literature on observational equivalence in π-calculus based on trace inclusion [43]. Extensions of π-calculus such as asynchronous π-calculus, distributed join calculus [79, 80], Dπ-calculus [166] deal with distributed systems challenges like asynchrony and failures respectively. MODP chooses *Actors* [5] as its model of computation, and our theory of compositional refinement uses trace inclusion based only on the externally visible behavior as it dramatically simplifies our *refinement testing* framework. In MODP, abstractions (modules) are state machines capable of expressing arbitrary trace properties. More recent work like session types [35, 60, 105] and behavioral-types [12] that have their roots in process calculi can encode abstractions in the type language (e.g., [28]).

(2) Reactive modules. Reactive modules [9] is a modeling language for concurrent systems. Modules communicate via single-writer multiple-reader shared variables and a global clock drives each module in lockstep. Dynamic Reactive Modules [77] (DRM) is an extension of Reactive Modules with support for the dynamic creation of modules and dynamic topology. Dynamic discrete systems [77] gives the semantics of dynamic reactive modules to model the creation of module instances and the refinement relation between dynamic reactive modules is defined using a specialized notion of transition system refinement. DRM does not formalize a compositionality theorem for the hide operation. Also, our module system is novel compared to DRM because of the fundamental differences in the supported programming model.

(3) I/O automata. Dynamic I/O automata (DIOA) [18] is a compositional model of dynamic systems, based on I/O automata [128]. DIOA is primarily a (set-theoretic) mathematical model, rather than a programming language or calculus. Our notion of parallel composition, trace monotonicity, and trace inclusion based on externally visible actions is inspired from DIOA and is formalized for the compositional reasoning of actor programs. MODP incorporates these ideas into a practical programming framework for building distributed systems.

Verification of distributed systems. There has been a lot of work towards reasoning about concurrent systems using program logics deriving from Hoare logic [78, 101] – which includes rely-guarantee reasoning [83, 191, 203] and concurrent separation logic [74, 123, 149]. Actor services [185] propose program logic for modular proofs of *actor* programs. DISEL [177] provides a language to implement and verify distributed systems compositionally. The goal of these techniques is similar to ours, enable compositional reasoning; they decompose reasoning along the syntactic structure of the program and emphasize modularity principles that allow proofs to be easily

constructed, maintained and reused. They require fine-grained specifications at the level of event-handler, in our case programmer writes specifications for components as abstractions. The focus on compositional testing instead of proof allows us to attach an abstraction to an entire protocol rather than individual actions within that protocol (e.g., Send-hooks in DISEL), thereby reducing the annotations required for validation. The goal of this paper is to scale automated testing to large distributed services and to achieve this goal we develop a theory of assume-guarantee reasoning for actor programs.

Many recent efforts like IronFleet [96], Verdi [200], and Ivy [152] have produced impressive proofs of correctness for the distributed system, but the techniques in these efforts do not naturally allow for horizontal composition. McMillan [135] extended Ivy with a specification idiom based on reference objects and circular assume-guarantee reasoning to perform modular verification of a cache-coherence protocol.

Systematic testing of distributed systems. Researchers have built testing tools [121, 175] for automated unit testing of Java actor programs. Mace [112], TeaPot [36] and P [53] provide language support for implementation, specification and systematic testing of asynchronous systems. MaceMC [111] and MoDist [205] operate directly on the implementation of a distributed system and explore the space of executions to detect bugs in distributed systems. DistAlgo [127] supports asynchronous communication model, similar to ours, and allows extraction of efficient distributed systems implementation from the high-level specification. None of these programming frameworks tackle the challenges of compositional testing addressed in this paper. The conclusion of most of the researchers who developed these systems is similar to ours: monolithic testing of distributed systems does not scale [90].

McCaffrey's article [131] provides an excellent summary of the approaches used in the industry for systematic testing of distributed systems. *Manual-targeted testing* is an effective technique where an expert programmer provides manually crafted test-cases for finding critical bugs. However, it requires considerable expertise and manual effort. MODP's focus is on scaling automated testing and hence do not consider manual-target testing as a baseline for comparison. *Property-based testing* is another popular approach in industry for the semi-automatic testing of distributed systems (e.g., QuickCheck) [14, 108]). MODP's compositional testing approach, as well as the monolithic testing method we compare it to, can both be viewed as property-based testing since they assert the safety properties specified as monitors given a non-deterministic test harness. The compositional testing methodology described in this paper is orthogonal to the technique used for analyzing the test declarations; other approaches such as manual-targeted or property-based testing can also be used for discharging the test declarations.

3

BUILDING DISTRIBUTED SYSTEMS COMPOSITIONALLY

> *the fault-tolerant distributed computing community has not developed the tools and know-how to close the gaps between theory and practice .. these gaps are non-trivial and they merit attention by the research community.*
>
> — *Chandra et al., in "Paxos Made Live - An Engineering Perspective"*

In Chapter 2, we introduced a programming framework, MODP, that leverages a new theory of compositional refinement for modular programming and scalable systematic-testing of distributed systems. In this chapter, we present how this theory of compositional reasoning can be applied in practice to build reliable distributed systems. Using MODP, we build two fault-tolerant distributed services; we present an empirical evaluation of the compositional systematic testing and runtime performance of these distributed services that combine 7 different protocols.

3.1 FROM THEORY TO PRACTICE

Theorems 2.4.3 and 2.4.4 indicate that there are two kinds of obligations that result from assume-guarantee reasoning—*safety* and *refinement*. Although these obligations can be verified using proof techniques, the focus of MODP is to use systematic testing to falsify them. MODP allows the programmer to write each obligation as a test declaration. The declaration `test tname: P` introduces a safety test obligation that the executions of module `P` do not result in a failure (module `P` is safe). The declaration `test tname: P refines Q` introduces a test obligation that module `P` refines module `Q`. These test obligations are automatically discharged using MODP's systematic testing engine (Section 3.2).

CASE STUDY: FAULT TOLERANT DISTRIBUTED SERVICES. Figure 3.1 shows two large distributed services that are representative of challenges in real-world distributed systems: (i) atomic commit of updates to decentralized, partitioned data using two-phase commit [88], and (ii) replicated data structures such as hash-tables and lists. These distributed services use State Machine Replication (SMR) for fault-tolerance [172]. Protocols for SMR, such as Multi-Paxos [118] and Chain-Replication [165], in turn use other protocols like leader election and fault detectors. To evaluate MODP, we implemented each sub-protocol (diagonal lines) as a separate module and performed compositional reasoning at each layer of the protocol stack. We also compare the performance of the hash-table distributed service against its open-source counterpart by benchmarking it on a cluster.

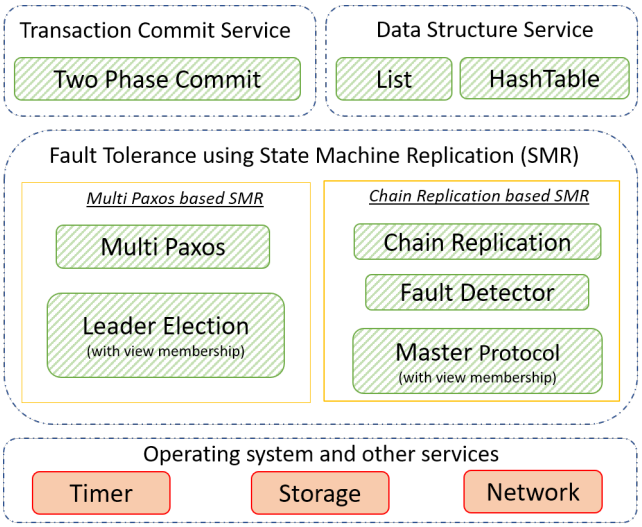


Figure 3.1: Fault-Tolerant Distributed Services

We illustrate using the protocol stack in Figure 3.1, how we used MODP to implement and test a complex distributed system compositionally. We implement distributed transaction commit using the two-phase commit protocol, which uses a single *coordinator* state machine to atomically commit updates across multiple *participant* state machines. Hashtable and list are implemented as deterministic state machines with `PUT` and `GET` operations. These services by themselves are not tolerant to node failures. We use SMR to make the two-phase commit and the data structures fault-tolerant by replicating the deterministic coordinator, participant, and hash-table (list) state-machines across multiple nodes. We implemented Multi-Paxos [118] and Chain-Replication [165] based SMR, these protocols guarantee that a consistent sequence of events is fed to the deterministic (replicated) state machines running on multiple nodes. These events could be operations on a data-structure or operations for two-phase-commit. Multi-Paxos and Chain-Replication, in turn, use different sub-protocols. Though both these protocols provide linearizability guarantees their implementations are very different

with distinct fault models and hence acts as an excellent case study for module (protocol) substitution. For example, Multi-Paxos uses $2n+1$ replicas to tolerate n failures whereas Chain Replication exploits a reliable failure detector to use only $n+1$ replicas for tolerating n failures. The protocols in the software stack use various OS services like timers, network channels, and storage services which are not implemented in MODP. We provide *over approximating models* for these libraries in MODP which are used during testing but replaced with the library, and OS calls for real execution.

COMPOSITIONALLY TESTING TRANSACTION-COMMIT SERVICE. The MODP approach would be to test each of the sub-protocol in isolation using abstractions of the other protocols. For example, when testing the two-phase commit protocol, we replace the Multi-Paxos based SMR implementation with its single process linearizability abstraction. Our evaluation demonstrates that such abstraction based decomposition provides orders of magnitude test-coverage amplification compared to monolithic testing. Further, our approach for checking refinement through testing is effective in finding errors in module abstractions, thus, helping ensure soundness. We checked the safety specifications (as spec. machines) of all the protocols as described in their respective paper. The table below shows examples of specifications checked for some of the distributed protocols.

Protocol	Specifications
2PC	Transactions are atomic [87] (**2PCSpec**)
Chain Repl.	All invariants in [165], cmd-log consistency (**CRSpec**)
Multi-Paxos	Consensus requirements [119], log consistency [193] (**MPSpec**)

Figure 3.2: Specifications checked for each protocol

Listing 3.1 presents a simplified version of the test-script used for compositionally testing the transaction-commit service. The modules `2PC`, `MultiPaxosSMR`, `ChainRepSMR` represent the implementations of the two-phase commit, Multi-Paxos based SMR, and Chain-Replication based SMR protocols respectively. The module `SMRLinearizAbs` represent the linearizability abstraction of the SMR service, both Multi-Paxos based SMR and Chain-Replication based SMR provide this abstraction. The module `SMRClientAbs` represent the abstraction of any client of the SMR service. `OSServAbs` implements the models for mocking OS services like timers, network channels, and storage. A failure injector machine that randomly halts machines in the program is also added as part of the `OSServAbs`. There are two sets of implementation modules $\mathcal{P}_m$ ={`2PC`, `MultiPaxosSMR`, `OSServAbs`} or $\mathcal{P}_c$ ={`2PC`, `ChainRepSMR`, `OSServAbs`} representing the Multi-Paxos and Chain-Replication based versions. The set of abstraction modules is $\mathcal{Q}$ ={`SMRClientAbs`, `SMRLinearizAbs`, `OSServAbs`}. The test obligation `mono` represents the monolithic testing problem for transaction-commit service.

```
1  //monolithic testing of software stack
2  test mono: (assert 2PCSpec in 2PC) || MultiPaxosSMR ||
       OSServAbs;
3
4  //Decomposition using compositional safety
5  test t1: (assert 2PCSpec in 2PC) || SMRLinearizAbs ||
       OSServAbs;
6  test t2: SMRClientAbs || MultiPaxosSMR || OSServAbs;
7  test t3: SMRClientAbs || MultiPaxosSMR || OSServAbs
8           refines SMRClientAbs || SMRLinearizAbs || OSServAbs;
9  test t4: 2PC || SMRLinearizAbs || OSServAbs
10          refines SMRClientAbs || SMRLinearizAbs || OSServAbs;
11 //Multi Paxos linearizability as specification machine
12 test t5: SMRClientAbs || assert MPSec in MultiPaxosSMR ||
       OSServAbs;
13
14 //test chain replication SMR
15 test t6: SMRClientAbs || ChainRepSMR || OSServAbs
16 test t7: SMRClientAbs || ChainRepSMR || OSServAbs
17          refines SMRClientAbs || SMRLinearizAbs || OSServAbs;
18 //Chain replication linearizability as specification machine
19 test t8: SMRClientAbs || assert CRSpec in ChainRepSMR ||
       OSServAbs;
20
21 //test 7
22 module LHS = ChainRepSMR || SMRClientAbs || TestDriver ||
       OSServAbs;
23 module RHS =
24     // hide replicated machine creation operation
25     (hidei SMRReplicatedMachineInterface in
26     //hide events used for interaction with replicated
          machine
27     (hidee eSMRReplicatedMachineOperation,
          eSMRReplicatedLeader in
28       SMRClientAbs || TestDriver || SMRReplicated ||
            OSServAbs));
29 test t7:  LHS refines RHS;
```

Listing 3.1: Compositional Testing of Transaction Commit Service

Similar to *property-based testing* [14], the programmer can attach specifications to modules under test using the `assert` constructor (e.g., Listing 3.1-line 5). Using Theorem 2.4.3, we can decompose the monolithic problem into safety tests `t1` and `t2` under the assumption that each module in $\mathcal{P}_m$ refines each module in $\mathcal{Q}$. This as-

sumption is then validated using the Theorem 2.4.4 and tests `t3`, `t4`. The power of compositional reasoning is substitutability; if the programmer wants to migrate the transaction commit service from using Multi-Paxos to use Chain-Replication then he just needs to validate `ChainRepSMR` in isolation using tests `t6` and `t7`. The tests `t5` and `t8` are substitutes for the refinement checks `t4` and`t7` since the spec. machines (from the table) assert the linearizability abstraction of these protocols.

The test declarations used in practice are a bit more involved than Listing 3.1. There are two main points: (1) For each test declaration, the programmer provides a finite test harness module comprising non-deterministic machines that close the module under test by either supplying inputs or injecting failures. The programmer may provide a collection of test harnesses modules for each test declaration to cover various testing scenarios for each test obligation. (2) In some cases, the module constructors like hide and rename have to be used to make modules composable or create the right projection relation. Listing 3.1 (line 22-29) represent the test-script we used to perform test `t7`. We had to hide internal events sent to the replicated machine to create the right projection relation for refinement.

3.2 IMPLEMENTATION OF THE MODP TOOL CHAIN

In this section, we describe the implementation of the MODP toolchain (Figure 3.3). The MODP toolchain is available as part of the P programming framework (`https://github.com/p-org/P`).

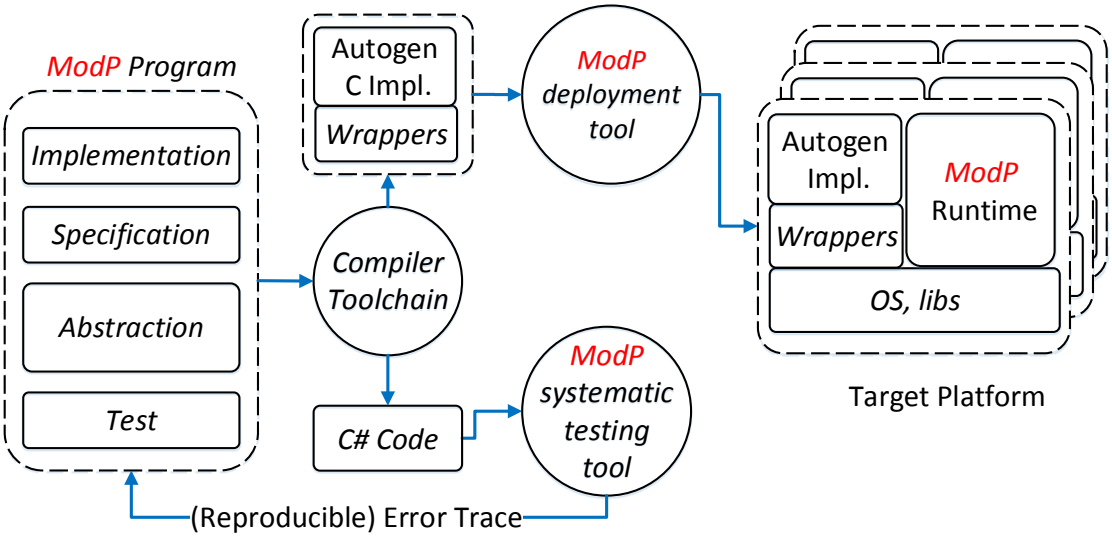


Figure 3.3: MODP Programming Framework

Compiler. A MODP program comprises four blocks — implementation modules, specifications monitors, abstraction modules and tests. The compiler static-analysis of the source code not only performs the usual type-correctness checks on the code of machines but also checks that constructed modules are well-formed, and test declarations are legal. The compiler generates code for each test declaration; this generated code makes all sources of nondeterminism explicit and controllable by the systematic testing engine, which generates executions in the test program checking each execution against implicit and explicit specifications. *For each test declaration, the compiler generates a standalone program that can be independently analyzed by the back-end systematic testing engine.* The compiler also generates C code which is compiled and linked against the MODP runtime to generate application executables.

Systematic testing engine. The MODP systematic testing engine efficiently enumerates executions resulting from scheduling and explicit nondeterministic choices. The MODP compiler generates a standalone program for each safety test declaration. We reuse the existing P testing backends for safety test declarations with modifications to take into account the extensions to P state machines. There are two backends provided by P: **(1)** a sampling-based testing engine that explicitly sample executions using delay-bounding based prioritization (Chapter 4), and **(2)** a symbolic execution engine with efficient state-merging using MultiSE [176, 204].

We extended the sampling based testing engine to perform refinement testing of MODP programs based on trace containment. Our algorithm for checking $P \preceq Q$ consists of two phases: **(1)** In the first phase, the testing engine generates all possible visible traces of the abstraction module Q and compactly caches them in memory. The abstraction modules are generally small, and hence, all the traces of Q can be loaded in memory for all our experiments. **(2)** In the second phase, the testing engine performs stratified sampling of the executions in P, and for each terminating execution checks if the visible trace is contained in the cache (traces of Q). A safety bug is reported as a sequence of visible actions that lead to an error state. In the case of refinement checking, the tool returns a visible trace in implementation that is not contained in the abstraction.

Distributed runtime. Figure 3.4 shows the structure of a MODP application executing on distributed nodes. We believe that the *multi-container* runtime is a generic architecture for executing programs with distributed state-machines. Each node hosts a collection of *Container* processes. *Container* is a way of grouping collection of MODP state machines that interact closely with each other and must reside in a common fault domain. Each Container process hosts a *listener*, whose job is to forward events received from other containers to the state machines within the container. State machines within a container are executed concurrently using a thread pool and as an optimization interacts without serializing/deserializing the messages.

Each node runs a *NodeManager* process which listens for requests to create new Container processes. Similarly, each Container hosts a single *ContainerManager* that

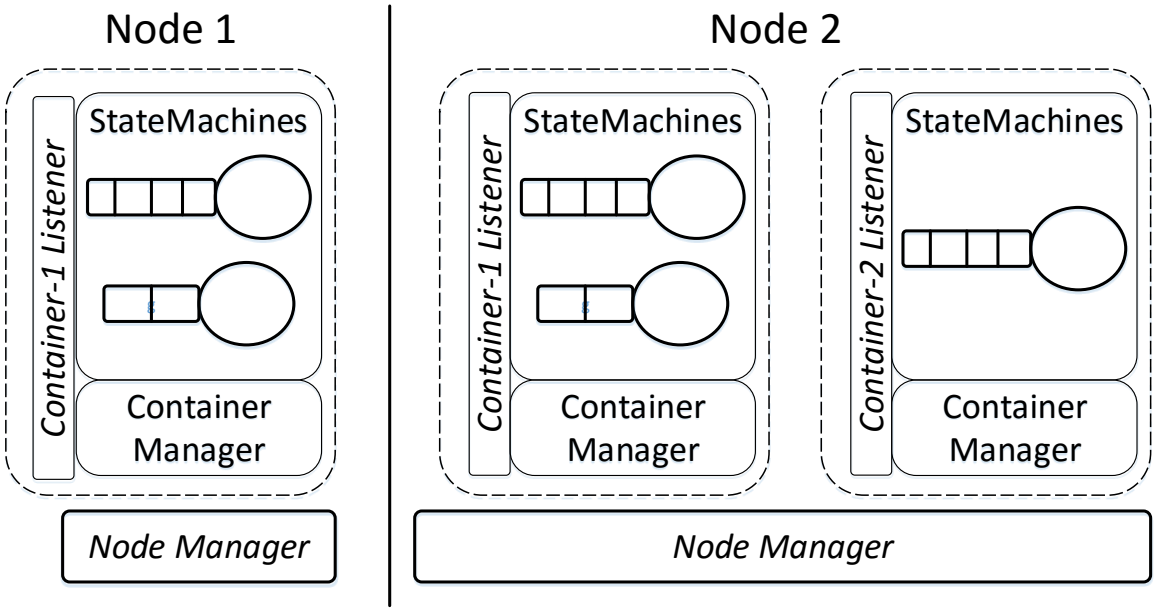


Figure 3.4: Structure of MODP application

services requests for creations of new state machines within the container. In the typical case, each node has one NodeManager process and one Container process executing on it, but MODP also supports a collection of Containers per node enabling emulation of large-scale services running on only a handful of nodes. A MODP state machine can create a new container by invoking runtime's `CreateContainer` function. A state machine can create a *new* local or remote state machine by specifying the hosting container's ID. Hence, the MODP runtime enables the programmer to distribute state-machines across distributed nodes and also group them within containers for optimizing the performance.

In summary, the runtime executes the generated C representation of the MODP program and has the capability to (1) create, destroy, and execute distributed state machines, (2) efficiently communicate among state machines that can be distributed across physical nodes, (3) serialize data values before sends and deserialize them after receives.

3.3 EVALUATION

We empirically evaluate MODP framework by compositionally implementing and testing the fault-tolerant distributed services software stack (Figure 3.1). The goal of our evaluation is twofold:

(Goal 1) Demonstrate that the theory of compositional refinement helps scale systematic testing to complex large distributed systems. We show that compositional testing leads

Protocol	Impl.	Specs.	Abst.	Test Driver	Test Decls
2 Phase Commit	441	61	41	35	128
Chain Rep. SMR	1267	220	173	130	105
Multi-Paxos SMR	1617	101	121	92	90
Data structures	276	25	-	89	25
Total	**3601**	**Others = 1436**			

Figure 3.5: Source lines of MODP code

to test-amplification in terms of both: increasing the test-coverage and finding more bugs (faster) than the monolithic testing approach (Section 3.3.2). We present anecdotal evidence of the benefits of refinement testing. It helps find bugs that would have been missed otherwise when performing abstraction-based compositional testing.
(Goal 2) Demonstrate that the performance of the (rigorously tested) distributed services built using MODP is comparable to the corresponding open-source baseline. We evaluate the performance of the hash-table distributed service by benchmarking it on Azure cluster (Section 3.3.3).

3.3.1 *Programmer Effort*

The Table below shows a five-part breakdown, in source lines of MODP code, of our implementation of the distributed service. The Impl. column represents the detailed implementation of each module whose – generated C code can be deployed on the target platform. Specs. column represents the component-level temporal properties (monitors). Abst. column represents abstractions of the modules used when testing other modules. The Driver column represents the different finite test-harnesses written for testing each protocol in isolation. The last column represents the test declarations across protocols to compositionally validate the "whole-system" level properties as described in Section 2.4.1.

3.3.2 *Compositional Testing*

The goal of our evaluation is to demonstrate the benefits of using the theory of compositional refinement in testing distributed systems, and hence, we use the same backend engine (Section 3.2) for testing both the monolithic test declaration and the corresponding compositional test declarations. We use the existing systematic testing engine of P that supports state-of-the-art search prioritization (Chapter 4) and other efficient bug-finding techniques for analyzing the test declarations. Note that the

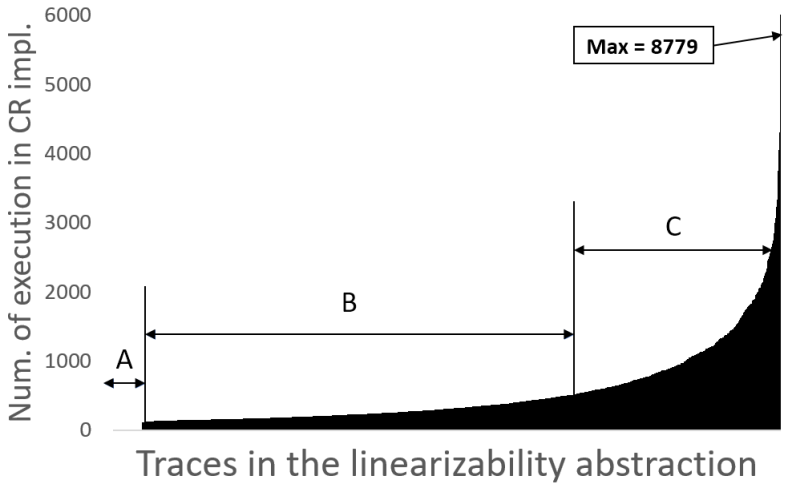


Figure 3.6: Test-Amplification via Abstractions: Chain-Replication Protocol

approach used for analyzing the test declarations is orthogonal to the benefits of using compositional testing.

Compositional reasoning led to the state-space reduction and hence amplification of the test-coverage, uncovering 20 critical bugs in our implementation of the software stack. To highlight the benefits of using MoDP-based compositional reasoning, we present two results in the context of our case-study: (1) abstractions help amplify the test-coverage for both the testing backends, the prioritized execution sampling and symbolic execution (Section 3.2), and (2) this test-coverage amplification results in finding bugs faster than the monolithic approach. For monolithic testing, we test the module constructed by composing the implementation modules of all the components.

Test-amplification via abstractions. Using abstractions simplifies the testing problem by reducing the state-space. The reduction is obtained because a large number of executions in the implementations can be represented by an exponentially small number of abstraction traces.

To show the kind of amplification obtained for the sampling based testing approach, we conducted an experiment to count the number of unique executions in the implementation of a protocol that maps to a trace in its abstraction. Figure 3.6 present the graph for the Chain-Replication (CR) protocol with a finite test-harness that randomly pumps in 5 update operations. The `x-axis` represents the traces in the abstraction sorted by `y-axis` values, where the `y-axis` represents the number of executions in the implementation that maps (projects) to the trace in abstraction. The linearizability abstraction (guaranteed by Chain-Replication protocol) has 1931 traces for the finite test-harness, and there were exponentially many executions in the CR implementation. We sampled 10^6 unique executions in the CR implementation for this experiment.

The graph in Figure 3.6 is highly skewed and can be divided into three regions of interest: region (A) correspond to those traces in the abstraction to which no execution

Protocol	Schedules Explored	
	Monolithic	CST
MPaxos (bug1)	13	**11**
2PC (bug2)	1944	**19**
ChainR (bug3)	2018	**13**
MPaxos (bug4)	NF	**91**
T2PC (bug5)	NF	**112**
ChainR (bug6)	NF	**187**
ChainR (bug7)	NF	**782**
MPaxos (bug8)	NF	**2176**

Figure 3.7: CST vs. Monolithic Testing. (NF: Bug not found)

mapped from the samples set of 10^6 implementation executions which could be either because these traces correspond to a very low probability execution in implementation or are false positives; region (B) represent those traces that correspond to low probability executions in the implementation; region (C) represent those executions that may lead to a lot of redundant explorations during monolithic testing. Using linearizability abstraction helps in mitigating this skewness and hence increases the probability of exploring low probability behaviors in the system leading to amplification of test-coverage (as in some cases exploring one execution in the abstraction is equivalent to exploring approx. 8779 executions in the implementation).

Next, we show that the compositional testing approach helps the sampling based back-end to find bugs faster. We randomly chose 8 bugs (out of 20) that we found in different protocols during the development process. We compared the performance of compositional testing (CST) against the monolithic testing approach where the entire protocol stack is composed together and considered as a single monolithic system. We use the number of schedules explored before finding the bug as the comparison metric. Figure 3.7 shows that MoDP-based compositional approach helps the sampling based back-end find bugs faster than the monolithic approach and in most cases, the monolithic approach fails to find the bug even after exploring 10^6 different schedules.

P also supports a symbolic execution back-end that uses the MultiSE [176, 204] based approach for state-merging. To evaluate the test amplification obtained for the symbolic execution back-end, we compared the performance of the testing engine for the monolithic testing problem and its decompositions from Listing 3.1. We performed the test `mono` using the symbolic engine for a finite test-harness where the `2PC` performs 5 transactions. The symbolic engine could not explore all possible execution of the problem even after 10 hrs. We performed the tests `t1, t2, t5, t8` (for the same finite test-harness) and the symbolic engine was able to explore all possible executions for

each decomposed test in 1.3 hours (total). The upshot of our module system is that we can get complete test-coverage (guaranteeing absence of bugs) for a finite test-harness which was not possible when doing monolithic testing.

We describe a few of these bugs in detail to illustrate the diversity of bugs found in practice.

1. ChainR (bug7) represents a consistency bug that violates the update propagation invariant in [165]. The bug was in the chain repair logic and can be reproduced only when an intermediate node in the chain that has uncommitted operations, first becomes a tail node because of tail failure and then a head node on the head failure. This specific scenario could not be uncovered using monolithic testing but is triggered when testing the Chain-Replication protocol in isolation because of the state-space reduction obtained using abstractions.

2. MPaxos (bug4) represents a bug in our acceptor logic implementation that violates the P2c invariant in [119]. For this bug to manifest, it requires multiple leaders (proposers) in the Multi-Paxos system to make a decision based on an incorrect promise from the acceptor. In a monolithic system, because of the explosion of non-deterministic choices possible the probability of triggering a failure that leads to choosing multiple leaders is extremely low. When compositionally testing Multi-Paxos, we compose it with a coarse-grained abstraction of the leader election protocol. The abstraction non-deterministically chooses any Multi-Paxos node as a leader and hence, increasing the probability of triggering a behavior with multiple leaders.

3. Meaningful testing requires that the abstractions used during compositional reasoning are sound abstractions of the components being replaced. We were able to uncover scenarios where bugs could have been missed during testing because of an unsound abstraction. The linearizability abstraction was used when testing the distributed services built on top of SMR. Our implementation of the abstraction guaranteed that for every request there is a single response. For Chain-Replication protocol (as described in [165]), in a rare scenario when the tail node of the system fails and after the system has recovered, there is a possibility that a request may be responded multiple times. Our refinement checker was able to find this unsound assumption in the linearizability abstraction which led to modifying our Chain-Replication implementation. This bug could have caused an error in the client of the Chain-Replication protocol as it was tested against the unsound linearizability abstraction.

During compositional systematic testing, abstractions are used for decomposition. False positives can occur if the abstractions used are too coarse-grained and contain behaviors not present in the implementation. The number of false positives uncovered

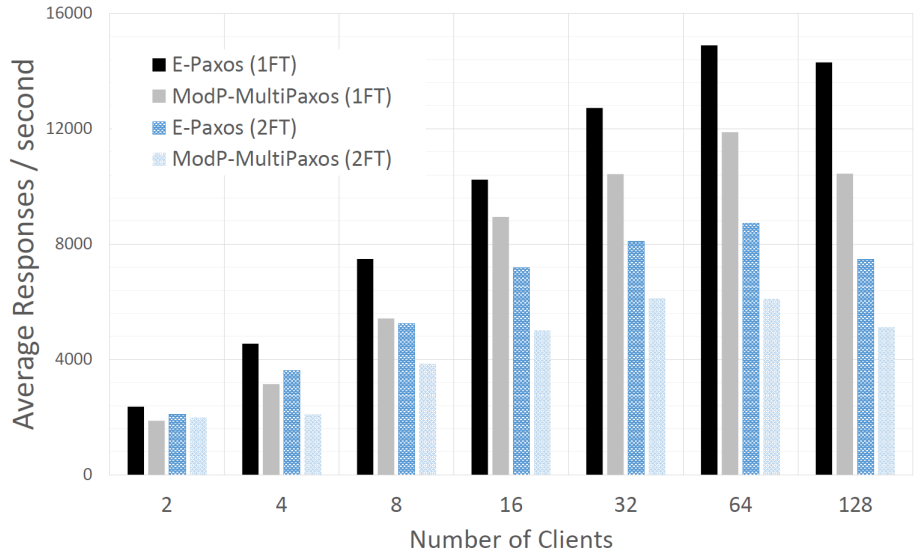


Figure 3.8: Performance of MODP HashTable using Multi-Paxos (MP) is comparable with an open source baseline implementation (mean over 60s close-loop client runs).

during compositional testing was low (4) compared to the real bugs that we found. We think that this could be because the protocols that we considered in this paper have well-studied and known abstractions.

3.3.3 *Performance Evaluation*

We would like to answer the question: *Can the distributed applications build modularly using* MODP *with the aim of scalable compositional testing rival the performance of corresponding state-of-the-art implementations?* We compare the performance of the code generated by MODP for the fault-tolerant hash-table built using Multi-Paxos against the hash-table built using the popular open-source reference implementation of Multi-Paxos from the EPaxos codebase [142, 143]. All benchmarking experiments for the distributed services were run on A3 Virtual Machine (with 4-core Intel Xeon E5-2660 2.20GHz Processor, 7GB RAM) instances on Azure.

To measure the update throughput (when there are no node failures in the system), we use clients that pump in requests in a closed loop; on getting a response for an outstanding request, the client goes right back to sending the next request. We scale the workload by changing the number of parallel clients from 2 to 128. For the experiments, each replica executes on a separate VM. Figure 3.8 summarizes our result for one fault-tolerant (1FT = 3 paxos nodes) and two fault-tolerant (2FT = 5 paxos nodes) hash-tables. We find the systematically tested MODP implementation achieves between 72%(2FT, 64 clients) to 80% (1FT, 64 clients) of peak throughput of the open source baseline (EPaxos codebase [142, 143]). The open source implementation of the

E-Paxos protocol suite is highly optimized and implemented in Go language (1169 LOC). We believe that the current performance gap between the two implementations can be further reduced by engineering our distributed runtime. The high-level points we would like to convey from these performance number is that it is possible to build distributed services using MODP that are rigorously tested and have comparable performance to the open source counterpart.

3.4 SUMMARY

MODP is a new approach that makes it easier to build, specify, and test distributed systems. We use MODP to implement and validate a practical distributed systems protocol stack. MODP is effective in finding bugs quickly during development and get orders of magnitude more test-coverage than monolithic approach. MODP 's compositional testing has the power to generate and reproduce within minutes, executions that could take months or even years to manifest in a live distributed system. The distributed services built using MODP achieve performance comparable to state-of-the-art open source equivalents.

Part II

VERIFICATION AND SYSTEMATIC TESTING OF EVENT-DRIVEN SYSTEMS

In Part i, we introduced the P language for modular and safe event-driven programming. The MODP module system (Chapter 2) allows programmers to perform compositional verification (or systematic testing) of P programs. Scalable analysis (using model-checking) of even the decomposed system is difficult because of the state-space explosion problem. State-space explosion occurs due to several reasons – explosion of the underlying data-space domain, explosion due to the myriad interleavings caused due to concurrency, and explosion due to the unbounded message buffers used for communication.

In this part, we describe two potential approaches for mitigating the state-space explosion problem: **(1)** ***Search prioritization-based Falsification*** (or bug-finding): Extending the model-checker with guided or directed search geared towards falsification of the property to be verified; and **(2)** ***Abstraction-based Verification***: Using a sound abstraction (*superset*) of the program behaviors to simply the overall verification problem.

Chapter 4 presents a scalable approach for systematic testing of P programs. We introduce the concept of a *delaying explorer* with the goal of performing prioritized exploration of the behaviors of an asynchronous reactive program. A delaying explorer stratifies the search space using a custom strategy, and a delay operation that allows deviation from that strategy. We show that prioritized search with a delaying explorer performs significantly better than existing approaches for finding bugs in P programs. In Chapter 5, we present an abstraction-based model-checking approach for verification of almost-synchronous event-driven systems implemented using P. We introduce *approximate synchrony*, a sound and tunable abstraction for verification of almost-synchronous systems. We show how approximate synchrony can be used for verification of both time-synchronization protocols and applications running on top of them. Moreover, we show how approximate synchrony also provides a useful strategy to guide state-space exploration during model-checking.

4

SYSTEMATIC TESTING OF ASYNCHRONOUS EVENT-DRIVEN PROGRAMS

Thorough testing is the touchstone of reliability in quality assurance and control of modern production engineering.

— C.A.R Hoare in "How Did Software Get So Reliable Without Proof?"

Systematic testing of asynchronous programs is notoriously difficult due to the nondeterministic nature of their computation; an error could result from a combination of some choice of inputs and some interleaving of event handlers. This chapter is concerned with the problem of systematic testing of complex P programs by automatically enumerating all sources of nondeterminism, both from environment input and from scheduling of concurrent processes.

The main challenge in scaling systematic testing to real-world P programs is a large number of behaviors that explode exponentially with the complexity of the implemented system. Techniques such as state caching [104] and partial-order reduction [84] have been developed to combat this explosion, yet their worst-case complexity remains exponential. In practice, the search often takes too long and has to be terminated because of a time-bound, thereby giving no information to the programmer. Therefore, researchers have been motivated to investigate prioritized search techniques, both deterministic [71, 146] and randomized [31], to provide partial coverage information. However, all of these techniques have been developed for shared-memory multithreaded programs. In asynchronous reactive programs, the primary mechanism for communication among concurrent processes is message-passing rather than shared memory. We have discovered empirically (Section 4.4) that prioritization techniques developed for multithreaded programs are not effective when applied to message-passing programs.

Systematic Testing using Delaying Explorer. We introduce a new technique for the systematic testing of asynchronous reactive programs. Our technique is inspired

by the notion of a delaying scheduler [71] for multithreaded programs. A delaying scheduler is a *deterministic* thread scheduler equipped with a delay operation whose invocation changes the default scheduling strategy. For asynchronous reactive programs, we generalize this notion to a delaying explorer of *all* nondeterministic choices (Section 4.1), both from the input and the interleaving of event handlers. The crucial observation that makes a delaying explorer suitable for systematic testing is that every execution can be produced by introducing a finite number of delays in the deterministic execution prescribed by the explorer (Theorem 4.1.1). We show that appropriately designed delaying explorers are significantly more effective than existing prioritization techniques in searching for errors in executions of asynchronous message-passing systems.

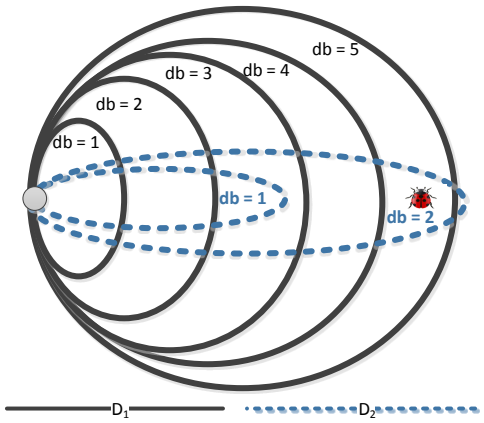


Figure 4.1: Stratification using Delaying Explorers. D_1 and D_2 represent two different search strategies induced by different delay explorers, and db represents the delay budget.

A delaying explorer induces stratification in the search space of all executions. A stratum is the set of executions that require the same number of delays. Figure 4.1 represents the stratification pictorially; $db = 1$ is the set of executions with one delay, $db = 2$ is the set of executions with two delays, and so on. A delaying explorer specifies a prioritized search that explores these strata in order. Since the number of possible executions increases exponentially with the delay budget, exploration for high budget values becomes prohibitively expensive. Therefore, a delaying explorer is practical only if bugs are uncovered at low values of the delay budget. Figure 4.1 shows the stratification induced by two different delaying explorers. The explorer D_2 is more effective than D_1 at discovering a particular bug if that bug lies in a lower stratum for D_2 than for D_1.

The difference in stratification induced by different delaying explorers has practical consequences. We have observed empirically that there is considerable variance in the

speed of detecting errors across different delaying explorers for different test problems[1]. Motivated by this observation, we have designed a general delaying explorer interface that helps programmers quickly write custom search strategies in a small amount of code, typically less than 50 LOC. Delaying explorers also provides developers and testers with a simple and elegant mechanism to express domain-specific knowledge regarding parts of the search space to prioritize. We have written several delaying explorers using our framework and used them to find bugs in implementations of distributed protocols that could not be discovered using any other method. We describe a particular case study in Section 4.4.

Given a delaying explorer, we need techniques for effectively exploring the strata induced by the explorer. In this chapter, we also present two algorithms —Stratified Exhaustive Search (SES) and Stratified Sampling (SS)— for solving this problem. SES performs a stratified search by iteratively incrementing the delay budget and exhaustively enumerating all schedules that can be explored with a given delay budget. Inspired by model checking techniques, we incorporate state caching to avoid redundant exploration of schedules. By caching the states visited along with execution, we can prune the search if an execution generated subsequently leads to a state in the cache. Incorporating state caching in delaying exploration is nontrivial because search is performed over executions of the composition of the program and the delaying explorer, both reading and updating their private state in each step of the execution. The naive strategy of caching the product of the program and the explorer state does not work because the delaying explorer can be an arbitrary program with a huge state space of its own. Instead, our algorithm caches only the program state yet guarantees that in the limit of increasing delay budgets, all executions of the program are covered. Our evaluation shows that SES finds bugs orders of magnitude faster than prior prioritization techniques on our benchmarks (Section 4.4).

Even though state caching is an important optimization, it is not a panacea to the explosion inherent in systematic testing. The complexity of the algorithm mentioned in the previous paragraph still grows exponentially with the number of allowed delays. Consequently, if a delaying explorer is unable to find a bug quickly within a few delays, the search must be stopped because of the external time bound. To further scale search to large delay budgets, we present the SS algorithm, which performs stratified sampling of the search space with probabilistic guarantees. Our algorithm guarantees that any execution that is visited with db delays is sampled with probability at least $1/L^{db}$, where L is the maximum number of program steps. SS is useful because it allows even distribution of the limited time resource over the entire search space. Furthermore, since each sample is generated independently of every other sample, random exploration can be easily and efficiently parallelized or distributed. Finally, for some systems state caching may not be possible because of the difficulty of taking a snapshot of the entire system state. In this situation, search based on random sampling

1 A test problem is the combination of a program and a specification.

could be very useful. We empirically show (Section 4.4) that on our benchmarks, SS can find bugs faster, often by an order-of-magnitude, compared to the prior best technique [31] for random sampling of executions of multithreaded programs.

We have implemented these algorithms in the P explorer for systematic testing of P [53] programs. We note that our techniques are not limited to the P language. They generalize to any programming system with two properties: (1) ability to create executable models of the execution environment of a program, and (2) control over all sources of nondeterminism in program semantics. The search prioritization approach using delaying explorers can then be applied to analyze programs in that framework by systematically enumerating non-deterministic choices with stratification.

We conclude this section by summarizing our contributions:

1. We introduce delaying explorers as a foundation for systematic testing of asynchronous reactive programs. We empirically demonstrate that for the domain of message-passing programs, delaying explorers are better, often by an order-of-magnitude than existing prioritization techniques.

2. We observe that the efficacy of a delaying explorer depends on the type of bug and scenario that causes it. To enable programmers to write custom explorers, we have created a flexible interface for specifying explorers. We have written four delaying explorers, each in less than 50 LOC, using our interface.

3. We present the SES algorithm that uses state-caching for efficiency while prioritizing search using a delaying explorer. The algorithm guarantees soundness even without caching the state of the delaying explorer.

4. We present the SS algorithm to efficiently sample executions with a fixed number of delays. Our algorithm guarantees that if a buggy execution exists with db delays for a given delaying explorer, then each sample triggers the bug with probability at least $1/L^{db}$ where L is the maximum number of steps in the program.

4.1 DELAYING EXPLORER

In this section, we provide intuition for delaying explorers and their use in systematic testing of asynchronous reactive systems. We begin by formally stating our model of programs and explorers.

A program $\mathcal{P}$ is a tuple (S, Cid, T, s_0):

1. S is the set of states of $\mathcal{P}$.

2. Cid is a finite set of nondeterministic choices that $\mathcal{P}$ can make during execution. This set includes both choices due to the scheduling of concurrent processes in $\mathcal{P}$ and choices due to nondeterministic input received by each process.

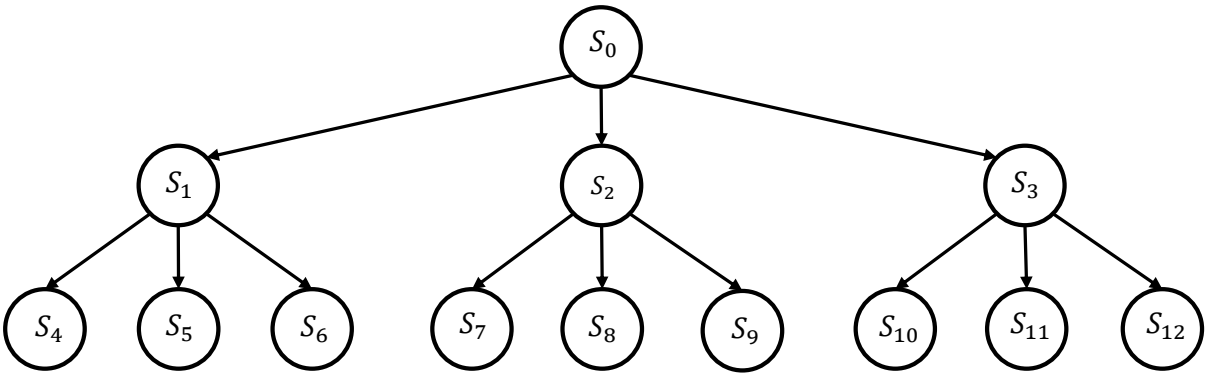


Figure 4.2: A concurrent program represented as a transition graph

3. $T \in \mathit{Cid} \times S \rightharpoonup S$ is the transition function of $\mathcal{P}$. If $s' = T(c, s)$, we say that (s, s') is a transition of $\mathcal{P}$. We define $\mathit{Choices}(s) = \{c \mid \exists s'.\ T(c, s) = s'\}$.

4. s_0 is the initial state of $\mathcal{P}$.

A sequence of states $s_0, s_1, s_2, \ldots, s_n$ is an *execution* of $\mathcal{P}$ if (s_i, s_{i+1}) is a transition of $\mathcal{P}$ for all $i \in [0, n)$. A state $s \in S$ is *reachable* if it is the final state of some execution. An infinite sequence of states $s_0, s_1, s_2, \ldots$ is an *infinite execution* of $\mathcal{P}$ if (s_i, s_{i+1}) is a transition of $\mathcal{P}$ for all $i \geqslant 0$. We assume that $\mathcal{P}$ is *terminating*, i.e., it does not have any infinite executions.

The formalization of the nondeterministic transition graph of an asynchronous reactive program is standard in the literature; it is depicted pictorially in Figure 4.2. The exploration algorithms popularized by model checking tools, e.g., SPIN [104], view the transitions coming out of a state as unordered; the order in which those transitions are explored is considered an implementation-level detail. A delaying explorer, formalized below, instead considers the order of transitions an important concern for efficient exploration. It provides a general interface for specifying this order based on the entire history of the program execution.

A delaying explorer $\mathcal{D}$ is a tuple $(D, \mathit{Next}, \mathit{Step}, \mathit{Delay}, d_0)$:

1. D is the set of states of $\mathcal{D}$. The state of the explorer typically includes a data structure, e.g., stack or queue, to maintain an ordering among the choices available to the program.

2. $\mathit{Next} \in D \to \mathit{Cid}$ is a total function. Given an explorer state d, the choice $\mathit{Next}(d)$ is prescribed by the explorer to be taken next.

3. $\mathit{Step} \in S \times D \to D$ is a total function. Suppose we have a program state s and a explorer state d, and we execute the choice $\mathit{Next}(d)$ at s. Then $\mathit{Step}(s, d)$ yields the explorer state corresponding to the program state $T(\mathit{Next}(d), s)$. The *Step* function enables building explorers which change their state in response to

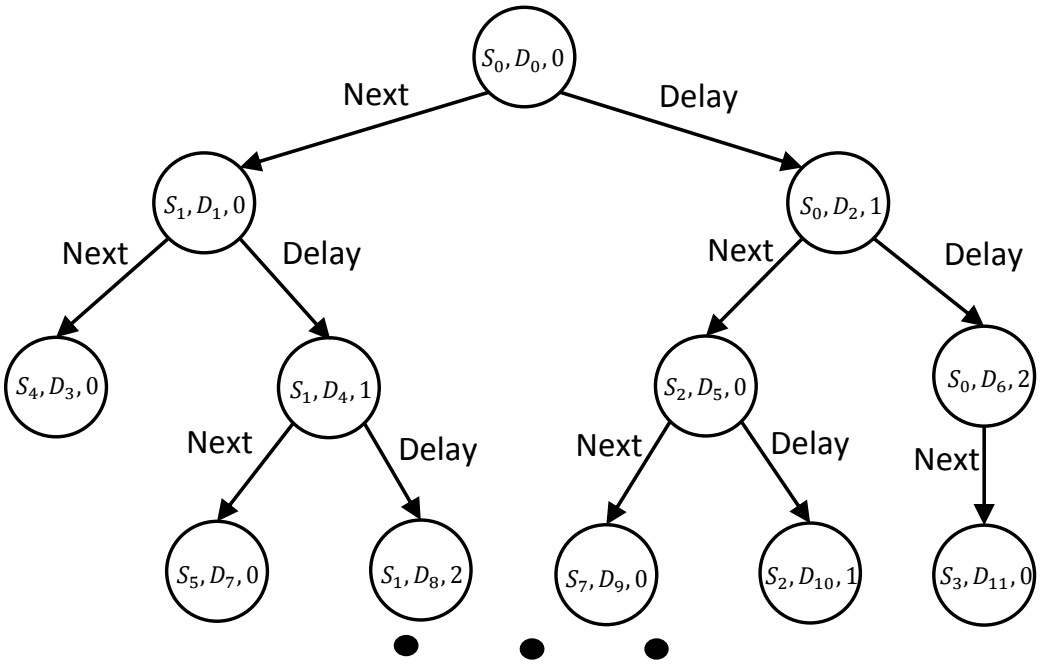


Figure 4.3: A concurrent program composed with a delaying explorer

specific events that occur during the execution of the program, such as sending or receiving of messages, creation of new processes, etc.

4. $Delay \in D \to D$ is a total function. Given an explorer state d, the application $Delay(d)$ yields a new explorer state. The *Delay* function provides a mechanism to change the next choice to be explored. We call the operation a *delay* operation as it delay's the current choice of the deterministic scheduler and moves to the next choice.

5. d_0 is the initial state of $\mathcal{D}$.

Consider a delaying explorer that attempts to order the outgoing transitions of each state left to right for the program in Figure 4.2. The unfolding of the nondeterminism in this program as controlled by such a delaying explorer is shown in Figure 4.3. We formalize and explain the intuition behind this figure below.

Let $(\mathcal{P}, \mathcal{D})$ denote the composition of a program $\mathcal{P}$ and a delaying explorer $\mathcal{D}$. A state of $(\mathcal{P}, \mathcal{D})$ is a triple (s, d, n), where s is the state of $\mathcal{P}$, d is the state of $\mathcal{D}$, and n is the number of consecutive delay operations applied in state s. A finite sequence $(s_0, d_0, n_0) \xrightarrow{x_0} (s_1, d_1, n_1) \xrightarrow{x_1} (s_2, d_2, n_2) \xrightarrow{x_2} \cdots$ is an execution of $(\mathcal{P}, \mathcal{D})$ if for all $i \geqslant 0$, either (1) $x_i = Next$, $n_{i+1} = 0$, $T(Next(d_i), s_i) = s_{i+1}$, and $Step(s_i, d_i) = d_{i+1}$, or (2) $x_i = Delay$, $n_{i+1} = n_i + 1$, $n_{i+1} < card(Choices(s))$, $s_i = s_{i+1}$, and $Delay(d_i) = d_{i+1}$.

In this execution, a transition $\xrightarrow{Next}$ is a *Next*-transition and $\xrightarrow{Delay}$ is a *Delay*-transition. In Figure 4.3, each state has these two outgoing transitions precisely. A triple (s, d, n) is a reachable state of $(\mathcal{P}, \mathcal{D})$ if it occurs on an execution. A *db-delay execution* of $(\mathcal{P}, \mathcal{D})$ is one in which the number of *Delay*-transitions is *db*. Thus, a delaying explorer $\mathcal{D}$ induces a stratification of the executions of a program $\mathcal{P}$ such that the i-th stratum contains precisely the set of i-delay executions.

In order to ensure that all behaviors are covered, the delaying explorer must ensure that successive applications of *Delay* generate all nondeterministic choices from a state. To formalize this requirement, we define $Delay^k$ (for $k \geqslant 0$) inductively as

$$\begin{aligned} Delay^0(d) &= d \\ Delay^{k+1}(d) &= Delay(Delay^k(d)) \end{aligned} \tag{4.1}$$

and $Next^k$ (for $k \geqslant 0$) inductively as

$$\begin{aligned} Next^0(d) &= \{\} \\ Next^{k+1}(d) &= Next^k(d) \cup \{Next(Delay^k(d))\}. \end{aligned} \tag{4.2}$$

A delaying explorer $\mathcal{D}$ is *sound* with respect to a program $\mathcal{P}$ if $Choices(s) = Next^{card(Choices(s))}(d)$ for every reachable state (s, d) of $(\mathcal{P}, \mathcal{D})$. This property states that all nondeterministic choices in a state are covered through iterative application of the *Delay* operation composed with *Next*. In Figure 4.3, all successors, S_1 through S_3, of state S_0 are reachable via at most two invocations of *Delay*. This property guarantees (Theorem 4.1.1) that reachability analysis on $(\mathcal{P}, \mathcal{D})$ is equivalent to reachability analysis on $\mathcal{P}$.

Theorem 4.1.1: Soundness of Delaying Explorer

Consider a program $\mathcal{P}$ and a delaying explorer $\mathcal{D}$ that is sound with respect to $\mathcal{P}$. A state s is reachable in $\mathcal{P}$ iff (s, d) is reachable in $(\mathcal{P}, \mathcal{D})$ for some d.

Proof. The proof is by induction on the length of an execution of $\mathcal{P}$. The base case for the initial state of $\mathcal{P}$ is trivial. For the inductive case, suppose s is reachable in k steps and s has a transition to s'. From the induction hypothesis, (s, d) is reachable in $(\mathcal{P}, \mathcal{D})$ for some d i.e., all non-deterministic choices can be explored from (s, d). Therefore, we can take a sequence of transitions in $(\mathcal{P}, \mathcal{D})$ to get to (s', d') for some d'. ■

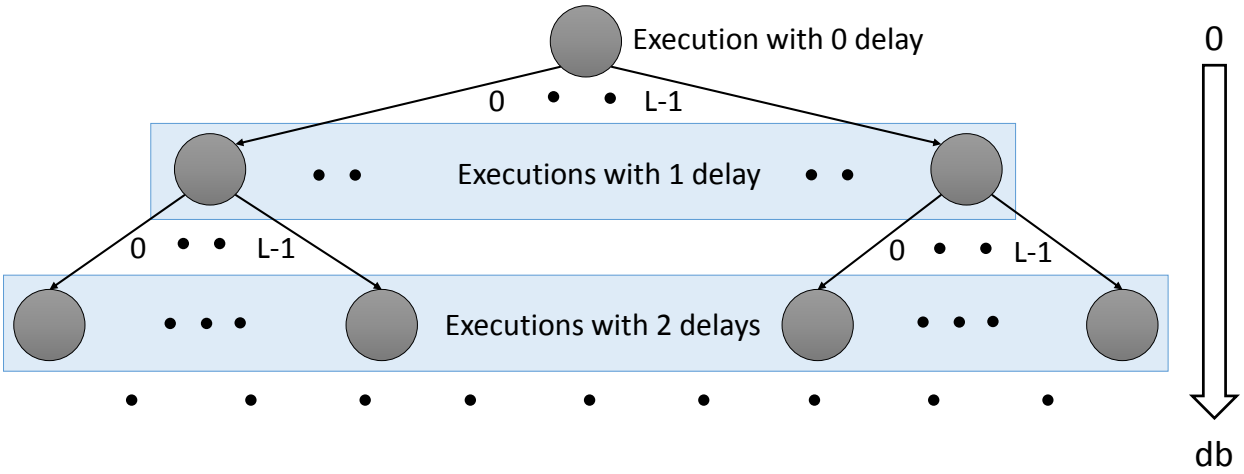


Figure 4.4: Stratified Exhaustive Search

Example 4.1.1: Round-Robin Delaying Explorer

Let us consider a simple program in which the only source of nondeterminism is the scheduling of concurrent processes. An example of a delaying explorer for this program is a round-robin process scheduler. The state D *of this scheduler is a queue of process ids initialized to contain the id of the initial process. Next returns the process id at the head of the queue. Step instruments the program's execution so that the id of a new process is added to the tail, the id of a terminated process is removed, and the id of a blocked process is moved to the tail. Delay moves the process id at the head to the tail. This explorer maintains the invariant that the ids of all enabled processes are present in the queue. By applying the Delay operation at most* n *times, where* n *is the size of the queue, any enabled process can be moved to the head and be returned by a subsequent call to Next. Therefore, this explorer is sound with respect to the program.*

4.2 STRATIFIED EXHAUSTIVE SEARCH

Figure 4.4 shows a pictorial representation of stratified exhaustive search of a program with respect to a delaying explorer. In this picture, L is the maximum number of steps in the program. In contrast to the graphs in Figure 4.2 and Figure 4.3 where a node represents the program state, each node in Figure 4.4 is a complete execution of the program. The root node is the execution with no delays. This execution presents at most L positions to insert a delay operation, each yielding another complete execution with a single delay operation. These executions are indicated by the nodes at the end of the edges coming out of the root node. This process can be continued until all executions have been generated. It is clear that there can be at most L^{db} executions

with no more than *db* delays. Thus, for small values of *db*, it is feasible to enumerate all executions even for large values of L. This observation suggests our stratified exhaustive search algorithm (SES) which generates executions level by level, exploring all executions at a level before moving to the next level. A delaying explorer induces a stratification of the executions of a program; in general, different delaying explorers induce different stratification for the same program. Thus, a delaying explorer is a mechanism to bias the search performed by our SES algorithm to different parts of the execution space.

Algorithm 4.2.1 Stratified Exhaustive Search

```
 1: var db : ℕ
 2: var Frontier : Dictionary⟨S, (D × ℕ)⟩
 3: var Cache : Set⟨S⟩
 4: function DelayBoundedDFS(s : S, d : D, i : ℕ, n : ℕ)
 5:     var s′ : S
 6:     while (i < card(Choices(s)))
 7:         s′ ← T(Next(d), s), i ← i + 1
 8:         if (s′ ∉ Cache) then
 9:             Cache ← Cache ∪ {s′}
10:             DelayBoundedDFS(s′, Step(s, d), 0, n)
11:         end if
12:         if (n = db ∧ i < card(Choices(s))) then
13:             Frontier(s) ← (d, i)
14:             return
15:         end if
16:         d ← Delay(s, d), n ← n + 1
17:     end
18: end function
19:
20: function SES
21:     var db′ : ℕ
22:     var Frontier′ : Dictionary⟨S, (D × ℕ)⟩
23:     db ← 0, Frontier ← ∅, Cache ← {s0}
24:     DelayBoundedDFS(s0, d0, 0, 0)
25:     while (Frontier ≠ ∅)
26:         Frontier′ ← Frontier, Frontier ← ∅
27:         db′ ← db, db ← db + δ
28:         for all ((s, d, i) ∈ Frontier′) do
29:             DelayBoundedDFS(s, Delay(s, d), i, db′ + 1);
30:         end
31:     end
32: end function
```

The Algorithm 4.2.1 takes as input a program $\mathcal{P}$, a delaying explorer $\mathcal{D}$, and a parameter $\delta > 0$. It uses three global variables. The integer *db*, initialized to 0 and iteratively incremented by δ, contains the current delay bound. During the search, a frontier of pending executions that go beyond the current delay bound, is maintained in the dictionary *Frontier*. For each state s in the frontier, *Frontier* contains a pair (d, i), where d is the explorer state just prior to the the execution of i-th transition from state s. The mapping from s to (d, i) is put into the frontier because the execution of the i-th transition would require more delays than the current bound. Finally, we optimize the search by using a cache of (hashes of) visited states maintained in the set *Cache*.

The workhorse of our algorithm is DELAYBOUNDEDDFS, a procedure with four parameters—program state s, explorer state d, transition count i, and delay count n. The goal of DELAYBOUNDEDDFS is to continue exploration from state s. The transition count i is the number of transitions already explored from s. The delay count n is the number of delays required, starting from the initial state, to execute the next transition out of s. DELAYBOUNDEDDFS iterates through the transitions from s by repeatedly invoking the *Next* operation of the delaying explorer to find out which transition to execute also, incrementing i to indicate the execution of another transition. For each discovered state s', if s' is not present in *Cache* then it is added to *Cache* and DELAYBOUNDEDDFS is called recursively on s'. The *Delay* operation of the delaying explorer needs to be invoked to move to the next schedule. If the current delay count n has already reached the current delay bound *db*, and there is at least one more transition to be executed, then exploration cannot continue from s and work for the remainder of exploration from s is added to the frontier. Otherwise, the *Delay* operation is used to update d, and the delay count n is incremented.

The top-level procedure of our algorithm is SES. This procedure initializes *db* to 0 and *Frontier* and *Cache* to $\emptyset$. It then executes two nested loops. The outer loop iterates over the value of *db* incrementing it by δ each time around. The goal of each iteration of this loop is to restart each pending exploration in the current frontier. To do this task, a copy of *Frontier* is made in *Frontier'* and *Frontier* is reset to $\emptyset$. The inner loop then picks each work item in *Frontier'* and invokes DELAYBOUNDEDDFS with it. The execution of the inner loop refills *Frontier* which is again emptied in the next iteration of the outer loop. Theorem 4.2.1 formalizes the correctness of the SES algorithm.

Theorem 4.2.1: Soundness of Stratified Exhaustive Search

Consider a program $\mathcal{P}$ and a delaying explorer $\mathcal{D}$ that is sound with respect to $\mathcal{P}$. The SES *Algorithm 4.2.1 terminates and visits a state s' iff s' is reachable from s_0.*

Proof. We argue that SES terminates for any program $\mathcal{P}$. Since $\mathcal{P}$ does not have any infinite executions and $\mathit{Choices}(s) \subseteq \mathit{Cid}$ is finite for any state s, any invocation of DELAYBOUNDEDDFS terminates. The inner loop in SES terminates because each iteration removes one entry from *Frontier'*. The termination of the outer loop is based

on the observation that the inner loop adds a state s to *Frontier* only if there is an execution reaching s with more delays than db'. The number of delays in an execution is bounded by $L \times Cid$, where L is the maximum number of steps in $\mathcal{P}$. Since the outer loop increments db' in each iteration and the number of delays for an execution is bounded, eventually *Frontier* will become empty.

Next, we argue that SES is safe and eventually visits all reachable states of $\mathcal{P}$. This argument depends on the crucial assumption that the delaying explorer $\mathcal{D}$ is sound with respect to $\mathcal{P}$. Because of this property, by applying delays repeatedly in a state all outgoing transitions are taken.

Neither the termination nor the safety argument for our algorithm depends on *Cache*. The only role of *Cache* is to optimize the search by avoiding redundant executions. Therefore, there is considerable flexibility in how much memory is devoted to the storage for *Cache*. The two extreme cases are when *Cache* is not used at all, and when all visited states are put into *Cache*. However, it is possible, and our implementation supports imposing a bound on the memory consumption for *Cache* beyond which states are either not added to *Cache* or added with replacement.

An important consideration in our use of *Cache* is that we store only the program state in it and avoid storing the explorer state. This design has the advantage that we get the maximum pruning out of the use of state caching. If a state s is first visited with explorer state d and later with explorer state d', the second visit is ignored even if it happened with fewer delays compared to the first visit. As a result, we can avoid re-exploration for the second visit. However, it may be possible that a state is discovered with a higher delay than the minimum delay required to visit it. We believe that this trade-off is good because the primary goal of a delaying explorer is to bias the search rather than enforcing strict priority.

Finally, we note that it is enough to store only a hash of a state in *Cache*. However, it is essential to store the full state both when it is passed as a parameter to DELAYBOUNDEDDFS or when it is stored in *Frontier* since the program needs to be executed from it. For the latter uses, a state could either be cloned or reconstructed by re-executing the program from the beginning.

■

4.3 STRATIFIED SAMPLING

In the previous section, we described the SES algorithm to perform an exhaustive stratified search over the executions of an asynchronous reactive program. In this section, we describe a complementary algorithm that enables stratified exploration via near-uniform random sampling of executions from the strata induced by a delaying explorer; we call this algorithm the stratified sampling algorithm (SS).

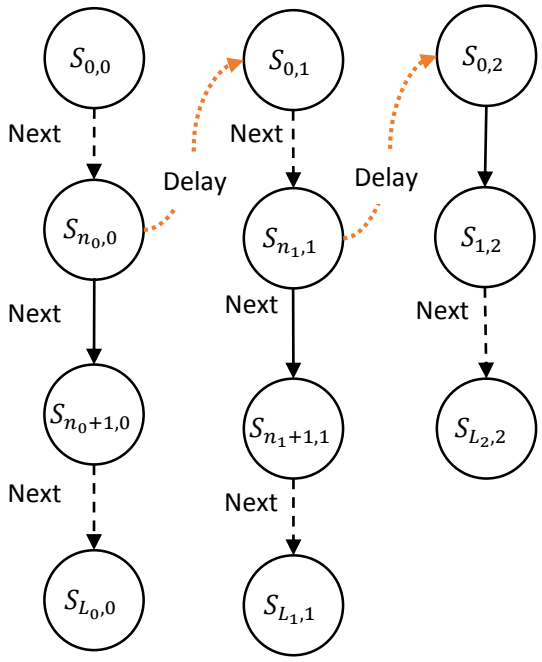


Figure 4.5: A run of SS algorithm

To motivate why random sampling is beneficial, we note that the complexity of the SES algorithm grows exponentially with the upper bound on the number of allowed delays. Consequently, if a delaying explorer is unable to find a bug quickly within a few delays, the search often takes more time than the programmer is willing to wait for. To deal with this common problem, a time bound is usually supplied in addition to the number of delays. When an external time bound could stop the search before the delay limit has been reached, random sampling has certain advantages over exhaustive deterministic exploration. First, unlike deterministic exploration, random sampling can sample *every* execution with a non-zero probability, making it possible to distribute the limited time resource over the entire search space. Second, since each sample is generated independently of every other sample, random exploration can be easily and efficiently parallelized, an important advantage in an era where parallelism is abundantly available via multicore and cloud computing.

Figure 4.5 shows how our algorithm samples an execution with two delay operations. First, the *ExecutePath* function (defined later in Algorithm 4.3.2) executes the program using a custom strategy defined by the delaying scheduler without introducing any delays. The *ExecutePath* function returns the length of the execution L_0 from the start state to the terminal state. Using **choose**(L_0) we uniformly pick a value n_0 in the range $[0, L_0)$ to insert the first delay. When *ExecutePath* is invoked again, it introduces a delay at n_0, deterministically executes the program up to termination, and returns L_1, the length of the path since the last delay. Using **choose**(L_1) we

uniformly pick a value n_1 in the range $[0, L_1)$ to insert the second delay. Finally, the execution $S_{0,0} \rightarrow^* S_{n_0,0} \rightarrow S_{0,1} \rightarrow^* S_{n_1,1} \rightarrow S_{0,2} \rightarrow^* S_{L_2,2}$ represents a random execution with two delays.

Given a program $\mathcal{P}$, a delaying explorer $\mathcal{D}$, and a delay bound *db*, an invocation of *DelayBoundedSample* (Algorithm 4.3.2) produces a terminating execution of $\mathcal{P}$ with no more than *db* delays. The random exploration performed by our algorithm is very different in spirit from the classical random walk algorithm on a state-transition graph (Figure 4.2) which starts from the initial state and executes the program by randomly selecting a transition out of the current state. This naive random walk, although it guarantees a non-zero probability for sampling any execution, suffers from the problem that the probability of sampling long executions decreases exponentially with the execution length. Instead, our algorithm performs a random walk, not on the state-transition graph, but a different graph (Figure 4.4) induced by the delaying explorer $\mathcal{D}$. In this graph, each node is a complete terminating execution (as opposed to a state), and an edge is a position in the execution for inserting a delay (as opposed to transition). We show later that the probability of sampling any execution requiring *db* delays is at least $\frac{1}{L^{db}}$. Unlike the naive random walk, the probability of sampling an execution is exponential in the number of required delays rather than the number of steps. A long execution has just as much chance to be produced as a short execution with the same number of delays, thereby eliminating the bias towards short executions.

The Algorithm 4.3.2 uses a single global variable *path*, a sequence of natural numbers. This sequence represents a path as follows. For each i starting from 0 and up to $path.Length - 1$, execute $\mathcal{P}$ for $path[i]$ steps followed by a delay. Finally, execute $\mathcal{P}$ until it terminates. The procedure *ExecutePath* performs the execution encoded by *path* and returns the number of steps performed after the last delay.

The procedure *DelayBoundedSample* invokes the procedure *ExecutePath* repeatedly to randomly sample an execution with *db* delays. If the initial state s_0 does not have any transitions, there is nothing to do. Otherwise, it sets *path* to the empty sequence and calls *ExecutePath* which executes $\mathcal{P}$ without any delays. The algorithm chooses a step at random from the number of steps returned by *ExecutePath* as the position to execute a delay operation. It extends *path* with it and invokes *ExecutePath* again to create a new execution. It continues to do so iteratively until the number of delays in the execution has reached *db*. A single invocation of *DelayBoundedSample* samples a single execution with *db* delays. To calculate this sample, it must re-execute the program *db* times and perform *db* random choices.

Algorithm 4.3.2 Stratified Sampling: Near-Uniform Random Sampling

```
1: var path : Sequence⟨ℕ⟩
2:
3: function ExecutePath
4:     var i, j : ℕ
5:     var s : S
6:     var d : D
7:     s ← s_0, d ← d_0, i ← 0
8:     while (i < path.Length)
9:         j ← 0
10:        while (j < path[i])
11:            s ← T(Next(d), s), d ← Step(s, d), j ← j + 1
12:        end
13:        d ← Delay(s, d), i ← i + 1
14:    end
15:    j ← 0
16:    while (0 < card(Choices(s)))
17:        s ← T(Next(d), s), d ← Step(s, d), j ← j + 1
18:    endreturn j
19: end function
20:
21: function DelayBoundedSample
22:    var i, l : ℕ
23:    if (card(Choices(s_0)) = 0) then return
24:    end if
25:    path ← ∅
26:    l ← ExecutePath
27:    i ← 0
28:    while (i < db)                                  ▷ invariant 0 < l
29:        path.Append(choose(l))
30:        l ← ExecutePath
31:        i ← i + 1
32:    end
33: end function
34:
35: function SS
36:    var i : ℕ
37:    db ← 1
38:    while true
39:        i ← 0
40:        while i < NumSamples(db))
41:            DelayBoundedSample
42:            i ← i + 1
43:        end
44:        db ← db + 1
45:    end
46: end function
```

Theorem 4.3.1: Probabilistic Guarantees for Stratified Sampling

Consider a program $\mathcal{P}$ and a delaying explorer $\mathcal{D}$ that is sound with respect to $\mathcal{P}$. Let L be the maximum number of steps along any execution of $\mathcal{P}$. For any integer $db \geqslant 0$ and any execution τ of $(\mathcal{P}, \mathcal{D})$ with db delays, the SS Algorithm 4.3.2 generates τ with probability at least $\frac{1}{L^{db}}$.

Proof. The SS algorithm performs a random walk on the graph in Figure 4.4. The branching factor of this graph is bounded by L, the maximum number of steps in $\mathcal{P}$, and its depth is bounded by the delay bound db. Since L^{db} bounds the number of terminal nodes in the graph, the probability of sampling any execution requiring db delays is at least $\frac{1}{L^{db}}$. ■

Algorithm 4.3.2 also shows a procedure *SS* that repeatedly invokes *DelayBoundedSample* to implement a stratified sampling algorithm. This procedure has a (timeout-terminated and infinite) outer loop that repeatedly increases the delay bound db. The inner loop samples $NumSamples(db)$ executions from the set of executions with exactly db delays by invoking *DelayBoundedSample* repeatedly. Our algorithm is parameterized by a function *NumSamples* that specifies the number of executions to be sampled for each delay bound. As we have explained before, the number of executions increases exponentially with the number of available delays. Therefore, we believe that a practical *NumSamples* function should also have an exponential dependency on the delay bound. For our evaluation (Section 4.4), we chose $c_1 + c_2^{db}$ to be the shape for $NumSamples(db)$; through trial and error, we found that $c_1 = 100$ and $c_2 = 3$ work well for the benchmarks.

4.4 EVALUATION

In this section, we first provide an overview of our implementation of the delaying explorers for the systematic testing of P programs and then present the empirical evaluation of the their efficacy in finding bugs in complex asynchronous reactive systems.

4.4.1 *Implementation of the Delaying Explorers*

Recollect that there are two sources of non-determinism in the semantics of P programs: **(1)** P has interleaving non-determinism because the language provides a primitive for dynamic machine creation. As a result, multiple machines can be executing concurrently. In each step, one machine can be chosen nondeterministically to execute, and it can either compute on the local state or dequeue a message or send a message to another machine. This non-determinism implicitly creates non-determinism in the

order in which messages are delivered to a machine. The code of a machine has to be programmed robustly and tested so that it continues to perform safely regardless of the reordering. **(2)** a P program may also make an explicit non-deterministic choice by using the special expression `$` whose evaluation results in a non-deterministic *Boolean* choice. This feature is extremely useful for modeling the environment of reactive systems; like a non-deterministic component failure or message loss. To find bugs quickly and debug them, it is essential to control both these sources of non-determinism.

We have implemented the algorithms in Section 4.2 and Section 4.3 in the P explorer. The component of P explorer most pertinent to our implementation is state caching and the scheduler that orchestrates the depth-first search of the state-transition graph of the input program. We modified the explorer to query an external object implementing the `IDelayingScheduler` interface. The explorer invokes the method `Next` to determine the process whose transition it should explore and the method `Delay` to inform the scheduler of its decision to delay the next process.

```
1  interface IDelayingScheduler
2  {
3      // Next is called to get the next process to be executed
4      int Next ();
5
6      // Delay is called to cycle through scheduling choices
7      void Delay ();
8
9      // Start is called when a new process is created
10     void Start  (int processId);
11
12     // Finish is called when a process is terminated
13     void Finish (int processId);
14
15     // Step is called to communicate information about
           execution,
16     // e.g. change priority, blocked process, etc.
17     void Step (params object [] P);
18 }
```

Listing 4.1: Delaying Explorer Implementation Interface

The methods `Start`, `Finish`, and `Step` together implement the capability formalized by the *Step* function described in Section 4.1; these methods inform the delaying scheduler of important events occurring during the execution. The method `Start` is invoked whenever a new process is created and the method `Finish` whenever a process terminates. The method `Step` is used to implement a general mechanism for instrumenting the program's execution for updating the scheduler state.

Controlling nondeterminism: The general approach of controlling schedules in systematic testing frameworks [31, 85, 146] is to instrument the program at every synchronization points. In the context of asynchronous message passing programs like P, the only synchronization points are at enqueue of a message, blocking at dequeue and creation of a new machine. The P compiler automatically instruments the program at these three points and passes the information to the delaying explorer using the `Step` function. In addition to prioritizing interleaving non-determinism, a delaying explorer must also prioritize explicit non-deterministic choice. We adopt the convention that **false** is ordered before **true**. For a language that provides non-deterministic choice over types other than *Boolean*, the choices may be controlled by expanding the `IDelayingScheduler` interface.

4.4.2 *Empirical Evaluations of the Delaying Explorers*

Our evaluation was directed towards the following goals:
(Goal 1) Evaluate the performance of SES and SS in comparison with the best known approaches, preemption bounding [146] and probabilistic concurrency testing [31], respectively (Section 4.4.3).
(Goal 2) Evaluate the performance of different delaying explorers in finding bugs, and demonstrate the need for flexible delaying explorer interface (Section 4.4.4).
(Goal 3) Demonstrate the benefit of writing custom explorer with a case study of chain replication protocol(Section 4.4.5).

Experimental setup: All the experiments are performed on Intel Xeon E5-2440, 2.40GHz, 12 cores (24 threads), 160GB machine running 64 bit Windows Server OS. The ZING model checker can exploit multiple cores during exploration as its iterative depth-first search algorithm is parallel [189]. We do not report the time taken to find bugs as it is dependent on the degree of parallelism and the parallel explorer implementation, but instead, we report the number of distinct states explored (in the case of SES), and the number of schedules explored (in the case of SS) before finding the bug. Time taken to find the bug is directly proportional to these parameters. The numbers reported for the evaluation of stratified sampling algorithm in Table 4.1 are a median over 5 runs of the experiment.

Benchmarks: We have used P to implement a fault tolerant Transaction Management Service (TMS) (3) and a Windows driver communicating with an OSR device. The buggy programs used for evaluation were collected during the development of this protocol suite. Each row in Table 4.1 represents a different bug. We only consider hard-to-find bugs that led to unhandled-event exceptions (system crash) and violation of global safety specifications (written as monitors).

Table 4.1: Evaluation Results for SS and SES using various delaying explorers (Numbers in blue represent the winning search strategy)

Programs	Stratified Sampling						Stratified Exhaustive Search			
	No. of schedules explored before finding bug						No. of states explored before finding bug			
	RS	IRS	PCT	SS + Delaying Explorer			PB	SES + Delaying Explorer		
				RR	RTC	PRR		RR	RTC	PRR
2pc_1	9842	1891	1983	781	**331**	816	793221	8851	6571	**6512**
2pc_2	*	*	*	10943	6378	**6300**	*	*	17690	**9090**
2pc_3	*	2966	9835	1823	**1018**	4109	48321	1898	**1123**	2189
2pc_4	*	7629	*	*	**3321**	*	*	*	**5101**	77212
ChainRep_1	9655	**652**	9832	5607	9999	1985	*	92178	9913	**936**
ChainRep_2	*	*	*	34034	**7829**	28221	74231	32166	**8821**	88732
ChainRep_3	*	*	13283	2032	**1711**	6093	*	19731	**3452**	8981
ChainRep_4	4213	**313**	4439	3452	4249	1238	59234	**672**	5441	11742
ChainRep_5	196	77	55	**53**	110	101	*	3973	**521**	6652
ChainRep_6	*	*	*	*	*	*	*	78443	**44331**	54981
ChainRep_7	*	*	*	*	*	*	*	*	**3538**	*
ChainRep_8	*	4561	*	5513	*	**2201**	8342	9791	*	**8218**
ChainRep_9	*	*	*	66381	**9425**	16559	*	37222	**7812**	37213
ChainRep_10	782	159	**74**	129	331	888	4561	5431	1944	**1781**
MultiPaxos_1	*	5211	9934	7821	**765**	5819	*	**82114**	89341	88129
MultiPaxos_2	*	*	*	9872	**8873**	11239	*	15563	9983	**1934**
Multipaxos_3	*	*	*	*	15023	**9589**	*	18831	8923	**1198**
Paxos_1	229	86	592	122	**53**	233	3320	2233	**1098**	4312
Paxos_2	*	2211	*	9563	**831**	1874	77834	4912	**833**	8831
Paxos_3	*	*	*	*	*	*	*	*	**14832**	*
TMS_1	224	64	227	**12**	305	34	**553**	2220	660	8965
TMS_2	*	*	*	*	*	*	*	*	**44832**	*
TMS_3	#	#	#	#	#	**3009214**	#	#	#	#
TMS_4	#	#	#	#	**5530042**	#	#	#	#	#
OSR_1	435	122	332	**75**	122	1009	5532	4421	**683**	55392
OSR_2	756	78	131	115	**66**	224	12864	12931	**1634**	3212

*** → the search ran out of memory budget of 60GB or exceeded the time budget of 2 hours.**

 # → the search exceeded the time budget of 5 hours (running for longer duration).

4.4.3 *Evaluation of* SES *and* SS

Evaluating SES: We applied the iterative SES algorithm with different delaying explorers to the set of buggy programs (incrementing the value of *db* by 1 after each iteration). For evaluating the performance of SES, we implemented iterative preemption bounding [146] (PB) with state-caching in P explorer. Table 4.1 shows the number of distinct states explored before finding the bug by both the approaches. It can be seen that PB fails to find the bug in most of the cases, and in cases where PB succeeds, SES with some delaying explorer is able to find the bug orders of magnitude faster (except for TMS_1 and ChainRep_8). Also, there is much variance in the performance of SES when combined with different delaying explorers, which motivates the need for a flexible interface to write custom delaying explorers.

Evaluating SS: We implemented random scheduler (RS) [187] as the baseline for comparison. Random scheduler fails to find most of the bugs, as the probability of finding a bug decreases exponentially with the length of buggy execution. We found that iterative random scheduler (IRS) that combines random scheduling with iterative depth bounding performs better than simple random scheduling. Stratification in IRS is obtained by iteratively incrementing the maximum depth bound. We incremented the depth bound by 100 after each iteration and sampled $100 + 3^i$ executions from each stratum (where i is the iteration number).

We compared the iterative SS algorithm described in Section 4.3 with the PCT [31] algorithm, which is considered as state of the art in probabilistic concurrency testing. PCT provides probabilistic guarantees of finding a bug with *bug-depth* d, by randomly inserting d priority inversions. Most of the concurrency bugs using PCT were found with bug depth of less than 3 in [31, 147]. The PCT algorithm assumes the maximum length of program execution (k), which is hard to compute statically in the case of asynchronous reactive programs. We use $k = 5000$ and $d = 5$ for our experiments. Table 4.1 shows that PCT fails to find most of the bugs, confirming that the bugs in asynchronous programs generally have a larger *bug-depth*. In the cases where PCT succeeds in finding the bug, SS with some delaying explorer is orders of magnitude faster. Similar to the behavior of SES, for SS also, we see the variance in performance of different delaying explorers across different problems.

Comparison between SES **and** SS: We have extensively used both SES and SS for finding bugs in our implementations. In our experience, the SES algorithm can find bugs faster than SS in most of the cases as it uses state-caching to prune redundant explorations. Furthermore, SES can find low-probability bugs that occur at smaller values of delay budget faster than SS. In the case of ChainRep_6 and Paxos_3 there was a low probability bug at small delay budget; SS fails to find it whereas SES finds it.

As the delay bound increases, search space explodes exponentially. If there is a bug that requires a large delay budget for a given stratification strategy, then SES may

fail to find it due to running out of memory. We came across scenarios (TMS_3 and TMS_4 in Table 4.1) where SES ran out of memory but after running SS for a long time we uncovered a bug. SS can be kept running for a long time without any memory constraints. Since it performs sampling with probabilistic guarantees, it may find a bug at a larger delay budget where SES fails.

We can fruitfully combine both approaches as follows. Perform SES first to find all shallow (few delays) bugs quickly and get strong coverage guarantees. Once SES has uncovered all shallow bugs and has almost consumed the memory budget, perform SS from the frontier states and get probabilistic guarantees. We leave the evaluation of this combination for future work.

4.4.4 *Experience with Delaying Explorers*

We have implemented three different delaying explorers. In this section, we explain the construction of each explorer and the reasons for the variance in their performance.

Run-to-completion explorer (RTC**)**: The default strategy in RTC is to follow the causal sequence of events, giving priority to the receiver of the most recently sent event. When a delay is applied, the highest priority process is moved to the lowest priority position. Even for small values of delay bound, this explorer is able to explore long paths in the program since it follows the chain of generated events. In our experience, this explorer is able to find bugs that are at large depth better than any other explorer. For example, bugs in ChainRep_7 and TMS_2 were found by RTC at a depth greater than 1500 and delay budget less than 4 while other explorers could not find these bugs.

Round-robin explorer (RR**)**: The round-robin delaying explorer, explained earlier in Section 4.1, cycles through the processes in process creation order. It moves to the next task in the list only on a delay or when the current task is completed. Round-robin explorer has been used in the past ([71, 187]) to test multithreaded programs. In our experience, in most of the cases (Table 4.1) other delaying explorers perform better than RR. RR can be used for finding bugs that manifest through a small number of preemptions or interleaving between processes. Our evaluation shows that most bugs in asynchronous programs do not fall in that category.

Probabilistic round-robin explorer (PRR**)**: A probabilistic delaying explorer is one in which the *Step* operation is allowed to make random choices. While a deterministic delaying explorer induces a fixed stratification over the executions of a program, a probabilistic delaying explorer induces a probability space over stratification. We have experimented with a cannibalistic version of the round-robin explorer (PRR). We believe that the culprit behind the poor performance of the round-robin explorer is its default process scheduling order, which is based on the order of process creation. The simplest way to change this default order is to randomize it. Instead of inserting a freshly-created process at the tail of the queue, insert it at a random position in the

queue; everything else carries over from the round-robin explorer. The probabilistic round-robin explorer is still sound since the definitions of *Next* and *Delay* do not change. Table 4.1 indicates that PRR typically performs better than RR.

4.4.5 *Writing a Custom Delaying Explorer*

After testing the chain replication protocol using the three delaying explorers explained earlier, we tested it for more specific scenarios. One such scenario is testing the system against random node failures. We provide a brief description of the chain replication protocol. Next, we show how we wrote a custom explorer to test for the node failure scenario and found a previously unknown bug in our implementation.

The chain replication protocol [165] is a distributed fault-tolerant protocol for replicating state machines. Consider an instance of a chain replication system with 6 machines—4 instances of `Server` machine ($S_1, \ldots, S_4$) connected in a chain, 1 instance of `Master` machine (M), and 1 instance of `Fault` machine (F). $S_1, \ldots, S_4$ communicate with each other to implement replication. M periodically monitors the health of $S_1, \ldots, S_4$ to detect if any of them has failed. If it detects a fault in S_i, it tells the neighbors of S_i to reconfigure. F is a machine that models fault injection. It maintains a set of numbers initialized to $\{1, \ldots, 4\}$. F repeatedly and nondeterministically removes a number i from this set and sends a failure message to S_i until the size of the set becomes 1. The chain replication protocol is expected to behave correctly for N servers as long as at most $N - 1$ fail.

When a distributed system starts up, there is an initialization phase involving the exchange of messages between nodes for setting up the network topology and other system configuration. Bugs during the initialization phase are straight forward, infrequent, and get discovered quickly. Subtle bugs are generally encountered after the system is initialized and has reached an interesting global state. Since we want to test our system against a specific scenario of failure occurring after the system has stabilized, the new delaying explorer should not spend much time injecting failures or monitoring the system during the initialization phase. We need stratification that gives less priority to particular interleaving in the initial phase.

To capture this intuition with a delaying explorer, we wrote a customized delaying explorer (*CustExplorer*). The explorer maintains the ordering of all dynamically-created machine and cycles through them based on the ordering. The program can change the ordering by invoking *ChangeOrder* callbacks (implemented using `Step`). Using *ChangeOrder* callback in the initialization phase, the machines $S_1, \ldots, S_4$ are ordered before machines M and F. After the initialization phase, the machines M and F are moved ahead in the ordering as compared to machines $S_1, \ldots, S_4$. Thus, *CustExplorer* helps in stratifying the search by giving less priority to interleaving the failure and monitor machines until the system has stabilized.

Using *CustExplorer*, we were able to find a previously unknown bug in chain replication, which occurred when the failure was injected simultaneously at two neighboring nodes after the initialization phase. *CustExplorer* was able to find the bug with SES by exploring 220103 states and with SS by exploring 193442 schedules. We applied the same strategy to ChainRep_6 as it had a similar bug related to node failure, and we were able to find the bug in 10445 states which is nearly 4 times faster than the next best.

4.5 RELATED WORK

Model checking [104, 195] is a classic technique applied to prove temporal properties on programs whose semantics is an arbitrary state-transition graph. Our use of state caching to prune search is inspired by model checking. Partial-order reduction [84] is another technique to prune search. Combining partial-order reduction with schedule prioritization techniques are known to be a challenging problem [144]. Coons et al. [41] have proposed a technique to combine preemption-bounding with partial-order reduction. In future work, we would like to investigate the feasibility of combining delayed exploration with partial-order reduction.

There is prior work on a random sampling of concurrent executions. Sen [173] provides an algorithm for sampling partially-ordered multithreaded executions. Similar to our work, the PCT algorithm [31] also exploits prioritization techniques to sample multithreaded executions adequately. The PCT algorithm characterizes a concurrency bug according to its depth and guarantees that the probability of finding a bug with depth d in a program with L steps and n threads is at least $1/nL^{d-1}$. The mathematical techniques underlying PCT and our sampling algorithm are different. PCT provides a custom algorithm for a particular notion of bug depth whose definition has a deep connection with the proof for the probability bound. On the other hand, our algorithm does not depend on the characterization of bugs. Instead, it is parameterized by a delaying explorer, a mechanism used by the programmer to stratify the search space. Consequently, the proof for our probability bound is a straightforward combinatorial argument on a bounded tree in terms of its branching factor and depth.

Predictive testing [174, 183, 197, 198] follows the basic recipe of executing the program, collecting information from the execution, constructing a model of the program from the collected information, and then re-executing the program based on new predicted interleavings likely to reveal errors. The various techniques differ in the information collected and the targeted class of errors. The search performed by predictive techniques is goal-driven but typically does not provide coverage guarantees. On the other hand, our search technique is not goal-driven but provides coverage guarantees.

CONCURRIT [70] proposes a domain specific language for writing debugging scripts that help the tester specify thread schedules for reproducing concurrency bugs. The

script guides the search without any prioritization. In contrast, our work is focused on finding rather than reproducing bugs. Instead of a debugging script, a tester writes a domain-specific scheduler with appropriate uses of sealing; iterative deepening with delays automatically prioritizes the search with respect to the given scheduler.

4.6 SUMMARY

We have demonstrated how delaying explorers help in scalable systematic testing of P programs. We also showed that using delay bounding [71] with a single default scheduler is not scalable for finding bugs. Different delaying explorers induce different stratification, and hence, writing custom delaying explorers as unit test strategies can make testing complex asynchronous protocols scalable. We also presented and evaluated two algorithms, (1) SES for exhaustive search with strong coverage guarantees and showed how state-caching can be used efficiently for pruning, (2) SS for sampling executions with probabilistic guarantees. We evaluated both these algorithms on real implementation of distributed protocols and showed that our techniques perform orders of magnitude better than state-of-art search prioritization techniques like preemption bounding and PCT.

5

VERIFYING ALMOST-SYNCHRONOUS EVENT-DRIVEN SYSTEMS USING APPROXIMATE SYNCHRONY ABSTRACTION

Forms of synchrony can greatly simplify modeling, design, and verification of distributed systems. The Time-Triggered Architecture (TTA) [167] provides an infrastructure for safety-critical systems of the kind used in autonomous robots, modern cars, and airplanes, and is more recently also being used for building high-performance industrial distributed systems [42]. Traditionally, a common sense of time is established using *time-synchronization* (*clock-synchronization*) protocols or systems such as the global positioning system (GPS), network time protocol (NTP), and the IEEE 1588 [68] precision time protocol (PTP). These protocols, however, synchronize the distributed clocks only within a certain bound. In other words, at any time point, clocks of different nodes can have different values, but time synchronization ensures that those values are within a specified offset of each other, i.e., they are *almost synchronized*.

In Chapter 4, we introduced *delaying-explorers* based search prioritization for scalable systematic testing of asynchronous distributed systems. In this chapter, we consider the problem of verification or systematic testing of "*almost-synchronous*" systems that are neither completely asynchronous or synchronous. Distributed protocols running on top of time-synchronized nodes are designed under the assumption that while processes at different nodes make independent progress, no process falls very far behind any other. Figure 5.1 provides examples of such real-world systems. For example, *Google Spanner* [42] is a distributed fault tolerant system that provides consistency guarantees when running on top of nodes that are synchronized using GPS and atomic clocks, wireless sensor networks [186, 188] use time synchronized channel hopping (TSCH) [1] as a standard for time synchronization of sensor nodes in the network, and IEEE 1588 precision time protocol (PTP) [68] has been adopted in industrial automation, scientific measurement [126], and telecommunication networks. The correctness of these protocols depends on having some synchrony between different processes or nodes.

When modeling and verifying systems that are almost-synchronous, it is essential to compose them using the right concurrency model. One requires a model that lies somewhere between completely synchronous (lock-step progress) and completely asynchronous (unbounded delay). Various such concurrency models have been pro-

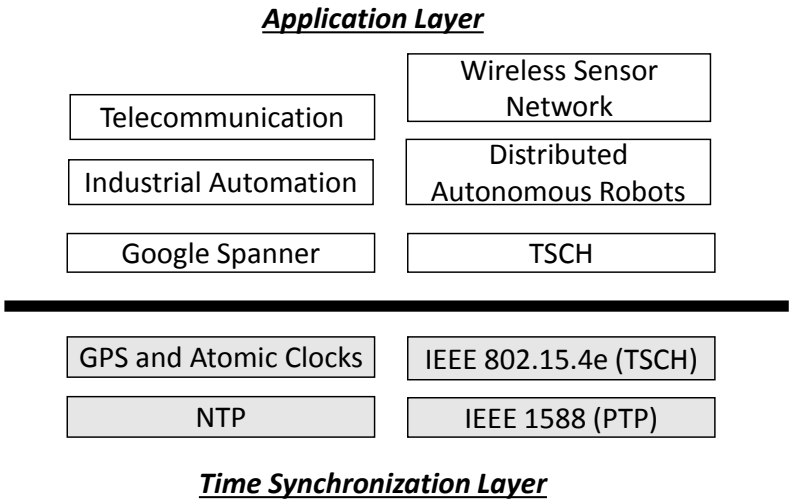


Figure 5.1: Almost-synchronous systems comprise an application protocol running on top of a time-synchronization layer.

posed in the literature, including *quasi-synchrony* [34, 92] and *bounded-asynchrony* [76]. However, as discussed in Section 5.5, these models permit behaviors that are typically disallowed in almost-synchronous systems. Alternatively, one can use formalism for hybrid or timed systems that explicitly model clocks (e.g., [8, 10]), but the associated methods (e.g., [81, 120]) tend to be less efficient for systems with a huge discrete state space, which is typical for distributed software systems.

For modeling and verification of such applications, we introduce *symmetric, almost-synchronous* (SAS) systems, a class of distributed systems in which processes have symmetric timing behavior. In our experience, protocols at both the application layer and the time-synchronization layer (Figure 5.1) can be modeled as SAS systems. Additionally, we introduce the notion of *approximate synchrony (*AS*)* as a concurrency model for almost-synchronous systems, which also enables one to compute a sound discrete abstraction of a SAS system. Intuitively, a system is approximately-synchronous if the number of steps taken by any two processes does not differ by more than a specified bound, denoted Δ. The presence of the parameter Δ makes approximate synchrony a *tunable* abstraction method.

We demonstrate three different uses of the approximate synchrony abstraction:

1. ***Verifying time-synchronized systems:*** Suppose that the system to be verified runs on top of a layer that guarantees time synchronization throughout its execution. In this case, we show that there is a sound value of Δ, which can be computed using a closed-form equation as described in Section 5.2.2.

2. ***Verifying systems with recurrent logical behavior:*** Suppose the system to be verified does not rely on time synchronization, but its traces contain recurrent logical conditions — a set of global states that are visited repeatedly during the protocol's operation. We show that an iterative approach based on model checking can identify such recurrent behavior and extract a value of Δ that can be used to compute a sound discrete abstraction for model checking (see Section 5.2.5). Protocols verifiable with this approach include some at the time-synchronization layer, such as IEEE 1588 [68].

3. ***Prioritizing state-space exploration:*** The approximate synchrony abstraction can also be used as a search prioritization technique for model checking. We show in Section 5.4 that in most cases it is more efficient to search behaviors for a smaller value of Δ ("more synchronous" behaviors) first for finding bugs.

We present two practical case studies: (i) a time-synchronized channel hopping (TSCH) protocol that is part of the IEEE802.15.4e [1] standard, and (ii) the best master clock (BMC) algorithm of the IEEE 1588 precision time protocol. The former is a system where the nodes are time-synchronized, while the latter is the case of a system with recurrent logical behavior. We implemented these systems/protocols in P and extended the P explorer to implement approximate synchrony abstraction. Our results show that approximate synchrony can reduce the state space to be explored by orders of magnitude while modeling relevant timing semantics of these protocols, allowing one to verify properties that cannot be verified otherwise. Moreover, we were able to find a so-called "rogue frame" scenario that the IEEE 1588 standards committee had long debated without resolution (see our companion paper written for the IEEE 1588 community [30] for details).

The Approximate Synchrony abstraction technique can be used with any finite-state model checker. We implemented it on top of P's Systematic Testing backend (Chapter 4), due to its ability to control the search using an external scheduler that enforces the approximate synchrony condition.

To summarize, we make the following contributions:

- The formalism of *symmetric, almost synchronous* (SAS) systems and its use in modeling an important class of distributed systems (Section 5.1.2);
- A tunable abstraction technique, termed *approximate synchrony* (Section 5.2);
- Automatic procedures to derive values of Δ for sound verification (Section 5.2.2 and Section 5.2.5);
- An implementation of approximate synchrony in the P explorer (Section 5.3), and

- The use of approximate synchrony for verification and systematic testing of two real-world protocols, the BMC algorithm (a key component of the IEEE 1588 standard), and the time synchronized channel hopping protocol (Section 5.4).

5.1 ALMOST-SYNCHRONOUS SYSTEMS

In this section, we define clock synchronization precisely and formalize the notion of *symmetric almost-synchronous* (SAS) systems.

5.1.1 *Clocks and Synchronization*

Each node in the distributed system has an associated (local) physical clock χ, which takes a non-negative real value. For purposes of modeling and analysis, we will also assume the presence of an ideal (global) reference clock, denoted t. The notation $\chi(t)$ denotes the value of χ when the reference clock has value t. Given this notation, we describe the following two basic concepts:

1. ***Clock Skew:*** The *skew* between two clocks χ_i and χ_j at time t (according to the reference clock) is the difference in their values $|\chi_i(t) - \chi_j(t)|$.
2. ***Clock Drift:*** The *drift* in the rate of a clock χ is the difference per unit time of the value of χ from the ideal reference clock t.

Time synchronization ensures that the skew between any two physical clocks in the network is bounded. The formal definition is as below.

Definition 5.1.1: Time-Synchronized Systems

A distributed system is time-synchronized *(or* clock-synchronized*) if there exists a parameter β such that for every pair of nodes i and j and for any t,*

$$|\chi_i(t) - \chi_j(t)| \leqslant \beta \tag{5.1}$$

For ease of exposition, we will not explicitly model the details of dynamics of physical clocks or the updates to them. We will instead abstract the clock dynamics as comprising arbitrary updates to χ_i variables subject to additional constraints on them such as Equation 5.1 (wherever such assumptions are imposed).

Example 5.1.1: IEEE 1588 Precision Time Protocol

The IEEE 1588 precision time protocol [68] can be implemented to bound the physical clock skew to the order of sub-nanoseconds [126], and the typical clock drifts to at most 10^{-4} [68].

5.1.2 *Symmetric, Almost-Synchronous Systems*

We model the distributed system as a collection of processes, where processes are used to model both the behavior of nodes as well as of communication channels. There can be one or more processes executing at a node.

Formally, the system is modeled as the tuple $\mathcal{M}_C = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathrm{Id}, \vec{\chi}, \vec{\tau})$ where

- $\mathcal{S}$ is the set of discrete states of the system,
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation for the system,
- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states,
- $\mathrm{Id} = \{1, 2, \ldots, k\}$ is the set of process identifiers,
- $\vec{\chi} = (\chi_1, \chi_2, \ldots, \chi_k)$ is a vector of local clocks, and
- $\vec{\tau} = (\tau_1, \tau_2, \ldots, \tau_k)$ is a vector of process timetables. The timetable of the ith process, τ_i, is an infinite vector $(\tau_i^1, \tau_i^2, \tau_i^3, \ldots)$ specifying the time instants according to local clock χ_i when process i executes (steps). [1] In other words, process i makes its jth step when $\chi_i = \tau_i^j$.

For convenience, we will denote the ith process by $\mathcal{P}_i$. Since in practice the dynamics of physical clocks can be fairly intricate, we choose not to model these details — instead, we assume that the value of a physical clock χ_i can vary arbitrarily subject to additional constraints (e.g., Equation 5.1).

The kth *nominal step size* of process $\mathcal{P}_i$ is the intended interval between the $(k-1)$th and kth steps of $\mathcal{P}_i$, viz., $\tau_i^k - \tau_i^{k-1}$. The *actual step size* of the process is the actual time elapsed between the $(k-1)$th and kth step, according to the ideal reference clock t. In general, the latter differs from the former due to clock drift, scheduling jitter, etc.

Motivated by our case studies with the IEEE 1588 and 802.15.4e standards, we impose two restrictions on the class of systems considered:

1. ***Common Timetable:*** For any two processes $\mathcal{P}_i$ and $\mathcal{P}_j$, $\tau_i = \tau_j$. Note that this does *not* mean that the process steps synchronously, since their local clocks may report different values at the same time t. However, if the system is time synchronized, then the processes step "almost synchronously."

2. ***Bounded Process Step Size:*** For any process $\mathcal{P}_i$, its actual step size lies in an interval $[\sigma^l, \sigma^u]$. This interval is the same for all processes. This restriction arises in practice from the bounded drift of physical clocks.

1 We make the simplifying the assumption that all processes make their initial step when their local clock is at 0. The results also apply to the case when the process timetables do not start at 0.

A set of processes obeying the above restrictions is termed a *symmetric, almost-synchronous* (SAS) system. The adjective "symmetric" refers only to the timing behavior — note that the logical the behavior of different processes can be very different. Note also that SAS systems may or may not be running on top of a time synchronization layer, i.e., SAS systems and time-synchronized systems are orthogonal concepts.

Example 5.1.2: Nodes in the IEEE 1588 system are Almost Synchronous

The IEEE 1588 protocol can be modeled as a SAS system. All processes intend to step at regular intervals called the announce time interval*. The specification [68] states the nominal step size for all processess as* 1 *second; thus the timetable is the sequence* $(0, 1, 2, 3, \ldots)$*. However, due to the drift of clocks and other non-idealities such as jitters due to OS scheduling, the step size in typical IEEE 1588 implementations can vary by* $\pm 10^{-3}$*. From this, the actual step size of processes can be derived to lie in the interval* $[0.999, 1.001]$*.*

Traces and Segments: A *timed trace* (or simply *trace*) of the SAS system $\mathcal{M}_C$ is a timestamped record of the execution of the system according to the global (ideal) time reference t. Formally, a timed trace is a sequence $h_0, h_1, h_2, \ldots$ where each element h_j is a triple $(s_j, \vec{\chi}_j, t_j)$ where $s_j \in \mathcal{S}$ is a discrete (global) state at time $t = t_j$ and $\vec{\chi}_j = (\chi_{1,j}, \chi_{2,j}, \ldots, \chi_{k,j})$ is the vector of clock values at time t_j. For all j, at least one process makes a step at time t_j, so there exists at least one i and a corresponding $m_i \in \{0, 1, 2, \ldots\}$ such that $\chi_{i,j}(t_j) = \tau_i^{m_i}$. Moreover, processes step according to their timetables; thus, if any $\mathcal{P}_i$ makes its m_ith and l_ith steps at times t_j and t_k respectively, for $m_i < l_i$, then $\chi_{i,j}(t_j) = \tau_i^{m_i} < \tau_i^{l_i} = \chi_{i,k}(t_k)$. Also, by the bounded process step size restriction, if any $\mathcal{P}_i$ makes its m_ith and $m_i + 1$th steps at times t_j and t_k respectively (for all m_i), $|t_k - t_j| \in [\sigma^l, \sigma^u]$. Finally, $s_0 \in \mathcal{I}$ and $\mathcal{T}(s_j, s_{j+1})$ holds for all $j \geqslant 0$ with the transition into s_j occurring at time $t = t_j$.
A *trace segment* is a (contiguous) sub-sequence $h_j, h_{j+1}, \ldots, h_l$ of a trace of $\mathcal{M}_C$.

5.1.3 *Verification Problem and Approach*

The central problem considered in this chapter is as follows:

Problem 5.1.1: Verification of SAS Systems

Given an SAS system $\mathcal{M}_C$ *modeled as above, and a linear temporal logic (LTL) [160] property* Φ *with propositions over the discrete states of* $\mathcal{M}_C$*, verify whether* $\mathcal{M}_C$ *satisfies* Φ*.*

One way to model $\mathcal{M}_C$ would be as a hybrid system [122] (due to the continuous dynamics of physical clocks), but this approach does not scale well due to the huge discrete state space. Instead, we provide a sound discrete abstraction $\mathcal{M}_A$ of $\mathcal{M}_C$

that preserves the relevant timing semantics of the almost-synchronous systems. (Soundness is formalized in Section 5.2).

There are two phases in our approach:

1. ***Compute Abstraction Parameter:*** Using parameters of $\mathcal{M}_C$ (relating to clock dynamics), we compute a parameter Δ characterizing the "approximate synchrony" condition, and use Δ to generate a sound abstract model $\mathcal{M}_A$.

2. ***Model Checking:*** We verify the temporal logic property Φ on the abstract model using finite-state model checking.

The key to this strategy is the first step, which is the focus of the following sections.

5.2 APPROXIMATE SYNCHRONY ABSTRACTION

We now formalize the concept of *approximate synchrony (AS)* and explain how it can be used to generate a discrete abstraction of almost-synchronous distributed systems. Approximate synchrony applies to both (segments of) traces and to systems.

Definition 5.2.1: Approximate Synchrony for SAS Traces

A trace (segment) of a SAS system $\mathcal{M}_C$ is said to satisfy approximate synchrony *(is* approximately-synchronous*) with parameter Δ if, for any two processes $\mathcal{P}_i$ and $\mathcal{P}_j$ in $\mathcal{M}_C$, the number of steps N_i and N_j taken by the two processes in that trace (segment) satisfies the following condition:*

$$|N_i - N_j| \leqslant \Delta$$

Although this definition is in terms of traces of SAS systems, we believe the notion of approximate synchrony is more generally applicable to other distributed systems also. An early version of this definition appeared in [54].

The definition extends to a SAS system in the standard way:

Definition 5.2.2: Approximate Synchrony for SAS Systems

A SAS system $\mathcal{M}_C$ satisfies approximate synchrony *(is* approximately-synchronous*) with parameter Δ if* all traces *of that system satisfy approximate synchrony with parameter Δ.*

We refer to the condition in Definition 5.2.1 above as the *approximate synchrony (AS) condition* with parameter Δ, denoted AS(Δ). For example, in Figure 5.2, executing step 5 of process *P1* before step 3 of process *P2* violates the approximate synchrony condition for $\Delta = 2$. Note that Δ quantifies the "approximation" in approximate synchrony. For example, for a (perfectly) synchronous system $\Delta = 0$, since processes

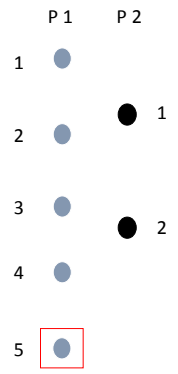


Figure 5.2: Approximate Synchrony condition violated for $\Delta = 2$

step at the same time instants. For a fully asynchronous system, $\Delta = \infty$, since one process can get arbitrarily ahead of another.

5.2.1 *Discrete Approximate Synchrony Abstraction*

We now present a discrete abstraction of a SAS system. The key modification is to (i) remove the physical clocks and timetables, and (ii) include instead an explicit scheduler that constrains execution of processes to satisfy the approximate synchrony condition AS(Δ).

Formally, given a SAS system $\mathcal{M}_C = (\mathcal{S}, \mathcal{T}, \mathcal{I}, Id, \vec{\chi}, \vec{\tau})$, we construct an Δ-*abstract model* $\mathcal{M}_A$ as the tuple $(\mathcal{S}, \mathcal{T}^a, \mathcal{I}, Id, \rho_\Delta)$ where ρ_Δ is a scheduler process that performs an asynchronous composition of the processes $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k$ while enforcing AS(Δ). Conceptually, the scheduler ρ_Δ maintains state counts N_i of the numbers of steps taken by each process $\hat{\mathcal{P}}_i$ from the initial state. The inclusion of step counts may seem to make the model infinite-state. We will show in Section 5.3 how the model checker can be implemented without explicitly including the step counts in the state space. A configuration of $\mathcal{M}_A$ is a pair (s, N) where $s \in \mathcal{S}$ and $N \in \mathbb{N}^k$ is the vector of step counts of the processes. The abstract model $\mathcal{M}_A$ changes its configuration according to its transition function $\mathcal{T}^a$ where $\mathcal{T}^a((s, N), (s', N'))$ iff (i) $\mathcal{T}(s, s')$ and (ii) $N'_i = N_i + 1$ if ρ_Δ permits $\mathcal{P}_i$ to make a step and $N'_i = N_i$ otherwise.

In an initial state, all processes $\mathcal{P}_i$ are enabled to make a step. At each step of $\mathcal{T}^a$, ρ_Δ enforces the approximate synchrony condition by only enabling $\mathcal{P}_i$ to step iff that step does not violate AS(Δ). Behaviors of $\mathcal{M}_A$ are *untimed traces*, i.e., sequences of discrete (global) states $s_0, s_1, s_2, \ldots$ where $s_j \in \mathcal{S}$, s_0 is an initial (global) state, and each transition from s_j to s_{j+1} is consistent with $\mathcal{T}^a$ defined above.

Note that approximate synchrony is a *tunable timing abstraction*. Larger the value of Δ, more conservative the abstraction. The key question is: for a given system, what value of Δ constitutes a *sound* timing abstraction, and how do we automatically

compute it? Recall that one model is a sound abstraction of another if and only if every execution trace of the latter (concrete model $\mathcal{M}_C$) is also an execution trace of the former (abstract model $\mathcal{M}_A$). In our setting, the Δ-abstract and concrete models both capture the protocol logic in an identical manner and differ only in their timing semantics. The concrete model explicitly models the physical clocks of each process as real-valued variables as described in Section 5.1.2. The executions of this model can be represented as *timed traces* (sequences of timestamped states). On the other hand, in the Δ-abstract model, processes are interleaved asynchronously while respecting the approximate synchrony condition stated in Definition 5.2.2. Execution of the Δ-abstract model is an *untimed trace* (sequences of states). We equate timed and untimed traces using the "untiming" transformation proposed by Alur and Dill [8] — i.e., the traces must be identical with respect to the discrete states.

5.2.2 *Computing Δ for Time-Synchronized Systems*

We now address the question of computing a value of Δ such that the resulting $\mathcal{M}_A$ is a sound abstraction of the original SAS system $\mathcal{M}_C$. We consider here the case when $\mathcal{M}_C$ is a system running on a layer that guarantees time synchronization (Equation 5.1) from the initial state. A second case, when nodes are not time-synchronized, and approximate synchrony only holds for segments of the traces of a system, is handled in Section 5.2.5.

Consider a SAS system in which the physical clocks are always synchronized to within β, i.e., Equation 5.1 holds for all time t and β is a tight bound computed based on the system configuration. Intuitively, if $\beta > 0$, then $\Delta \geqslant 1$ since two processes are not guaranteed to step at the same time instants, and so the number of steps of two processes can be off by at least one. The main result of this section is that SAS systems that are time-synchronized are also approximately-synchronous, and the value of Δ is given by the following theorem.

Theorem 5.2.1: Approximate Synchrony for Time-Synchronized Systems

Any SAS system $\mathcal{M}_C$ satisfying Equation 5.1 is approximately-synchronous with parameter $\Delta = \left\lceil \frac{\beta}{\sigma^l} \right\rceil$.

Proof. Consider two arbitrary processes $\mathcal{P}_i$ and $\mathcal{P}_j$. We show that it is always the case that $|N_i - N_j| \leqslant \lceil \frac{\beta}{\sigma^l} \rceil$.

Consider an arbitrary time point t according to an ideal time reference. Without loss of generality, assume $N_i(t) > N_j(t)$ (i.e., that $\mathcal{P}_i$ has made more steps than $\mathcal{P}_j$) and that $\mathcal{P}_j$ has performed a step at time t. We seek to bound the number of additional steps that $\mathcal{P}_i$ has made over $\mathcal{P}_j$.

By the "Common Timetable" assumption, $\mathcal{P}_i$ and $\mathcal{P}_j$ step at the same values of their respective clocks. Therefore, it must be the case that $\chi_i > \chi_j$. Further, due to time synchronization, we also have $\chi_i - \chi_j \leqslant \beta$. Also, the step size of $\mathcal{P}_i$ is bounded below by σ^l. Thus, the number of additional steps $\mathcal{P}_i$ could have taken at time t over $\mathcal{P}_j$ is bounded above by

$$\lceil \frac{\chi_i - \chi_j}{\sigma^l} \rceil \leqslant \lceil \frac{\beta}{\sigma^l} \rceil$$

Thus, $|N_i - N_j| \leqslant \lceil \frac{\beta}{\sigma^l} \rceil$ at time t, for any t. This yield the desired value of Δ. ■

Suppose the abstract model $\mathcal{M}_A$ is constructed as described in Section 5.2.1 with Δ as given in Theorem 5.2.1 and σ^l is the lower bound of the step size defined in Section 5.1.2. Then as a corollary, we can conclude that $\mathcal{M}_A$ is a sound abstraction of $\mathcal{M}_C$: every trace of $\mathcal{M}_C$ satisfies AS(Δ) and hence is a trace of $\mathcal{M}_A$ after untiming.

Example 5.2.1: Time-Synchronized Channel Hopping Protocol

The Time-Synchronized Channel Hopping (TSCH) [1] protocol is being adopted as a part of the low power Medium Access Control standard IEEE802.15.4e. It can be modeled as a SAS system since it has a time-slotted architecture where processes share the same timetable for making steps. The TSCH protocol has two components: one that operates at the application layer, and one that provides time synchronization, with the former relying upon the latter. We verify the application layer of TSCH that assumes that nodes in the system are always time-synchronized within a bound called the "guard time" which corresponds to β. Moreover, in practice, β is much smaller than σ^l and thus Δ is typically 1 for implementations of the TSCH.

This is the case for the TSCH protocol (more details in Technical Report [54]).

TSCH protocol could be modeled as a SAS system; it has time-slotted architecture that is captured using the *common timetable* formalism in the SAS system. Approximate synchrony could accurately capture the notion of nodes in the wireless sensor network using TSCH being time-synchronized [2]. Using this abstraction, we verified the sub-part of TSCH (at the application layer) that helps in maintaining synchronization and low power operation.

5.2.3 *Systems with Recurrent Logical Conditions*

We now consider the case of a SAS system that does not execute on top of a layer that guarantees time synchronization (i.e., Equation 5.1 may not hold). We identify behavior of certain SAS systems, called *recurrent logical conditions*, that can be exploited

2 TSCH has time-slotted architecture moreover, because of time synchronization difference between the slot numbers at different nodes (steps) is bounded

for abstraction and verification. Specifically, even though AS(Δ) may not hold for the system for any finite Δ, it may still hold for *segments* of every trace of the system.

Definition 5.2.3: Recurrent Logical Condition

For a SAS system $\mathcal{M}_C$*, a* recurrent logical condition *is a predicate* logicConv *on the state of* $\mathcal{M}_C$ *such that* $\mathcal{M}_C$ *satisfies the LTL property* **G F** logicConv*.*

Our verification approach is based on finding a finite Δ such that, for every trace of $\mathcal{M}_C$, segments of the trace between states satisfying logicConv satisfy AS(Δ). This property of system traces can then be exploited for efficient model checking.

An example of such a SAS system is the *best master clock* (BMC) algorithm, a vital component of the IEEE 1588 time-synchronization protocol. The BMC algorithm makes no assumptions about the clocks at various nodes being synchronized. However, its operation has a unique structure, comprising two phases. In the first phase, nodes in the system execute a distributed algorithm to agree on a stable network configuration (e.g., spanning tree). This stable configuration is then used in the second phase to synchronize the physical clocks. We refer to this agreement on a stable configuration in the first phase as *logical convergence*. Formally it is represented as a predicate logicConv on the global state of the system. We show in this section that, if logical convergence holds in every trace of a system in a *recurrent* fashion, then one can compute a finite Δ for segments of the trace between states satisfying logicConv. Put another way, the only traces of the system are those in which, between states satisfying logicConv, the processes obey AS(Δ). This property of system traces can then be exploited for efficient model checking.

We begin with an example of a recurrent logical condition case in the context of the IEEE 1588 protocol (Section 5.2.4). We then present our verification approach based on inferring Δ for trace segments via iterative use of model checking (Section 5.2.5).

5.2.4 *Example: IEEE 1588 protocol*

The IEEE 1588 standard [68], also known as the *precision time protocol (PTP)*, enables precise synchronization of clocks over a network. The protocol consists of two parts: the *best master clock (BMC)* algorithm and a *time synchronization phase*. The BMC algorithm is a distributed algorithm whose purpose is two-fold: (i) to elect a unique *grandmaster clock* that is the best clock in the network, and (ii) to find a unique *spanning tree* in the network with the grandmaster clock at the root of the tree. The combination of a grandmaster clock and a spanning tree constitutes the global stable configuration known as *logical convergence* that corresponds to the recurrent logical condition. The second phase, the time synchronization phase uses this stable configuration to synchronize or correct the physical clocks (more details in [68]).

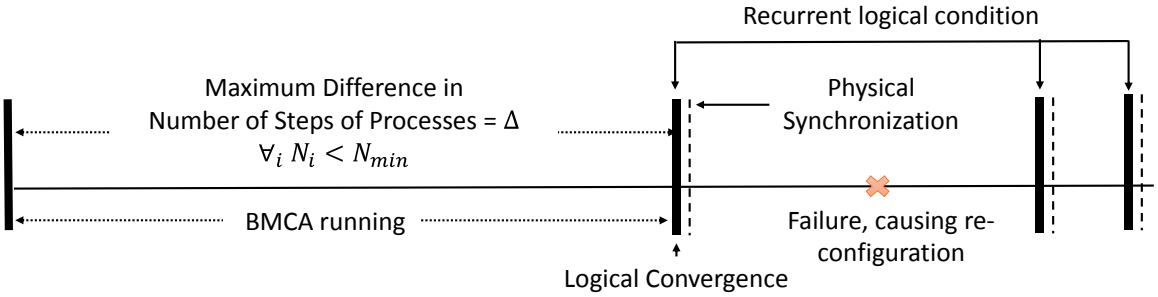


Figure 5.3: Phases of the IEEE 1588 time-synchronization protocol

Figure 5.3 gives an overview of the phases of the IEEE 1588 protocol execution. The distributed system starts executing the first phase (e.g., the BMC algorithm) from an initial configuration. Initially, the clocks are not guaranteed to be synchronized to within a bound β. However, once logical convergence occurs, the clocks are synchronized shortly after that. Once the clocks have been synchronized, it is possible for a failure at a node or link to break clock synchronization. The BMC algorithm operates continually, to ensure that, if time synchronization is broken, the clocks will be re-synchronized. Thus, a typical 1588 protocol execution is structured as a (potentially infinite) repetition of the two phases: logical convergence, followed by clock synchronization. We exploit this *recurrent* structure to show in Section 5.2.5 that we can compute a value of Δ obeyed by segments of any trace of the system. The approach operates by iterative model checking of a specially-crafted temporal logic formula.

Note that the time taken by the protocol to logically converge depends on various factors, including network topology and clock drift. In Section 5.4, we demonstrate empirically that the value of Δ depends on the number of steps (length of the segment) taken by BMCA to converge which in turn depends on factors mentioned above.

5.2.5 *Iterative Algorithm to Compute Δ-Abstraction for Verification*

Given a SAS system $\mathcal{M}_C$ whose traces have a recurrent structure, and an LTL property Φ, we present the following approach to verify whether $\mathcal{M}_C$ satisfies Φ:

1. ***Define recurrent condition:*** Guess a *recurrent logical condition*, $\mathtt{logicConv}$, on the global state of $\mathcal{M}_C$.

2. ***Compute*** N_{min}***:*** Guess an initial value of Δ, and compute, from parameters σ^l, σ^u of the processes in $\mathcal{M}_C$, a number N_{min} such that the AS(Δ) condition is satisfied on all trace segments where no process makes N_{min} or more steps. We describe the computation of N_{min} in more detail below.

3. ***Verify if Δ is sound:*** Verify using model checking on $\mathcal{M}_A$ that, every trace segment that starts in an initial state or a state satisfying $\mathtt{logicConv}$ and ends in another state in $\mathtt{logicConv}$ satisfies AS(Δ). This is done by checking that no process makes N_{min} or more steps in any such segment. Note that verifying $\mathcal{M}_A$ in place of $\mathcal{M}_C$ is sound as AS(Δ) is obeyed for up to N_{min} steps from *any* state. Further details, including the LTL property checked, are provided below.

4. ***Verify $\mathcal{M}_C$ using Δ:*** If the verification in the preceding step succeeds, then a model checker can verify Φ on a discrete abstraction $\widehat{\mathcal{M}}_A$ of $\mathcal{M}_C$, which, similar to $\mathcal{M}_A$, is obtained by dropping physical clocks and timetables and enforcing the AS(Δ) condition to segments *between visits to* $\mathtt{logicConv}$. Formally, $\widehat{\mathcal{M}}_A = (\mathcal{S}, \widehat{\mathcal{T}^a}, \mathcal{I}, \mathrm{Id}, \rho_\Delta)$ where $\widehat{\mathcal{T}^a}$ differs from $\mathcal{T}^a$ only in that for a configuration (s, N), $N'_i = 0$ for all i if $s' \in \mathtt{logicConv}$ (otherwise it is identical to $\mathcal{T}^a$). However, if the verification in Step 3 fails, we go back to Step 2 and increment Δ and repeat the process to compute a sound value of Δ.

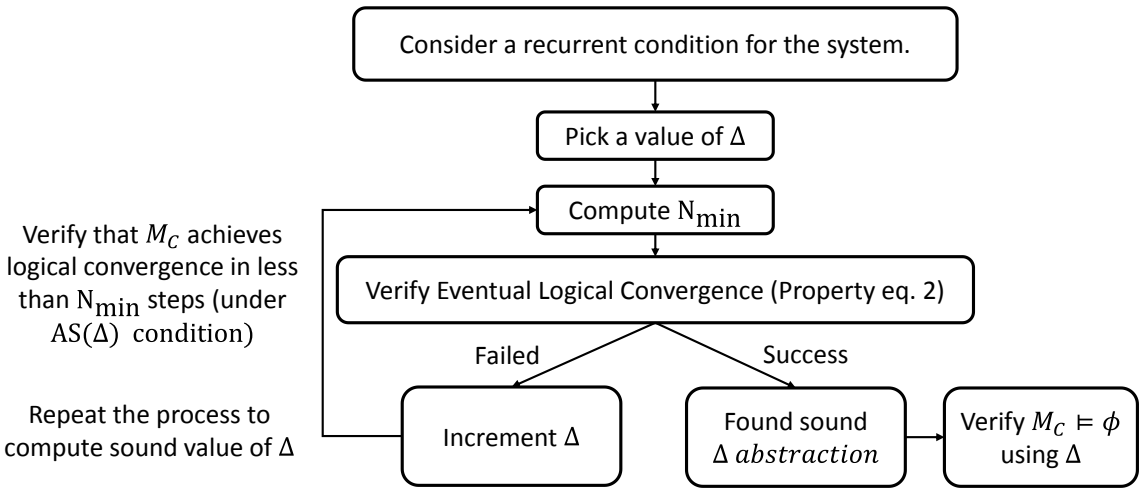


Figure 5.4: Iterative algorithm for computing Δ exploiting logical convergence

Figure 5.4 depicts this iterative approach for the specific case of the BMC algorithm. We now elaborate on Steps 2 and 3 of the approach.

Step 2: Computing N_{min} for a given Δ. Recall from Section 5.1.2 that the actual step size of a process lies in the interval $[\sigma^l, \sigma^u]$. Let $\mathcal{P}_f$ be the fastest process (the one that makes the most steps from the initial state), and $\mathcal{P}_s$ be the slowest (the fewest steps). Denote the corresponding number of steps by N_f and N_s respectively. Then the approximate synchrony condition in Definition 5.2.2 is always satisfied if $N_f - N_s \leqslant \Delta$. We wish to find the smallest number of steps taken by the fastest process when AS(Δ)

is violated. We denote this value as N_{min} and obtain it by formulating and solving a linear program.

Suppose first that $\mathcal{P}_s$ and $\mathcal{P}_f$ begin stepping at the same time t. Then, since the time between steps of $\mathcal{P}_f$ is at least σ^l and that between steps of $\mathcal{P}_s$ is at most σ^u, the total elapsed must be at least $\sigma_l N_f$ and at most $\sigma^u N_s$, yielding the inequality $\sigma^l N_f \leqslant \sigma^u N_s$.

However, processes need not begin making steps simultaneously. Since each process must make its first step at least σ^u seconds into its execution, the maximum initial offset between processes is σ^u. The smallest value of N_f occurs when the fast process starts σ^u time units after the slowest one, yielding the inequality:

$$\sigma^l N_f + \sigma^u \leqslant \sigma^u N_s$$

We can now set up the following integer linear program (ILP) to solve for N_{min}:

$$\min N_f \ \text{ s.t.}$$
$$N_f \geqslant N_s, \quad N_f - N_s > \Delta, \quad \sigma^l N_f + \sigma^u \leqslant \sigma^u N_s, \quad N_f, N_s \geqslant 1$$

N_{min} is the optimal value of this ILP. In effect, it gives the fewest steps any process can take (smallest N_f) to violate the approximate synchrony condition $AS(\Delta)$.

Example 5.2.2: Computing N_{min} for IEEE 1588

For the IEEE 1588 protocol, as described in Section 5.1.2, the actual process step sizes lie in $[0.999, 1.001]$. *Setting* $\Delta = 1$, *solving the above ILP yields* $N_{min} = 1502$.

Step 3: Temporal Logic Property. Once N_{min} is computed, we verify on the discrete abstraction $\mathcal{M}_A$ whether, from any state satisfying $\mathcal{I} \vee \mathrm{logicConv}$, the model reaches a state satisfying $\mathrm{logicConv}$ in less than N_{min} steps. This also verifies that all traces in the BMC algorithm satisfy the recurrent $\mathrm{logicConv}$ property and the segments between $\mathrm{logicConv}$ satisfy $AS(\Delta)$. We perform this by invoking a model checker to verify the following LTL property, which references the variables N_i recording the number of steps of process $\mathcal{P}_i$:

$$(\mathcal{I} \vee \mathrm{logicConv}) \implies \mathbf{F}\left[\mathrm{logicConv} \wedge \left(\bigwedge_i (0 < N_i < N_{min})\right)\right] \tag{5.2}$$

We show in Section 5.3 how to implement the above check without explicitly including the N_i variables in the system state. Note that it suffices to verify the above property on the discrete abstraction $\mathcal{M}_A$ constrained by the scheduler ρ_Δ because we explore no more than N_{min} steps of any process and so $\mathcal{M}_A$ is a sound abstraction. The overall soundness result is formalized below.

Theorem 5.2.2: Soundness of Approximate Synchrony Abstraction

If the abstract model $\mathcal{M}_A$ satisfies Property 5.2, then all traces of the concrete model $\mathcal{M}_C$ are traces of the model $\widehat{\mathcal{M}}_A$ (after untiming)

Proof. From the computation of N_{min} we know that if, in any trace segment, no process makes N_{min} or more steps, then that trace segment satisfies AS(Δ). In particular, this applies to every trace of the concrete model $\mathcal{M}_C$.

Since $\mathcal{M}_A$ satisfies Property 5.2, every segment of a trace of $\mathcal{M}_A$ starting in a state satisfying $\mathcal{I} \vee \mathrm{logicCon}\nu$ must reach another state in $\mathrm{logicCon}\nu$ before any process makes N_{min} steps. In other words, every trace of $\mathcal{M}_A$ has the form

$$s_0, s_1, s_2, \ldots, s_{i_1}, \ldots, s_{i_2}, \ldots, s_{i_3}, \ldots$$

where $s_0 \in \mathcal{I}$ and $s_{i_j} \in \mathrm{logicCon}\nu$ for all j, and furthermore, during the trace segments between states s_0, s_{i_1}, s_{i_2} etc., no process makes N_{min} or more steps.

We now argue that this type of recurrent behavior is also present in traces of $\mathcal{M}_C$. Let us hypothesize that, to the contrary, there is a trace of $\mathcal{M}_C$ with a prefix of the form $(s_0, \vec{\chi}_0, t_0), (s_1, \vec{\chi}_1, t_1), (s_2, \vec{\chi}_2, t_2), \ldots, (s_k, \vec{\chi}_k, t_k)$ where $s_0 \in \mathcal{I}$, $s_i \notin \mathrm{logicCon}\nu$ for any i, and some process makes its N_{min}th step with the transition into s_k. Note that the untimed prefix $s_0, s_1, s_2, \ldots, s_{k-1}$ is a valid prefix of some trace of $\mathcal{M}_A$, since no process has made N_{min} or more steps, and hence AS(Δ) holds. However, we know that $\mathcal{M}_A$ satisfies Property 5.2, which implies that some state s_i, $i = 0, 1, \ldots, k-1$ must be in $\mathrm{logicCon}\nu$. This contradicts our hypothesis. Similar reasoning also applies to a hypothesized trace with a suffix that starts in a state in $\mathrm{logicCon}\nu$ rather than $\mathcal{I}$. Altogether these imply that all traces of $\mathcal{M}_C$ must visit a state in $\mathrm{logicCon}\nu$ infinitely often with no process making N_{min} or more steps between visits. By construction of $\widehat{\mathcal{M}}_A$, the untiming of each of these traces is a trace of $\widehat{\mathcal{M}}_A$, from which the theorem follows. ■

5.3 MODEL CHECKING WITH APPROXIMATE SYNCHRONY

We implemented approximate synchrony within the P systematic testing backend (Chapter 4). It performs a "constrained" asynchronous composition of processes, using an external scheduler to guide the interleaving. Approximate synchrony is enforced by an external scheduler that explores only those traces satisfying AS(Δ) by scheduling, in each state, only those processes whose steps will not violate AS(Δ). Section 5.2.5 described an iterative approach to verify whether a Δ-abstract model of a protocol is sound. The soundness proof depends on verifying Property 5.2. A naïve approach for checking this property would be to include a local variable N_i in each process as part of the process state to keep track of the number of steps executed by each process,

causing state space explosion. Instead, we store the values of N_i for each i external to the system state, as a part of the model checker explorer.

Algorithm 5.3.1 Verification of Property 5.2

```
1: var StateTable : Dictionary⟨S, List⟨int⟩⟩
2: function BoundedDFS(s : S)
3:     var i : int, s′ : State, steps′ : List⟨int⟩
4:     i ← 0
5:     while i < #Processes(s)
6:         steps′ ← IncElement(i, StateTable[s]))
7:         if ¬ CheckASCond(steps′) ∨ steps′[i] > (N_min + Δ) ∨ s ⊨ logicConv then
8:             continue
9:         end if
10:        s′ ← NextState(s, i)
11:        if steps′[i] = N_min then
12:            assert(s′ ⊨ logicConv)
13:        end if
14:        if s′ ∉ Domain(StateTable) ∨ ¬(steps′ ⩾_pt StateTable[s′]) then
15:            StateTable[s′] ← steps′
16:            BoundedDFS(s′)
17:        end if
18:        i ← i + 1
19:    end
20: end function
21:
22: function Verify
23:    StateTable[s_initial] ← newList⟨int⟩
24:    BoundedDFS(s_initial)
25: end function
```

The Algorithm 5.3.1 performs a systematic bounded depth-first search for a state $s_{initial}$, belonging to the set of all possible initial states. To check whether all traces of length N_{min} satisfy eventual logical convergence under AS(Δ) constraint, we enforce two bounds: first, the final depth bound is $(N_{min} + \Delta)$ and second, in each state a process is enabled only if executing that process does not violate AS(Δ). If a state satisfies $logicConv$ then we terminate the search along that path.

The BoundedDFS function is called recursively on each successor state and it explore only those traces that satisfy AS(Δ). If the steps executed by a process is N_{min} then the $logicConv$ monitor is invoked to assert if $s' \models logicConv$ (i.e. we have reached logical convergence state) and if the assertion fails we increment the value of Δ as described in Section 5.2.5. N_{min} and Δ values are derived as explained in Section 5.2.5.

$StateTable$ is a map from reachable state to the tuple of steps with which it was last explored. $steps'$ is the vector of number of steps executed by each process and is stored as a list of integers. #Processes(s) returns the number of enabled processes in the state s. IncElement(i, t) increments the i^{th} element of tuple t and returns the updated tuple. CheckASCond$(steps')$ checks the following condition that $\forall s_1, s_2 \in steps'\ |s_1 - s_2| \leqslant \Delta$.

To avoid re-exploring a state which may not lead to new states, we do not re-explore a state if it is revisited with $steps'$ greater than what it was last visited with. The operator $\geqslant_{pt}$ does a pointwise comparison of the integer tuples. We show in the following section that we can obtain significant state space reduction using this implementation.

5.4 EVALUATION

In this section, we present our empirical evaluation of the approximate synchrony abstraction, guided by the following goals:
(Goal 1) Verify two real-world standards protocols: **(1)** the best master clock algorithm in IEEE 1588 and **(2)** the time synchronized channel hopping protocol in IEEE 802.15.4e.
(Goal 2) Evaluate if we can verify properties that cannot be verified with full asynchrony (either by reducing state space or by capturing relevant timing constraints).
(Goal 3) Evaluate approximate synchrony as an iterative bounding technique for finding bugs efficiently in almost-synchronous systems.

5.4.1 *Modeling and Experimental Setup*

Both the case studies, the BMC algorithm, and the TSCH protocol are modeled in the P language. Each node in the protocol is modeled as a separate P state machine. Faults and message losses in the protocol are modeled as non-deterministic choices. The LTL properties were implemented as monitors that are synchronously composed with the model.

All experiments were performed on a 64-bit Windows server with Intel Xeon ES-2440, 2.40GHz (12 cores/24 threads) and 160 GB of memory. P explorer can exploit parallelism as its iterative depth-first search algorithm is completely parallelized. All timing results reported in this section are when the P explorer is run with 24 threads. We use the number of states explored and the time taken to explore them as the comparison metric.

5.4.2 *Verification and Testing using Approximate Synchrony*

We applied approximate synchrony in three different contexts : **(1)** Time synchronized Channel Hopping protocol (*time synchronized system*) **(2)** Best Master Clock Algorithm in IEEE 1588 (*exploiting recurrent logical condition*) **(3)** Approximate Synchrony as a bounding technique for finding bugs.

Protocol	Temporal Property	Description
BMCA	**F G** ($\mathtt{logicConv}$)	Eventually the BMC algorithm stabilizes with a unique spanning tree having the grandmaster at its root. The system is said to be in $\mathtt{logicConv}$ state when the system has converged to the expected spanning tree.
TSCH	$\bigwedge_{i \in n}$ **G**($\neg \mathtt{desynched}_i$)	A node in TSCH is said to be *desynched* - if it fails to synchronize with its master within the threshold period. The desired property of a correct system is that the nodes are always synchronized.

Table 5.1: Temporal properties verified for the case studies

Verification of the TSCH Protocol. Time Synchronized Channel Hopping (TSCH) is a Medium Access Control scheme that enables low power operations in wireless sensor network using time-synchronization. It assumes that the clocks are always time-synchronized within a bound, referred to as the 'guard' time in the standard. The low power operation of the system depends on the sensor nodes being able to maintain synchronization [3] (desynchronization property in Table 5.1). A central server broadcasts the global schedule that instructs each sensor node when to perform operations. Whether the system satisfies the desynchronization property depends on this global schedule, and the standard provides no recommendation on these schedules.

We modeled the TSCH as a SAS system and used Theorem 5.2.1 to calculate the value of Δ [4]. We verified the desynchronization property (Table 5.1) in the presence of failures like message loss, interference in a wireless network, etc. For the experiments, we considered three schedules (1) round-robin: nodes are scheduled in a round robin fashion, (2) shared with random back-off: all the schedule slots are shared, and conflict is resolved using random back-off (3) Priority Scheduler: nodes are assigned fixed priority, and conflict is resolved based on the priority.

3 Nodes losing synchronization may lead to more messages being transmitted which in turn leads to power wastage.

4 For system of nodes under consideration, the maximum clock skew, $\epsilon = 120\mu s$ and nominal step size of $100ms$, the value of $\Delta = 1$

We were able to verify if the property was satisfied for a given topology under the global schedule, and generated a counterexample otherwise (Table 5.2) which helped the TSCH system developers in choosing the right schedules for low power operation. Using sound approximate synchrony abstraction (with $\Delta = 1$), we could accurately capture the "almost synchronous" behavior of the TSCH system.

Verification of BMC Algorithm. The BMC algorithm is a core component of the IEEE 1588 precision time protocol. It is a distributed fault-tolerant protocol where nodes in the system perform operations periodically to converge on a unique hierarchical tree structure, referred to as the *logical convergence* state in Section 5.2.4. Note that the convergence property for BMCA holds *only in the presence of almost synchrony* — it does not guarantee convergence in the presence of unbounded process delay or message delay. Hence, it is essential to verify BMC using the right form of synchrony.

We generated various verification instances by changing the configuration parameters such as the number of nodes, clock characteristics, also, the network topology. The results in Table 5.2 for the BMC algorithm are for 5 and 7 nodes in the network with linear, star, ring, and random topologies. The Δ value used for verification of each of these configurations was derived by using the iterative approach described in Section 5.2.5. The results demonstrate that the value of Δ required to construct the sound abstraction varies depending on network topology, and clock dynamics. Table 5.2 shows the total number of states explored and time taken by the model checker for proving the safety and convergence property (Table 5.1) using the sound Δ-abstract model. Approximate synchrony abstraction is orders of magnitude faster as it explores the reduced state-space. BMCA algorithm satisfies safety invariant even in the presence of complete asynchrony. For demonstrating the efficiency of using approximate synchrony, we also conducted the experiments with complete asynchronous composition, exploring all possible interleaving (for safety properties). The complete asynchronous model is simple to implement but fails to prove the properties for most of the topologies.

An upshot of our approach is that we are the *first* to prove that the BMC algorithm in IEEE 1588 achieves logical convergence to a unique stable state for some interesting configurations. This was possible because of the *sound and tunable* approximate synchrony abstraction. Although experiments with 5/7 nodes may seem small, networks of this size do occur in practice, e.g., in industrial automation where one has small teams of networked robots on a factory floor.

Endlessly circulating (*rogue*) frames in IEEE 1588: The possibility of an endlessly circulating frame in a 1588 network has been debated for a while in the standards committee. Using a formal model of BMC algorithm under approximate synchrony, we were able to reproduce a scenario were rogue frame could occur. Existence of a rogue frame can lead to network congestion or cause the BMC algorithm never to converge. The counterexample was cross-validated using simulation and is described in detail in [30]. It was well received by the IEEE 1588 standards committee and

Verification of BMC Algorithm											
Network Topology (#Nodes)	Safety Property							Convergence Property			
	Fully Asynchronous Model			Model with Approximate Synchrony				Model with Approximate Synchrony			
	States Explored	Time (h:mm)	Property Proved	Δ	States Explored	Time (h:mm)	Property Proved	Δ	States Explored	Time (hh:mm)	Property Proved
Linear(5)	1.2 E+9	7:12	Yes	1	9.5 E+5	0:35	Yes	1	5.3 E+8	6:33	Yes
Star(5)	2.4 E+10	9:40	Yes	1	5.8 E+5	0:54	Yes	1	4.1 E+7	5:10	Yes
Random(5)	9.19 E+9	9:01	Yes	2	5.5 E+6	1:44	Yes	2	1.8 E+9	9:10	Yes
Ring(5)	7.1 E+12*	*	No	1	4.8 E+7	3:44	Yes	1	8 E+9	8:04	Yes
Linear(7)	1.4 E+13*	*	No	1	4.6 E+7	3:05	Yes	1	1.0 E+8	6:21	Yes
Star(7)	1.1 E+13*	*	No	2	3.7 E+8	5:06	Yes	2	3.3 E+10	13:34	Yes
Ring(7)	3.3 E+12*	*	No	2	6.8 E+8	8:04	Yes	2	2.1 E+10	11:11	Yes
Random(6)	1.1 E+13*	*	No	3	5.7 E+9	6:00	Yes	3	1.3 E+10	10:34	Yes
Random(7)	1.1 E+13*	*	No	3	8.1 E+8	7:11	Yes	3	9.9 E+10	10:11	Yes

Verification of TSCH Protocol									
Network Topology (#Nodes)	Round-Robin Scheduler			Shared with CSMA			Priority Scheduler		
	States Explored	Time (h:mm)	Property Satisfied	States Explored	Time (h:mm)	Property Satisfied	States Explored	Time (h:mm)	Property Satisfied
Linear(5)	4.4 E+4	0:20	Yes	1.2 E+2#	0:03	No	2.4E +3#	0:09	No
Random(5)	3.6 E+2#	0:05	No	6.2 E+3#	0:12	No	1.9E +6	0:35	Yes
Mesh(5)	1.7 E+7	4:05	Yes	9.1 E+6	2:01	Yes	9.3 E+5	0:31	Yes

** denotes end of exploration as model checker ran out of memory, # denotes property violated and counter example is reported*

Table 5.2: Verification results using Approximate Synchrony.

Buggy Models	Iterative Depth Bounding with Random Search			Non-Iterative AS			Iterative AS		
	Depth	States Explored	Time (h:mm)	Δ	States Explored	Time (h:mm)	Δ	States Explored	Time (h:mm)
BMCA_Bug_1	51	1.4 E+3	0:05	2	1.1 E+3	0:04	0	2.1 E+2	0:02
BMCA_Bug_2	64	5.9 E+5	0:15	2	6.1 E+4	0:14	0	1.6 E+3	0:04
BMCA_Bug_3	101	9.4 E+7	0:45	3	3.3 E+5	0:17	1	9.1 E+2	0:05
ROGUE_FRAME_Bug_1	44	3.9 E+5	0:18	2	9.7 E+6	0:29	1	5.6 E+4	0:12
ROGUE_FRAME_Bug_2	87	4.4 E+4	0:09	2	2.1 E+3	0:05	1	1.1 E+3	0:03
SPT_Bug_1	121	8.4 E+8	1:05	3	8.1 E+4	0:11	0	5.5 E+2	0:04

Table 5.3: Iterative Approximate Synchrony with bound Δ for finding bugs faster.

acknowledged in the standards report that a rogue frame bug is possible in certain network topologies.

Approximate Synchrony as a Search Prioritization Technique. Approximate synchrony can also be used as a bounding technique to prioritize search. We collected buggy models during the process of modeling the BMC algorithm and used them as benchmarks, along with a buggy instance of the Perlman's Spanning Tree Protocol [154] (SPT). We used AS as an iterative bounding technique, starting with $\Delta = 0$ and incrementing Δ after each iteration. For $\Delta = 0$, the model checker explores only synchronous system behaviors. Increasing the value could be considered as adding bounded asynchronous behaviors incrementally. Table 5.3 shows a comparison between iterative AS, non-iterative AS with a fixed value of Δ taken from Table 5.2 and iterative depth bounding with random search. The number of states explored and the corresponding time taken for finding the bug is used as the comparison metric. Results demonstrate that most of the bugs are found at small values of Δ (hence iterative search is beneficial for finding bugs). Some bugs like the rogue frame error that occurs only when there is asynchrony were found with minimal asynchrony in the system ($\Delta = 1$). These results confirm that prioritizing search based on approximate synchrony is beneficial in finding bugs. Other bounding techniques such as delay bounding [71] and context bounding [145] can be combined with approximate synchrony, but this is left for future work.

5.5 RELATED WORK

The concept of *partial synchrony* has been well-studied in the theory of distributed systems [61, 67, 162]. There are many ways to model partial synchrony depending on the type of system and the end goal (e.g., formal verification). Approximate synchrony is one such approach, which we contrast against the most closely-related work below.

Hybrid/Timed Modeling: The choice of modeling formalism greatly influences the verification approach. A time-synchronized system can be modeled as a hybrid system [10]. However, it is important to note that, unlike traditional hybrid systems examples from the domain of control, the discrete part of the state space for these protocols is very large. Due to this, we observed that leading hybrid systems verification tools, such as SpaceEx [81], cannot explore the entire state space.

There has been work on modeling timed protocols using real-time formalisms such as *timed automata* [8], where the derivatives of all continuous-time variables are equal to one. While tools based on the theory of timed automata do not explicitly support modeling and verification of multi-rate timed systems [120], there do exist techniques for approximating multirate clocks. For instance, Huang *et al.* [107] propose the use of *integer clocks* on top of UPPAAL models. Daws and Yovine [44] show how multi-rate timed systems can be over-approximated into timed automata. Vaandrager and Groot [190] models a clock that can proceed with different rate by defining a clock

model consisting of one location and one self transition. Such models only approximately represent multirate time systems. By contrast, our approach algorithmically constructs abstractions that can be refined to be more precise by tuning the value of Δ, and results in a *sound* untimed model that can be directly checked by a finite-state model checker. Consequently, for the systems we consider, our approach does not suffer from any approximation on integer clocks, and we do not need to resort to advanced real-time model checkers such as UPPAAL.

Synchrony and Asynchrony: There have been numerous efforts devoted towards mixing synchronous and asynchronous modeling. Multiclock Esterel [164] and communicating reactive processes (CRP) [25] extend the synchronous language Esterel to support a mix of synchronous and asynchronous processes. Multiclock Esterel provides language extensions to partition clocks into two categories: those that tick simultaneously and those that can have unbounded skew and drift. In time-synchronized systems, there is a guarantee of a fixed bound which is captured by approximate synchrony but cannot be captured by these abstractions. *Bounded asynchrony* is another such modeling technique with applications to biological systems [76]. It can be used to model systems in which processes can have different but *constant* rates, and can be interleaved asynchronously (with possible stuttering) before they all synchronize at the end of a global "period." Approximate synchrony has no such synchronizing global period. The *quasi-synchronous (QS)* [34, 92] approach is designed for communicating processes that are *periodic* and have almost the *same* period. QS [92] is defined as "Between any two successive activations of one period process, the process on any other process is activated either 0, 1, or at most 2 times". As a consequence, in both quasi-synchrony and bounded asynchrony, the difference of the absolute number of activations of two different processes can grow unboundedly. In contrast, the definition of AS does not allow this difference to grow unbounded.

5.6 SUMMARY

This chapter has introduced two new concepts: a class of distributed systems termed as *symmetric, almost-synchronous* (SAS) systems, and *approximate synchrony*, an abstraction method for such systems. We evaluated applicability of approximate synchrony for verification in two different contexts: (i) application-layer protocols running on top of time-synchronized systems (TSCH), and (ii) systems that do not rely on time synchronization but exhibit recurrent logical behavior (BMC algorithm). We also described an interesting search prioritization technique based on approximate synchrony with the key insight that, prioritizing synchronous behaviors can help in finding bugs faster.

We integrated approximate synchrony based model-checking into the P explorer for validating almost-synchronous systems or protocols implemented using P. In particular, we used an extension of approximate synchrony abstraction for systematically

testing robotics systems that consists of concurrently executing periodic processes (Chapter 7).

Part III

ASSURED AUTONOMY FOR ROBOTICS SYSTEMS

The recent drive towards achieving greater autonomy and intelligence in robotics has led to increasing levels of complexity in the robotics software stack. *Assured autonomy* requires a robot to make correct and timely decisions, where the robotics software stack is implemented as a concurrent, reactive, event-driven system that may also use advanced machine learning-based components. This trend has resulted in a widening gap between the complexity of systems being deployed and those that can be certified for safety and correctness of operation. In Chapter 6, we provide an overview of these challenges and describe the existing approaches and their short-comings.

Our approach towards acheiving *assured autonomy* for robotics systems consists of two parts: **(1)** a *high-level programming language* for implementing and validating the reactive robotics software stack; and **(2)** an integrated *runtime assurance* system to ensure that the assumptions used during design-time validation of the high-level software hold at runtime. Combining high-level programming language and model-checking with runtime assurance helps us bridge the gap between design-time software validation that makes assumptions about the untrusted components (e. g., low-level controllers), and the physical world, and the actual execution of the software on a real robotic platform in the physical world. We implemented our approach in Drona (introduced in Chapter 6), a programming framework for building safe robotics systems.

In Chapter 7, we consider the problem of building *safe distributed mobile robotics system* and describe how Drona can be used for programming and validating the complex multi-robot event-driven software stack. We advocate the use of principles of runtime assurance to ensure the safety of the robotics systems in the presence of untrusted components like third-party libraries or machine learning-based components. In Chapter 8, we present the runtime assurance framework integrated into Drona and demonstrate how it enables guaranteeing the safety of the robotics system, including when untrusted components have bugs or deviate from the desired behavior.

https://drona-org.github.io/Drona/

6

ASSURED AUTONOMY: CHALLENGES AND ADVANCES

- A robot may not injure a human being or, through inaction, allow a human being to come to harm.
- A robot must obey orders given to it by a human being except where such orders would conflict with the first law.
- A robot must protect its own existence as long as such protection does not conflict with the first or second law.

— Isaac Asimov's Three Laws of Robotics

Recent advances in robotics have led to the adoption of *autonomous mobile robots* across a broad spectrum of applications like surveillance, precision agriculture, warehouse, delivery systems, and personal transportation. As autonomous robots are finding applications in complex real-world systems that have acute *safety* and *reliability* requirements, programmability with high assurance and provable robustness guarantees remains a significant barrier to their large-scale adoption.

This drive towards autonomy is also leading to ever-increasing levels of software complexity. This complexity stems from two central requirements: **(1)** event-driven, real-time, concurrent software required for ensuring reactive and safe robotics system, **(2)** integration of advanced data-driven, machine-learning components in the software stack required to enable autonomous decision making in complex environments. Moreover, the dependence of robotic systems on third-party off-the-shelf components and machine-learning techniques is predicted to increase. However, advances in formal verification and systematic testing have yet to catch up with this increased complexity [178]. This has resulted in a widening gap between the complexity of systems being deployed and those that can be certified for safety and correctness.

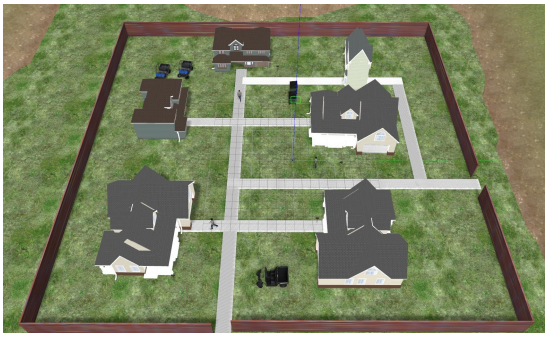
In this chapter, we first provide an overview of our robotics case study: *a safe autonomous drone surveillance system*, and also discuss the corresponding robotics software stack design (Section 6.1). We next highlight the main challenges involved in

building safe autonomous robotics systems (Section 6.2) and finally, present our approach implemented in the DRONA toolchain (Section 6.3) to address these challenges of programming safe reactive robotics software stack (Section 6.2.1) and guaranteeing safety in the presence of untrusted components (Section 6.2.2).

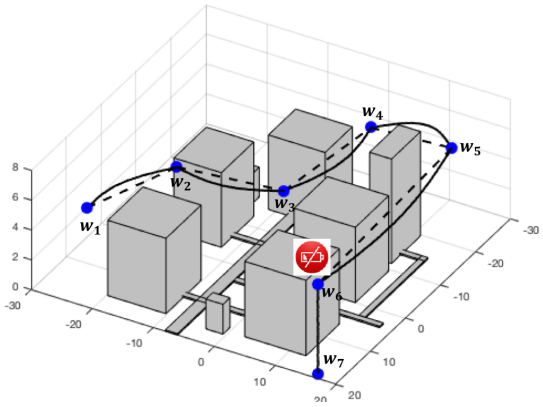
6.1 CASE STUDY: AUTONOMOUS DRONE SURVEILLANCE SYSTEM

In this thesis, we use the unmanned aerial vehicles, also called *drones* as the target robotics platform to highlight both, the challenges in building safe robotics systems and the efficacy of our approach. The approach presented in this thesis, and implemented in the DRONA toolchain is independent of the target robotics platform and is not specific to drones.

We consider an autonomous drone surveillance system where a drone must autonomously patrol a set of locations in a city. Figure 6.1a shows a snapshot of the workspace in the Gazebo simulator [114]. Figure 6.1b presents the obstacle map for the workspace with the surveillance points (dots) and a possible path that the autonomous drone can take when performing the surveillance task (solid trajectory).



(a) A Gazebo [114] workspace for simulating the surveillance mission. The workspace models a city with obstacles like houses, cars, and pedestrians on the streets.



(b) The static obstacle-map for the workspace. The waypoints $w_1 \ldots w_6$ represent a potential motion plan, and the dotted lines represent the reference trajectory for the drone. The solid line represents the actual trajectory of the drone, which deviates from the reference trajectory because of the underlying dynamics and disturbances.

Figure 6.1: Case Study: Autonomous Drone Surveillance System

The surveillance system requires that the autonomous drone must satisfy the following specifications:

(S1) *Sequencing and Coverage* (ϕ_{app}): The drone must visit all surveillance points in a priority order. The surveillance points to be monitored can be added or removed dynamically. Hence, the drone must be capable of handling of dynamically generated tasks. The drone must eventually visit all surveillance points.

(S2) *Collision avoidance* (ϕ_{col}): The drone must never collide with an obstacle.

(S3) *Battery safety* (ϕ_{bat}): The drone must never crash because of low-battery. Instead, when the battery is low it must prioritize either landing safely or visiting a battery charging station.

For our case study, we consider a simplified setting where all the obstacles (houses, cars, etc.) are static, known a priori, and that there are no environment uncertainties like the wind. Even for such a simplified setup, the corresponding robotics software stack (Figure 6.2) is complex: consisting of multiple components interacting with each other and uses uncertified/untrusted components (red blocks).

6.1.1 *Reactive Robotics Software Stack*

At the heart of an autonomous robot is the specialized onboard software that ensures safe operation without any human intervention. Figure 6.2 presents the robotics software stack for an autonomous drone surveillance system. We next briefly introduce each component in the software stack; they are discussed in more detail and formally defined in Section 7.1.1 and Section 8.1.1.

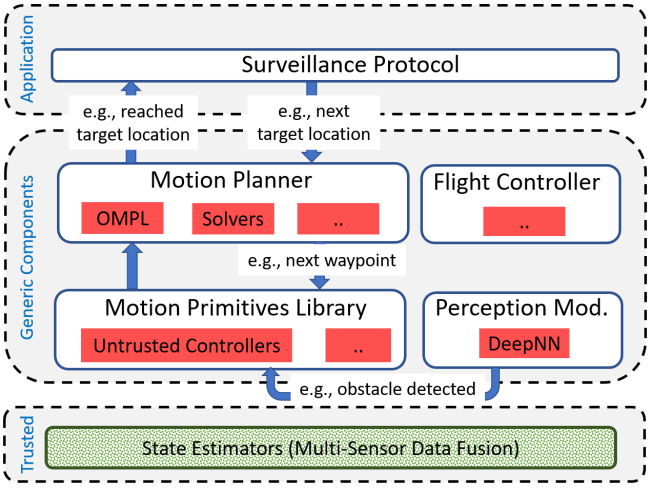


Figure 6.2: Reactive Robotics Software Stack for the Autonomous Drone Surveillance System

1. **Task Planner (Application)**: The *task planner* implements the application-specific protocol, which ensures that the system satisfies the desired application-specific properties.

For example, in our case, the surveillance protocol at the top ensures that the sequence of tasks performed by the drone satisfies the desired system specifications, e. g., the drone must repeatedly visit the surveillance points in fixed priority order. The surveillance protocol generates the sequence of next target locations (tasks) for the drone and sends it to the motion planner.

The rest of the components in the software stack are the *generic components* present in most mobile robotics systems, they together ensure safe movement of the robot in the workspace.

2. **Motion Planner:** The Motion planner [117] solves the navigation problem for a robot by breaking down the desired movement task into discrete motions that satisfy movement constraints and possibly optimize some aspect of the movement.

   The motion planner computes a motion plan, which is a sequence of waypoints from the current location to the target location. The waypoints $w_1 \dots w_6$ in Figure 6.1b represent one such motion plan generated by the planner, and the dotted lines represent the reference trajectory for the drone. The motion planner on receiving a target location from the task planner generates the safe motion plan that does not collide with any obstacle and forwards it to the motion primitives library.

   Implementing an on-the-fly motion planner may involve solving an optimization problem or using an efficient graph search technique that relies on a solver or a third-party library (e. g., OMPL [184]).

3. **Motion Primitives:** Motion primitives are a set of short closed loop trajectories of a robot under the action of a set of precomputed control laws [117, 136]. The set of motion primitives form the basis of the motion for a robot.

   The motion primitives library on receiving the next waypoint generate the required low-level controls necessary to follow the reference trajectory from the current location to the target waypoint. Given the complex dynamics of a robot, noisy sensors, and environmental disturbances, ensuring that the robot precisely follows a fixed trajectory under the influence of a motion primitive is extremely hard. The trajectory in Figure 6.1b represents the actual path of the drone, which deviates from the reference trajectory because of the underlying dynamics and disturbances.

   These motion primitives are either designed using machine-learning techniques like Reinforcement Learning [109], or optimized for specific tasks without considering safety, or are off-the-shelf controllers provided by third parties [151].

4. **Flight Controller:** During a complex autonomous mission, a drone might have to switch between different modes of operation. The flight controller module

implements the switching protocol, which ensures that the critical events are prioritized correctly, and the robot always operates in the correct mode to guarantee over-all safety of the mission. More details about the flight controller are provided in the Section 6.2.1.

5. **Perception Module:** The perception module is responsible for detecting obstacles and passing the information to the planner and controller to avoid a collision. Machine learning techniques, primarily based on Deep Neural Networks [86] have been responsible for the advances in solving the perception problem in autonomous robotics.

6. **State Estimators and Sensors:** State estimation [21] for robotics is the field that deals with the challenge of using onboard sensors and appropriate mathematical tools to estimate the vehicle state (typically the combination of position, velocity, orientation, angular velocity, etc.).

   Most of the robot (drone) manufacturing companies provide a software development kit (SDK) [151] that implements basic primitives for programmatically controlling a robot and estimating its state. We leverage the PX4 [151] and ROS [163][1] SDKs for implementing the state-estimation component of the software stack.

Remark: We assume that the state estimators are *trusted* and can accurately provide the system state within known bounds.

6.2 CHALLENGES IN BUILDING SAFE ROBOTICS SYSTEMS

We would like to re-emphasize two important characteristics of the components in the robotics software stack presented in Figure 6.2:

1. ***Reactive and event-driven:*** Each component implements a complex protocol that involves making discrete-decisions and continuous interactions with other components to ensure that the robot safely achieves its goals. This requires the software to be implemented as a event-driven system.

2. ***Untrusted components:*** These component may depend on *untrusted*[2] software, e. g., the motion primitives library may use third-party libraries that implement closed-loop controllers.

1 The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. http://wiki.ros.org/ROS/Introduction

2 we refer to a software component as untrusted if it is hard to reason about its correctness, e.g. could be third-party libraries or machine-learning based algorithms

These characteristics makes it notoriously hard to provide high-assurance of correctness that the autonomous drone will always satisfy properties **(S1)**-**(S3)**. Next, we discuss these challenges in further details with motivating examples.

6.2.1 *Programming Safe Reactive Event-Driven Robotics Software*

Components in a robotics software stack are generally implemented as concurrent event-driven systems as they must be reactive to inputs from the physical world and from other software components.

To illustrate the complexity and event-driven nature of components in robotics software stack, let us consider the flight controller component which ensures that the robot always operates in correct mode and switches from one mode to another depending on the changes in the state of the system. Figure 6.3 presents an abstract version of the flight controller state machine implemented in the PX4 [151] drone software stack.

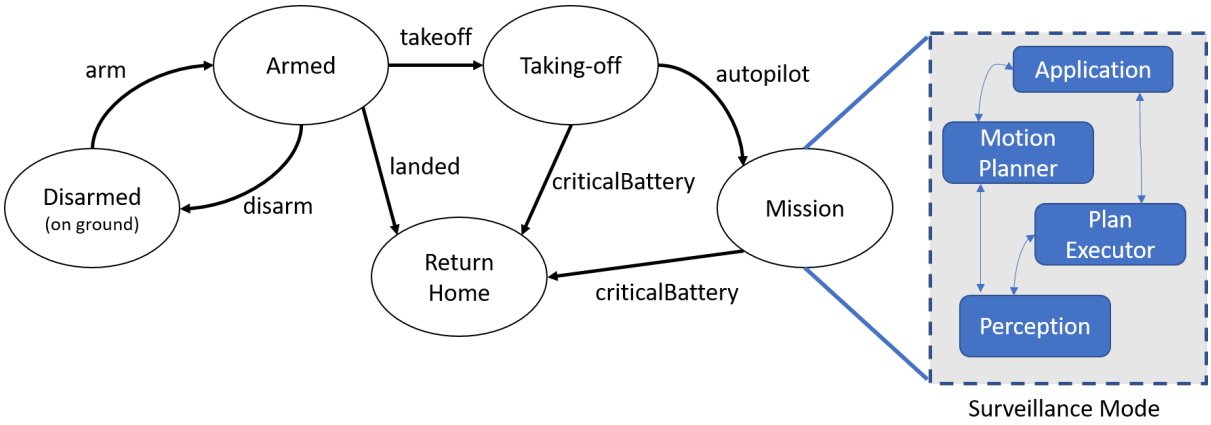


Figure 6.3: Flight Controller Protocol for an Autonomous Drone.

The controller operates in different states (modes) and transition between states based on the events received. An execution of the flight controller can look like: the drone starts in `Disarmed` state; on receiving the `arm` command it moves to `Armed` state where rotors are started; on receiving the `takeoff` command followed by the `autopilot` command, the drone moves to the `Mission` mode where it starts performing the surveillance mission.

In each mode, different components cooperate with the goal of performing the mode-specific operations. For instance, in the mission mode components like application, motion planner, and motion primitives together ensure that the robot safely performs the surveillance mission. Irrespective of the mode of operation, the flight controller must handle critical events that can happen at any time. For example, a `criticalBattery` event must be handled correctly by aborting all operations and

safely returning to home location. For this, the flight controller must always safely transition to the `Return Home` mode which may in turn involve sending an event to all the other components like application, motion planner, and motion primitives so that they coordinate together to land the drone safely. To guarantee battery safety (ϕ_{bat}) the flight controller must satisfy the property that "*globally if* `criticalBattery` *then eventually drone has returned home and landed*"

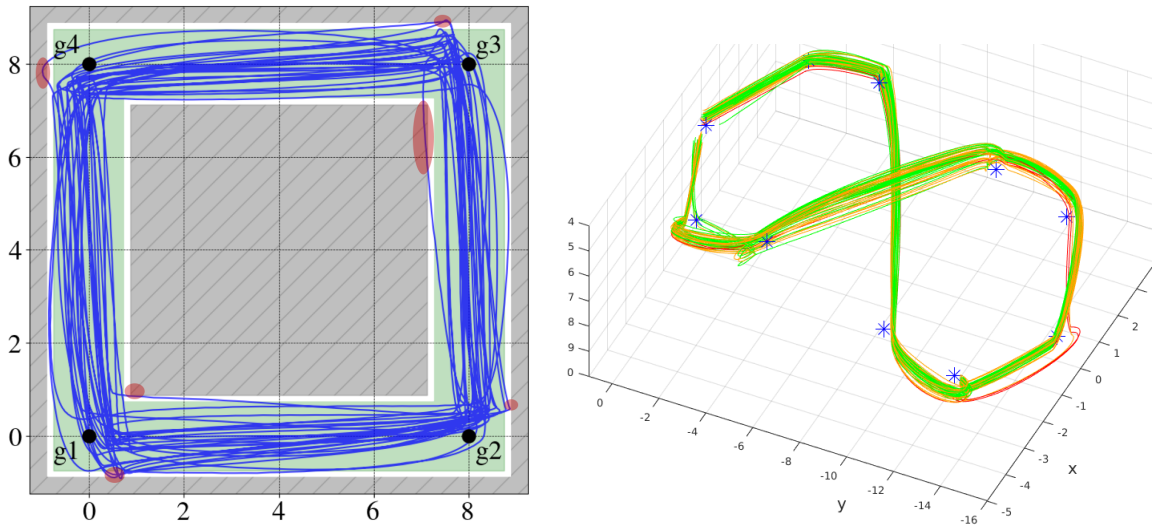
All the components in the robotics software stack implement similar complex protocols. Hence, to provide formal guarantees of correctness for the software stack (properties **(S1)** to **(S3)**), there is a need for a framework that allows implementing, specifying and verifying complex event-driven software.

To demonstrate the value of safe event-driven programming in the context of robotics systems, we modeled the flight controller of PX4 Autopilot [151] in P and found a bug in the protocol (within a few seconds) where the drone could have crashed as the `criticalBattery` event was not handled correctly. This further motivated the need for building a programming framework around P for implementing safe robotics system.

6.2.2 *Guaranteeing Safety in the Presence of Untrusted Components*

As described in Section 6.1, most of the components in the robotics software stack end-up using untrusted software like third-party controllers or modules that are built using data-driven approaches like machine-learning or deep learning. We treat these as untrusted/unsafe since they often exhibit unsafe behavior in off-nominal conditions and uncertain environments, and even when they do not, it is hard to be sure since their complexity makes verification or exhaustive testing prohibitively expensive. Furthermore, the trend in robotics is towards *advanced*, data-driven controllers, such as those based on neural networks (NN), that usually do not come with safety guarantees. To demonstrate that using such untrusted components can lead to failures we conducted two experiments with the motion primitives library in the robotics software stack.

Figure 6.4a presents an experiment where the drone was tasked to repeatedly visit locations g_1 to g_4 in that order, i.e., the sequence of waypoints $g_1, \ldots g_4$ are passed to the motion primitives library. The motion primitives library generates control actions to traverse the reference trajectory from current position to the target waypoint using a low-level controller provided by the third-party robot manufacturer (e.g., we use the PX4 Autopilot [151]). These low-level controllers generally use approximate models of the dynamics of the robot and are optimized for performance rather than safety, making them unsafe. The blue lines represent the actual trajectories of the drone. Given the complex dynamics of a drone and noisy sensors, ensuring that it precisely follows a fixed trajectory (ideally a straight line joining the waypoints) is extremely hard. The low-level controller (untrusted) optimizes for time and, hence, during high



(a) Motion Primitive Library implemented using third-party controller (PX4 Autopilot). The drone was tasked to repeated visit g1 . . . g4 in a tube surrounded by obstacle in gray. The region in red represent cases where the drone deviated too far from the reference trajectory and could have collided with obstacle.

(b) Motion Primitive Library implemented using machine learning based controller. The drone was tasked to repeated visit in a eight-shaped loop (blue stars). The trajectories in red represents cases where the drone deviated dangerously away from the desired trajectory.

Figure 6.4: Experiments with (untrusted) third-party and machine-learning controllers

speed maneuvers the reduced control on the drone leads to overshoot and trajectories that collide with obstacles (represented by the red regions).

We also conducted a similar experiment with a different low-level controller designed using data-driven approach (Figure 6.4b) where we tasked the drone to follow a eight loop. The trajectories in green represent the cases where the drone closely follows loop, the trajectories in red represent the cases the drone dangerously deviates from the reference trajectory. Note that in both cases, the controllers can be used during majority of their mission except for a few instances of unsafe maneuvers.

The key observation we would like to make from these experiments is that the usage of untrusted software can lead to failures, and as complexity of robotics systems increases the dependence on these untrusted components cannot be avoided, hence, we need techniques that ensure safety of the system in the presence of these untrusted components (red block in the Figure 6.2).

6.3 OUR APPROACH: THE DRONA PROGRAMMING FRAMEWORK

Let us revisit the drone surveillance system case study in Section 6.1. We would like the system to satisfy the properties **(S1)** to **(S3)**. These properties involve different

reasoning domains and robot components. For instance, property **(S1)** is application specific and comprise discrete events. Contrarily, properties **(S2)** and **(S3)** are generic (i.e., they should be satisfied by any safe robotics system) and concern both discrete and continuous domains. In particular, property **(S2)** can be further decomposed into two parts: **(S2a)** *Safe Motion Planner*: The motion planner must always generate a motion-plan such that the reference trajectory does not collide with any obstacle, **(S2b)** *Safe Motion Primitives*: When tracking the reference trajectory between any two waypoints generated by the motion planner, the controls generated by the motion primitives must ensure that the drone closely follows the trajectory and avoids collisions. Note that **(S2a)** must be ensured by the discrete motion planner that generates discrete trajectories (i.e., a sequence of waypoints), whereas the property **(S2b)** is dependent on the low-level controllers (continuous). These observations motivate the need for decomposing the verification problem into sub-problems that can be tackled by using the right technique. For instance, traditional programming abstractions and model checking approaches can address property **(S1)**, and it could also be used to reason about the discrete part of the property **(S2)** by making an assumption about the (continuous behavior) property **(S2b)**. And use a different approach to ensure that the assumption **(S2b)** is guaranteed by the system.

Our approach combines discrete modeling (programming), and design-time verification with runtime assurance. We use the modular P programming language framework (Part i) to implement and specify the reactive robotics software stack, and use the systematic testing backend (Part ii) to validate the system (i.e., satisfies the properties **(S1)** to **(S3)**). When validating the software we use discrete abstractions of the components that involve reasoning about dynamics of the robotics or that are hard to analyze like third-party components. We ensure that these assumption (discrete abstractions) made during design-time validation hold at runtime using principles of runtime assurance. We next describe the DRONA tool chain that implements our approach.

The DRONA ***tool-chain*** (Figure 6.5) consists of three main building blocks —

1. ***Reactive programming language:*** The DRONA framework uses the P programming language for implementing and specifying reactive event driven robotics software. DRONA extends the P framework to enable programmers provide details about the robot workspace, like size of the workspace grid, location of static obstacles, location of battery charging points, starting location of each robot, etc. It also provides language primitives to programmatically design systems with runtime assurance architecture (more details in Chapter 8).

2. ***Compiler and Model Checker:*** DRONA extends the P compiler to generates C code that can be deployed on the ROS robotics platform. It also extends the P explorer with approaches for scalable model-checking of robotics software which consists of both event-driven and periodic processes (more details in Chapter 7).

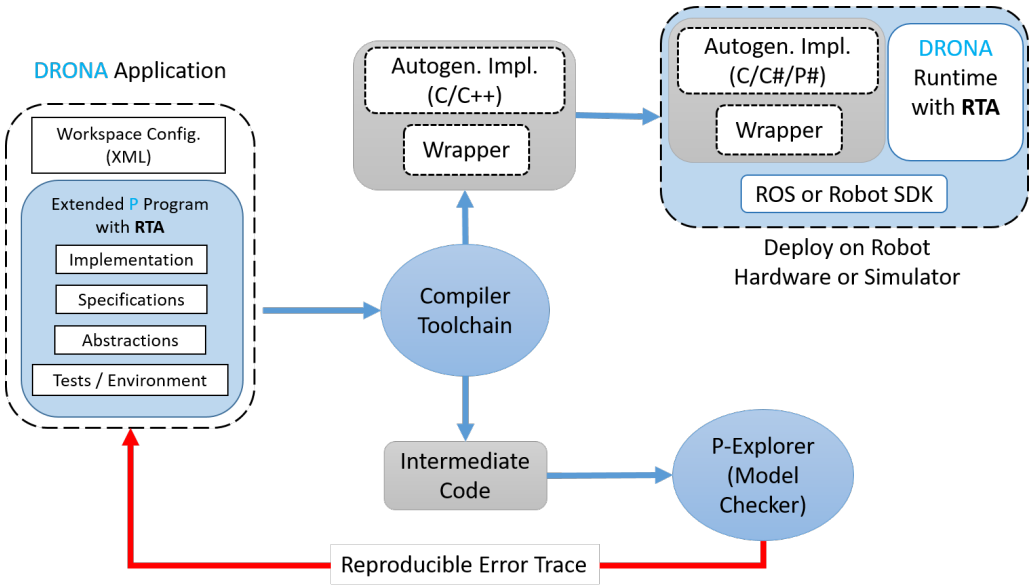


Figure 6.5: DRONA Tool Chain (RTA: Runtime Assurance)

3. ***Robotics Runtime:*** DRONA extends the P runtime with runtime assurance capabilities to safely integrate untrusted components into the robotics software stack. It also implements primitives for efficient communication between robots used for implementing distributed mobile robotics applications (more details in Chapter 7).

6.4 RELATED WORK

There has been a lot of research targeted towards addressing the problem of building robotics systems with high-assurance of correctness [89]. The techniques proposed span across domains, for example: **(1)** using high-level *programming abstractions* to simplify the process building safe robotics systems, **(2)** using *temporal-logic based reactive synthesis* to auto-generate critical parts of the robotics software stack, **(3)** using design-time verification approaches like *reachability analysis* to prove properties about the behavior of the robot (dynamics) under the influence of a software controller, **(4)** using *falsification* approaches to find bugs in the implementation of the robotics software by running it in a loop with a realisitc simulator, **(5)** using *runtime assurance* based approach for guaranteeing safety of the robot by monitoring the state of the robot and enforcing safe operation at runtime. In this section, we first discuss some of the state-of-the-art approaches from each of these domains and show how they all fail to address all the challenges described in Section 6.2. We conclude by presenting how DRONA combines ideas from these domains to be the first framework capable of addressing both the challenges.

Programming Abstractions. The closest work related to DRONA is the recently proposed StarL [125] framework, that unifies programming, specification and verification of distributed robotics systems. Antlab [82] is another end-to-end system that takes streams of user task requests and executes them using collections of robots. The tasks are specified as temporal specifications and the Antlab framework automatically generates and assigns the task optimally to the set of robots in the system. The software stack implemented in the Antlab framework is capable of synthesizing multi-robot motion-plans and performs the required coordination between robots. Programming frameworks like Giotto [98] have been used for building critical distributed embedded systems software. Giotto provides an abstract model for the implementation of periodic software tasks with real-time constraints. All these frameworks provide abstractions that help programmers implement safe mobile robotics systems; framework like StarL also provides backend verification engine to prove correctness of the implementation. But they fail to address the challenge of guaranteeing safety in the presence of untrusted components.

Reactive Synthesis. There is increasing interest towards synthesizing reactive robotics controllers from temporal logic [73, 115, 168, 182]. Tools like TuLip [202], BIP [4, 23], and LTLMoP [75] construct a finite transition system that serves as an abstract model of the physical system and synthesizes a strategy, represented by a finite state automaton, satisfying the given high-level temporal specification. Fly-by-Logic [153] presents an approach to solve the problem of safe multi-quadrotor missions by allowing the programmer to encode these missions using STL (Signal Temporal Logic). Though the generated strategy is guaranteed to be safe in the abstract model of the environment, this approach has limitations: (1) there is gap between the abstract models of the system and its actual behavior in the physical world; (2) there is gap between the generated strategy state-machine and its actual software implementation that interacts with the low-level controllers; and finally (3) the synthesis approach scale poorly both with the complexity of the mission and the size of the workspace. Recent tools such as SMC [182] generate both high-level and low-level plans, but still need additional work to translate these plans into reliable software on top of robotics platforms. All these synthesis-based approaches target *correct-by-construction* strategy for implementing complex controller software, but faces scalability issues when building complex real-world robotics software stack. Also, it cannot handle the challenge of guaranteeing safety in the presence of untrusted components.

Reachability Analysis. Reachability analysis tools [39, 65, 81] have been used to verify robotics systems modeled as hybrid systems. The upshot of this approach is that if the analysis terminates then it provides a proof that the system (or its model) satisfies the desired specifications. Reachability methods require an explicit representation of the robot dynamics and often suffer from scalability issues when the system has a large number of discrete states. The analysis is performed using the models of the system, and hence, there is a gap between the models being verified and their

implementation. Also, it cannot handle the challenge of guaranteeing safety in the presence of untrusted components.

Simulation-based Falsification. Simulation-based tools for the falsification of CPS models (e.g., [62]) are more scalable than reachability methods, but generally, they do not provide any formal guarantees. In this approach, the entire robotics software stack is tested by simulating it in a loop with a high-fidelity model of the robot and hence, this approach does not suffer from the gap between model and implementation described in the previous approaches. However, a challenge to achieving scalable coverage comes from the considerable time it can take for simulations. Also, not that these are falsification based approaches and does not provide any guarantee of correctness, hence, cannot address the two challenges.

Runtime Verification and Assurance. Runtime verification has been applied to robotics [48, 59, 103, 106, 124, 130, 155] where online monitors are used to check the correctness (safety) of the robot at runtime. Schierman *et al.* [171] investigated how a *runtime assurance* framework can be used at different levels of the software stack of an unmanned aircraft system. The idea of using an advanced controller under nominal conditions; while at the boundaries, using optimal safe control to maintain safety has also been used in [6] for operating quadrotors in the real world. In [16] the authors use a switching architecture ([17]) to switch between a nominal safety model and learned performance model to synthesize policies for a quadrotor to follow a trajectory. Similarly, ModelPlex [140] combines offline verification of CPS models with runtime validation of system executions for compliance with the model to build correct by construction runtime monitors which provides correctness guarantees for CPS executions at runtime. Note that most prior applications of RTA do not provide high-level programming language support for constructing provably-safe RTA systems in a compositional fashion while designing for *timing and communication behavior* of such systems. They are all instances of using runtime assurance as a *design methodology* for building reliable systems in the presence of untrusted components.

Our Approach implemented in Drona framework. In order to ease the construction of safe robotics systems, there is a need for a general programming framework that supports run-time assurance principles, and also considers implementation aspects such as timing and communication. As described in Section 6.3 the DRONA framework combines ideas from different domains to address the short-coming of the related work. It integrates a state-machine based programming language for safe event-driven robotics software, leverages advances in scalable systematic-testing techniques for validation of the actual implementation of the software, and, provides language support for runtime assurance to ensure safety in the real physical world.

7

PROGRAMMING SAFE DISTRIBUTED MOBILE ROBOTICS SYSTEMS

There are known knowns. These are things we know that we know. There are known unknowns. That is to say, there are things that we know we don't know. But there are also unknown unknowns. There are things we don't know we don't know.

— Donald Rumsfeld

In Chapter 6, we discussed two challenges that the DRONA framework help address, first being the safe programming of reactive robotics software and second is guaranteeing safety of the system in the presence of untrusted components. In this chapter, we consider the first challenge and demonstrate the efficacy of DRONA (Section 6.3) by taking a principled approach of specifying, implementing, and verifying a *distributed mobile robotics* (DMR) system. The challenges described in the Chapter 6 are amplified for an DMR system as there are multiple robots involved and they have to coordinate with each other continuously to ensure safe mission completion.

When building a reliable DMR software stack using DRONA, we also had to solve the fundamental problem of *safe multi-robot motion planning*. For example, in the multi-robot surveillance system, as surveillance requests are generated in real-time and the drones must move simultaneously in the shared workspace computing collision-free paths on-the-fly. To address this problem, we present a provably correct multi-robot motion planner (MRMP) which is *decentralized*, *asynchronous*, and *reactive* to dynamically generated task requests. Prior work on multi-robot motion planning (e.g., [33, 169, 170, 194, 196]) assumes that the robots in the system step synchronously, or in other words, their local clocks are synchronized. However, in distributed systems, there is no perfect synchrony, and hence, this unsound assumption can lead to motion planner computing colliding trajectories. One of the salient features of MRMP implemented and verified using DRONA is that it does not assume perfect synchrony of the

distributed clocks. It produces safe collision-free trajectories taking into account the "almost synchronized" nature of a time-synchronized DMR system.

We make the following contributions in this chapter:

- We present a novel and provably correct decentralized asynchronous motion planner that can perform on-the-fly collision-free planning for dynamically generated tasks. Moreover, the motion planner is the first to take into account the fact that distributed robots may have clocks that are only synchronized up to a tolerance (Section 7.2). Our results show that the MRMP scales efficiently for systems with a large number of robots (up to 128 robots), and can be used for on-the-fly computation of safe-trajectories in real deployments (Section 7.4).
- A DMR software stack consists of both event-driven asynchronous processes and periodic processes. For verifying a DMR system, we formalize it as a *mixed synchronous* system, present a sound abstraction-based model-checking approach for scalable analysis, and implement it as part of the P systematic testing backend. (Section 7.3).
- We demonstrate the advantages of using DRONA for safe programming and verification of DMR systems by implementing the *multi-drone surveillance system* as a case study. Using DRONA, we found several critical bugs in our implementation and successfully deployed it on real drone platforms (Section 7.4).

In the rest of this chapter, we first briefly describe our DMR case study of a multi-robot surveillance system. We then present our implementation of the DMR software stack, in particular, the novel multi-robot motion planner (Section 7.2) and the abstraction-based approach used for verifying our implementation (Section 7.3). Finally, we present the empirical evaluation of the DRONA to demonstrate its efficacy towards building reliable distributed robotics systems (Section 7.4).

7.1 OVERVIEW

Multi-Robot Surveillance System: Figure 7.1 shows the discretized 2-D grid-map of a city area in which a fleet of drones operates to perform surveillance (similar to the workspace in Figure 6.1). The black blocks represent buildings and are the static obstacles in the workspace. The dotted blocks are battery charging locations that the drones must visit to charge their batteries. In the multi-robot surveillance system, a fixed set of drones shares a known workspace with static obstacles and the surveillance points to be monitored by each drone are generated dynamically; hence, the drones must be capable of performing reactive task-planning. Each drone must, in turn, satisfy the properties **(S1)**-**(S3)** described in Section 6.1. The collision avoidance property **(S2)** is modified to ensure that the drone must not collide with

the static obstacles as well as with other drones moving simultaneously in the shared workspace.

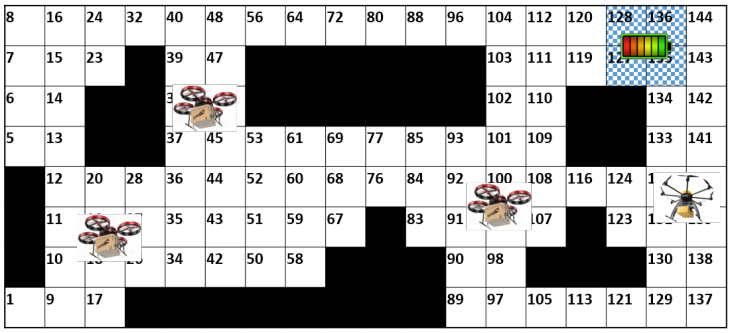


Figure 7.1: Workspace for the Multi-Robot Surveillance System.

Each robot in the DMR system executes the robotics software stack presented in Figure 6.2. To reiterate, at the top is the task-planner (TP) that implements the application specific protocol to guarantee that the system satisfies application-specific goals. For example, the surveillance protocol is responsible for ensuring that the surveillance points are visited eventually and are always in priority order. For the DMR system, the motion planner module in Figure 6.2 consists of two sub-components: a multi-robot motion planner (MRMP) and a plan-executor (PE). The MRMP must not only ensure collision avoidance with static obstacles but also with other robots operating in the workspace. It is the role of the MRMP module to compute safe and collision-free trajectory for the robot by coordinating with other robots in the workspace. The plan-executor module ensures that the robot correctly follows the trajectory computed by the motion planner by invoking the motion-primitives periodically. More details about the dependence of the correctness of the trajectory computed by the MRMP on PE are described in Section 7.2. The rest of the components in the software stack are similar to those described in Figure 6.2.

7.1.1 *Terminology and Definitions*

Workspace: We represent the workspace for a DMR application as a 3-D occupancy grid map, the top view of an example 3-D workspace is shown in Figure 7.1. The grid decomposes the workspace into cube-shaped blocks. The size of a workspace is represented using the number of blocks along each dimension. For example, if the workspace contains n_x, n_y and n_z blocks along the x, y and z dimension respectively, the size of the workspace is represented as $[n_x \times n_y \times n_z]$. Each block is assigned a unique identifier which represents the *location* of that block in the workspace. The set of all locations in the workspace is denoted by the set W. Static obstacles can occupy some parts of the workspace. If an obstacle partially occupies a grid block, we mark

the entire grid block to be covered by an obstacle. The set of locations covered by obstacles is denoted by Ω. The set of free locations in the workspace is denoted by F, where $F = W \setminus \Omega$. The fixed set of robots operating in the workspace is denoted by the set $R = \{r_1, \ldots, r_{|R|}\}$.

Tasks: In a DMR application, tasks can be generated dynamically and assigned to a robot. To complete a task, the robot needs to visit the goal location associated with the task. A task is denoted as the tuple (l, p), where $l \in F$ denotes the goal location where the robot needs to reach for finishing the task, and $p \in \mathbb{N}$ denotes the unique identifier of the task. We denote by T the set of all atomic tasks. A complex task can be represented as a sequence of atomic tasks. We will use the term *task* to refer to an atomic task.

Motion primitives: Motion primitives are a set of short closed-loop trajectories of a robot under the action of a set of precomputed control laws [117, 136]. The set of motion primitives form the basis of the motion for a robot. A robot moves from its current location to a destination location by executing a sequence of motion primitives. We denote by Γ the set of all motion primitives available for a robot. For example, in the most simple case a ground robot has five motion primitives: $\{\texttt{H}, \texttt{L}, \texttt{R}, \texttt{U}, \texttt{D}\}$, where the primitive `H` keeps the robot in the same grid block and the primitives `L`, `R`, `U` and `D` move the robot to the adjacent left, right, upper, and lower grid block respectively.

For a grid location l and a motion primitive $\gamma \in \Gamma$, we denote by $\texttt{post}(l, \gamma)$ the location where the robot moves when the motion primitive γ is applied at l. We use $\texttt{intermediate}(l, \gamma)$ to denote the set of locations through which the robot may traverse after applying γ at location l (including l and $\texttt{post}(l, \gamma)$). For a motion primitive $\gamma \in \Gamma$, we denote by $\texttt{cost}(\gamma)$ the cost (e.g., energy expenditure) to execute the motion primitive. We assume that for all robots in the system, each motion primitive requires τ unit time for execution. This assumption may not hold for heterogeneous systems and extending our approach for such systems is left as future work.

Motion plan: Now we formally define a *motion plan*.

Definition 7.1.1: Motion Plan

A motion plan is defined as a sequence of motion primitives to be applied to a robot r_i *to move from its current location* l_c^i *to a goal location* l_g^i*. A motion plan is denoted by* $\rho_i = (\gamma_1 \ldots \gamma_k)$*, where,* $\gamma_q \in \Gamma$ *for* $q \in \{1, \ldots, k\}$*.*

Timed trajectories: The trajectory of a robot r_i can be represented as a sequence of timestamped locations $(\tau_0^i, l_0^i), (\tau_1^i, l_1^i) \ldots$, where τ_n^i represents the n-th periodic time step for robot r_i. In the rest of the chapter, we refer to (τ_n^i, l_n^i) as l_n^i representing the location of robot r_i in the n-th time step. The size of the period $|\tau_n^i - \tau_{n+1}^i| = \tau$, where τ is the time it takes to execute any motion primitive.

Definition 7.1.2: Trajectory

Given the current location l_c^i of the robot r_i and a motion plan $\rho_i = (\gamma_1 \dots \gamma_k)$ that is applied to the robot at the time step τ_n^i, the trajectory of the robot is a sequence of locations $\xi_i = (l_n^i l_{n+1}^i \dots l_{n+k}^i)$, such that $l_n^i = l_c^i, \forall q \in \{0, \dots, k-1\}$, γ_{q+1} is applied to the robot at location l_{n+q}^i at the time step τ_{n+q}^i and $l_{n+q+1}^i = post(l_{n+q}^i, \gamma_{q+1})$.

Safe task-completion property: The trajectory computed by the motion planner must always satisfy the *safe task-completion* property (Φ_{st}) which is a conjunction of following three properties: (a) obstacle avoidance (ϕ_o), (b) collision avoidance (ϕ_c), and (c) successful task completion (ϕ_f). The property ϕ_o requires that a robot never attempts to pass through a location $l \in \Omega$ associated with a static obstacle. The property ϕ_c entails that two robots never collide with each other. The property ϕ_f captures the requirement that if a robot follows the trajectory, then it will eventually reach the goal location.

7.2 BUILDING DISTRIBUTED MOBILE ROBOTICS (DMR) SYSTEM

We designed and implemented a safe DMR software stack for the distributed surveillance system. We implemented all the components in the software stack using P, and model-checked that the system satisfied the desired specifications **(S1)** to **(S3)**. One of the key component required for ensuring the property **(S2)** of the DMR system is the distributed motion planner that must provide a provably correct solution for the Problem 7.2.1. In this section, we describe our novel distributed asychronous multi-robot motion planner that can compute safe trajectories on-the-fly and also account for the clock synchronization error in a distributed system.

7.2.1 *Distributed Multi-Robot Motion Planner*

We present the multi-robot motion planner (MRMP) implemented in DRONA. MRMP is *asynchronous*, *decentralized*, and robust to clock skew in distributed systems.

Problem 7.2.1: Motion Planning Problem in DMR Systems

Given a set of robots $R = \{r_1, \dots, r_{|R|}\}$ operating in a common workspace W, if a dynamically generated task $(l, p) \in T$ is assigned to a robot $r_i \in R$, find trajectory ξ_i such that it satisfies safe task-completion property Φ_{st}.

We decompose the above motion planning problem into two sub-problems:

1. **Trajectory coordination problem:** For computing the collision-free trajectory of a robot, the motion planner must have consistent information (*consistent snapshot*) about the trajectories of all other robots in the system (Section 7.2.2).

2. **Safe plan-generation problem:** Given the set of current trajectories of all the robots (Ψ), synthesize a trajectory that is robust against time-synchronization errors and satisfies Φ_{st} (Section 7.2.3).

7.2.2 *Distributed Trajectory Coordination*

In a DMR system, tasks are generated dynamically. Hence, the motion planner for such a system should be able to compute trajectories on-the-fly and in a decentralized fashion. The decentralized motion-planner for robot $r_i \in R$ is shown in Algorithm 7.2.1 in the form of a state machine, which is executed by each robot in the system. It is presented in the form of pseudo-code that closely represents the syntax of the P programming language. The function ***broadcast*** (ev, pd) broadcasts event ev with payload pd to all the robots in workspace, including oneself.

The motion-planner state machine has three states: WaitForTaskRequest, CoordinateAndGeneratePlan, and WaitForPlanCompletion. Planner starts executing in the WaitForTaskRequest state. On receiving a `NewTask` event from the task-planner, it updates the task information ($currTask_{id}$ and l^i_g) and moves to the CoordinateAndGeneratePlan state. If the planner receives a `ReqForCurrentTraj` event from another robot $r_j \in R$, it sends its current location l^i_c to robot r_j.

Algorithm 7.2.1 Decentralized Motion Planner

```
machine DecentralizedMotionPlanner {
    start state WaitForTaskRequest {
        entry { l_c^i ←getCurrentLocation() }
        on NewTask (task : T) do {
            currTask_id ← task.id, l_g^i ← task.goal
            goto CoordinateAndGeneratePlan
        }
        on ReqForCurrentTraj (task_id, r_j) do {
            send (r_j, CurrentTraj, (r_i, [l_c^i]))
        }
    }
    state CoordinateAndGeneratePlan {
        entry {
            R_pend ← {} , R_recv ← {}, Ψ_i ← {}
            broadcast (ReqForCurrentTraj, (currTask_id, r_i))
        }
        on CurrentTraj (r_j, ζ_j) do {
            R_recv ← R_recv ∪ {r_j}, Ψ_i ← Ψ_i ∪ {ζ_j}
            if (sizeof(R_recv) = |R|) then
                ρ_i ← synthesizeMotionPlan(l_c^i, l_g^i, Ω, Ψ_i)
                SendMotionPlanToPlanExecutor(ρ_i)
                ξ_i ← ConvertMotionPlanToTraj(ρ_i)
                foreach r_j ∈ R_pend
                    send (r_j, CurrentTraj, (r_i, ξ_i))
                end
                goto WaitForPlanCompletion
            end if
        }
        on ReqForCurrentTraj (task_id, r_j) do {
            if (task_id ⩽ currTask_id) then
                send (r_j, CurrentTraj, (r_i, [l_c^i]))
            else
                R_pend ← R_pend ∪ {r_j}
            end if
        }
    }
    state WaitForPlanCompletion {
        on ReqForCurrentTraj (task_id, r_j) do {
            send (r_j, CurrentTraj, (r_i, ξ_i))
        }
        on Reset () do {
            goto WaitForTaskRequest
        }
    }
}
```

Upon entering the CoordinateAndGeneratePlan state, planner broadcasts `ReqForCurrentTraj` event with the identifier of the current task and its own identifier, asking for trajectories of all robots in the workspace. R_{recv} stores identifiers of the robots that have sent their trajectories as a response to the `ReqForCurrentTraj` event, and Ψ_i stores the current trajectories of all those robots. R_{pend} is used for storing identifiers of all robots from which it has received `ReqForCurrentTraj` and have to send its newly computed trajectory. Upon receiving the `CurrentTraj` event from another robot r_j, the planner adds robot r_j to set R_{recv} and its trajectory ζ_j to the set Ψ_i. The planner state machine is blocked in CoordinateAndGeneratePlan state until it receives `CurrentTraj` event from all the robots.

On receiving trajectories from all the robots (line 19), the planner invokes the `synthesizeMotionPlan` function with its current location l_c^i, the goal location l_g^i, the set of static obstacles Ω and the set of trajectories of all the robots Ψ_i. The implementation of plan generator function `synthesizeMotionPlan` is described in Section 7.2.3. The motion-plan returned by the `synthesizeMotionPlan` function is sent to the plan-executor module so that the robot can start executing it, and the corresponding trajectory is sent to all the robots whose identifiers are present in the set R_{pend} and are blocked waiting for the trajectory of robot r_i.

If two robots r_i and r_j attempt to generate motion plans simultaneously, then a race situation arises as both of them are waiting for the current trajectory of the other robot. This deadlock situation is resolved based on the unique identifier assigned to each task. If the planner of r_i receives a `ReqForCurrentTraj` event from r_j in the CoordinateAndGeneratePlan state and if the task identifier $task_{id}$ in the event is less than its current task identifier $currTask_{id}$ then it implies that the robot r_j is dealing with a higher priority task. In such a case, the motion planner of r_i sends its current location l_c^i to the motion planner of r_j to unblock it and waits for r_j's computed trajectory. Otherwise, it adds the robot r_j to the set R_{pend}, and once it computes its own trajectory, sends the trajectory to unblock r_j (Line 23-25).

In the WaitForPlanCompletion state, motion planner waits for a `Reset` event from the plan-executor indicating that the task is completed, on receiving which it moves to WaitForTaskRequest.

Notice that if the planner for robot r_i generates trajectory ξ_i, then ξ_i is always safe as the coordination protocol guarantees that all future trajectories computed by any other robot r_j will have ξ_i in Ψ_j.

7.2.3 *Safe Plan Generator*

In this section, we present an approach for synthesizing a motion plan to generate a trajectory that satisfies the safe task-completion property Φ_{st}.

7.2.3.1 *Motion Plan Synthesis Problem*

The inputs to the motion plan synthesis problem (Algorithm 7.2.1, line 20) for a robot r_i is the current location of the robot (l_c^i), the goal location (l_g^i), the set of static obstacles (Ω), and the set of current trajectories of other robots (Ψ_i). We call the tuple $\mathcal{P}_i = \langle l_c^i, l_g^i, \Omega, \Psi_i \rangle$ as the *motion plan synthesis problem* instance for robot r_i.

Recall that a trajectory ξ_i of robot r_i is a sequence of locations $(l_n^i, l_{n+1}^i, \ldots, l_{n+k}^i)$, where the trajectory starts at the n-th time step. We adopt a technique based on the composition of motion primitives [169, 170] to solve the motion-plan synthesis problem. To generate such a trajectory ξ_i, we must synthesize a motion plan (Definition 7.1.1) $\rho_i = (\gamma_1, \gamma_2, \ldots, \gamma_k)$, where $\gamma_q \in \Gamma$, $1 \leqslant q \leqslant k$. Recollect that the desired trajectory (Definition 7.1.2) is realized by applying motion primitive γ_{q+1} to the robot at time step τ_{n+q}^i.

We now define the motion plan synthesis problem:

Problem 7.2.2: Safe Motion Plan Synthesis

Given a motion plan synthesis problem instance $\mathcal{P}_i$ for robot r_i, a set of motion primitives Γ, and the time step τ_n^i when the plan executor will start executing the motion plan, synthesize a motion plan $\rho_i = (\gamma_1 \ldots \gamma_k)$ such that the trajectory $\xi_i = (l_n^i l_{n+1}^i \ldots l_{n+k}^i)$ generated by the plan executor by executing the motion plan ρ_i satisfies the safe task-completion property Φ_{st}.

Accounting for clock skew: Each robot $r_i \in R$ operates based on its own local clock χ_i. Let t denote an ideal global time reference (just for purposes of formalization). We denote by $\chi_i(t)$ the valuation of the clock χ_i at the global time t. Synchronization of these clocks plays an important role in the correctness of our distributed motion planning algorithm with respect to the collision avoidance property ϕ_c.

We assume that the DMR software stack implements a time-synchronization protocol [68] that bounds the *skew* between two clocks, given by $|\chi_i(t) - \chi_j(t)| \leqslant \beta$. If $\beta = 0$, we say that the clocks of the robots are in *perfect synchrony*. Otherwise, the clocks are almost-synchronous with precision $\beta > 0$.

To capture the skew between timed trajectories of two robots, we reuse the *approximate synchrony* condition introduced in Chapter 5.

Theorem 7.2.1: Using Approximate Synchrony Condition

If the local clocks of robots r_i and r_j are time-synchronized with a synchronization precision β, and at some global time point t, if robot r_i takes the time step τ_p^i and robot r_j takes the time step τ_q^j, then $|p - q| \leqslant \Delta$, where Δ is given by $\Delta = \left\lceil \frac{\beta}{\tau} \right\rceil$ where τ is the duration of a time step (Theorem 5.2.1).

Informally, Theorem 7.2.3.1 states that if the clocks of two robots are synchronized within a bound β, then the difference between the number of periodic steps taken by the two robots is bounded by Δ. Hence, for collision avoidance, while synthesizing motion plan it is important to know precisely where the other robots in the system would be for a time-step window of size $\pm\Delta$. The parameter Δ determines how conservative a robot should be when computing its trajectory that avoids collision with other robots.

7.2.3.2 *Motion Plan Generation*

We now describe how a motion plan $\rho_i = (\gamma_1, \ldots, \gamma_k)$ is synthesized from a motion plan synthesis problem instance $\mathcal{P}_i = \langle l_c^i, l_g^i, \Omega, \Psi \rangle$. We formulate the problem as an optimization problem where the decision variables are the motion primitives to be applied at different time steps, and the objective is to minimize the total cost to execute the trajectory. The functions `post`, `cost`, and `intermediate` used in this section are defined in Section 7.1.1.
The objective function is given as follows:

$$\underset{(\gamma_1,\gamma_2,\ldots,\gamma_k)}{\texttt{minimize}} \sum_{j=1}^{k} \texttt{cost}(\gamma_j) \tag{7.1}$$

The constraints for the optimization problem is a conjunction of four constraints as described below:
(1) Initial and final location: The first location in ξ_i is the current location, l_c^i of the robot. Similarly, the last location in ξ_i must be the goal location l_g^i.

$$l_n^i = l_c^i \wedge l_{n+k}^i = l_g^i \tag{7.2}$$

(2) Trajectory continuity: A location in a trajectory is reachable from the previous location using the motion primitive applied at the previous location.

$$\forall q \in \{0, \ldots, k-1\} : \; l_{n+q+1}^i \in \texttt{post}(l_{n+q}^i, \gamma_{q+1}) \tag{7.3}$$

(3) Obstacle avoidance: No location on the trajectory should be covered with obstacles.

$$\forall q \in \{0, \ldots, k-1\} \, \forall l \in \texttt{intermediate}(l_{n+q}^i, \gamma_{q+1}) : l \notin \Omega \tag{7.4}$$

This constraint ensures the obstacle avoidance component ϕ_o of the *safe task-completion* property Φ_{st}.

(4) Collision avoidance: If the local clocks of all the robots are in perfect synchrony, ensuring collision avoidance would require that the robots do not occupy the same grid location in the workspace at the same time period according to their local clock. Motion plan synthesizer must ensure collision avoidance of robot r_i's trajectory represented as $\xi_i = (l^i_n l^i_{n+1} \dots l^i_{n'})$ with the trajectories of other robots captured in the set Ψ. The trajectory of any other robot r_j is denoted by $(l^j_m, \dots, l^j_n, \dots, l^j_{m'}) \in \Psi$, where $m \leqslant n$.

The following constraint guarantees collision avoidance property ϕ_c for a perfectly synchronous system:

$$
\begin{aligned}
&\forall r_j \in R \setminus \{r_i\}, (l^j_m, \dots, l^j_n, \dots, l^j_{m'}) \in \Psi : \\
&((\forall q \in \{n, \dots, \mathtt{min}(n', m')\} : \ l^i_q \neq l^j_q) \wedge \\
&\text{/* The robot } r_i \text{ reaches destination before robot } r_j \text{ */} \\
&(n' < m' \Rightarrow \forall q \in \{n'+1, \dots, m'\} : l^i_{n'} \neq l^j_q) \wedge \\
&\text{/* The robot } r_i \text{ reaches destination after robot } r_j \text{ */} \\
&(n' > m' \Rightarrow \forall q \in \{m'+1, \dots, n'\} : l^i_q \neq l^j_{m'}))
\end{aligned}
\tag{7.5}
$$

Once a robot reaches its destination, it stays there unless it computes a new trajectory using the motion planner. Equation 7.5 comprises a conjunction of three constraints (one per line). The first constraint enforces that two robots cannot occupy the same location at the same instant while moving. The second and third constraint specifies that a robot that is moving does not occupy the location of a stationary robot (that has stopped after reaching the destination).

When the clocks are not perfectly synchronous, then one must consider the synchronization precision β. We do so using the notion of approximate synchrony introduced in Chapter 5. Specifically, to ensure collision avoidance with another robot, the plan synthesizer of a robot should ensure that its location at time step τ^i_n does not overlap with the location of the other robot *at any step in the range of* $(\tau^i_n - \Delta, \tau^i_n + \Delta)$. Equation 7.6 extends Equation 7.5 to encode collision avoidance constraint with an approximate synchrony bound of Δ.

$$
\begin{aligned}
&\forall r_j \in R \setminus \{r_i\}, (l^j_m, l^j_{m+1}, \ldots, l^j_n, \ldots, l^j_{m'}) \in \Psi : \\
&\quad ((\forall q \in \{n, \ldots, \min(n', m')\}\, \forall p \in \{q-\Delta, \ldots, q+\Delta\} : \\
&\qquad (n \leqslant p \leqslant m' \Rightarrow l^i_q \neq l^j_p) \wedge \\
&\qquad (p < m \Rightarrow l^i_q \neq l^j_m) \ \wedge \ (p > m' \Rightarrow l^i_q \neq l^j_{m'})) \wedge \\
&/* \text{ The robot } r_i \text{ reaches destination before robot } r_j \ */ \\
&((n' < m') \Rightarrow \forall q \in \{n'+1, \ldots, m'\}\, \forall p \in \{q-\Delta, \ldots, q+\Delta\} : \\
&\qquad (p \leqslant n' \Rightarrow l^i_p \neq l^j_q) \ \wedge \ (p > n' \Rightarrow l^i_{n'} \neq l^j_q)) \wedge \\
&/* \text{ The robot } r_i \text{ reaches destination after robot } r_j \ */ \\
&((n' > m') \Rightarrow \forall q \in \{m'+1, \ldots, n'\}\, \forall p \in \{q-\Delta, \ldots, q+\Delta\} : \\
&\qquad (p \leqslant m' \Rightarrow l^i_q \neq l^j_p) \ \wedge \ (p > m' \Rightarrow l^i_q \neq l^j_{m'})))
\end{aligned}
\tag{7.6}
$$

SMT solver based safe plan-generator: To synthesize the motion plan using a satisfiability modulo theories (SMT) solver [22], we first start by initializing the length of the trajectory (k) to be the *manhattan distance* between the current location of the robot and its goal location. The constraints (Eq.(1)-Eq.(6)) are from the theory of linear integer arithmetic and the theory of equality with uninterpreted functions. We represent the obstacles using an uninterpreted function. If there exists a solution for the set of constraints, the solution provides us the desired motion plan. If no solution exists, we increase the value of k by 1 and attempt to solve the constraints again. We iterate that process until the value of k is less than or equal to L^i_{max} (a parameter that represents the maximal length to be considered for generating the trajectory for robot r_i). If no motion plan of length less than or equal to L^i_{max} is found, it is guaranteed that there does not exist a feasible motion plan of length less than equal to L^i_{max} for the given problem instance.

However, as our experimental results reveal (Section 7.4), an SMT based solution suffers from a lack of scalability for large grid sizes and multi-robot systems as constraints become hard to solve.

A* based safe plan-generator: To have a scalable implementation, we extend the well-known A* search algorithm [94] to generate *safe* motion plans. A* search algorithm can natively handle the objective function Equation 7.1 and the constraints Equation 7.2-(7.4) for static obstacles. We extended the function that computes adjacent nodes in A* to incorporate the constraints in Eq. (7.5) and Equation 7.6. We associate a time-stamp value to each node in the A* search graph. The time-stamp denotes the number of steps required to reach the current node from the start node. During adjacent node calculation, we use time-stamp at a node to encode the constraints in Equation 7.5 and Equation 7.6 to ensure that the trajectory through the potential adjacent node will not be in collision with the trajectory of any other robot.

7.2.4 *Plan Executor*

The plan-executor (PE) module plays an important role in the overall correctness of MRMP. It is the responsibility of the plan-executor module to ensure that the robot correctly follows its computed trajectory. The plan-generator (Section 7.2.3) generates a safe trajectory under the assumption that all robots in the system will follow their timed-trajectories that they communicated to other robots.

Recollect that the MRMP protocol (Algorithm 7.2.1, line 21) on computing a motion plan ρ_i sends it to the plan-executor module. The plan-executor executes the sequence of motion-primitives in ρ_i such that the robot r_i realizes its timed-trajectory ξ_i (Definition 7.1.1). It is implemented as a periodic state-machine with the duration of each period as τ, executing the next motion-primitive at each period.

For all the robots to follow their timed-trajectories correctly, the path-executor processes across robots must step periodically with a symmetric period τ, i.e, $\forall r_i \in R, \forall n, |\tau_n^i - \tau_{n+1}^i| = \tau$. Since path-executor at each robot r_i step using its local clock χ_i, the path-executors across the system do not step perfectly synchronously but almost-synchronously with a bound $\pm\Delta$ which the plan-generator has accounted for in Equation 7.6.

7.2.5 *Provably Correct Motion Planner*

Recollect that when computing a trajectory for a robot r_i, the execution of MRMP is decomposed into two phases: first, the coordination protocol computes the avoid trajectories set Ψ_i which is then used by the safe plan-generator for computing the collision-free trajectory ξ_i. We say that the avoid trajectories set Ψ_i is *consistent* if $\forall \zeta_j \in \Psi_i, \zeta_j = \xi_j$, where ζ_j is the trajectory sent by robot r_j to robot r_i and ξ_j is the actual trajectory being executed by robot r_j.

As described in Section 7.2.3.2, the A* based plan-generator always generates trajectories that satisfy the safe task-completion property Φ_{st} under the assumption that avoid trajectory set Ψ_i is *consistent*. In other words, given the set of trajectories Ψ_i, if the plan-generator computes trajectory ξ_i then $\texttt{consistent}(\Psi_i) \implies (\xi_i \models \Phi_{st})$.

In order to prove that the assumption $\texttt{consistent}(\Psi_i)$ holds, we verify (using model-checking) the following properties about the coordination protocol: (1) *Safety:* The avoid trajectory set Ψ_i computed by the coordination protocol is always *consistent*. (2) *Liveness:* If a dynamically generated task $(l, p) \in T$ is assigned to the robot r_i then it eventually computes *consistent* Ψ_i.

The multi-robot motion planner described in this section satisfies the following soundness theorem:

Theorem 7.2.2: Safe Task Completion

If a dynamically generated task (l_g^i, p) *is assigned to a robot* r_i *then the corresponding trajectory* ξ_i *computed by* MRMP *always satisfies the safe task-completion property* Φ_{st}.

Proof. As stated earlier, if ξ_i is the trajectory computed by the plan-generator using Ψ_i then it provides the guarantee that $\mathtt{consistent}(\Psi_i) \implies (\xi_i \models \Phi_{st})$ and we proved using model-checking that the coordination protocol always satisfies $\forall \Psi_i, \mathtt{consistent}(\Psi_i)$. ■

However, MRMP is not complete due to the following reason: for a given task, the corresponding robot may not be able to reach the destination because the other stationary robots may block its possible trajectories.

Ensuring properties (S1) to (S3) for a DMR system. The Theorem 7.2.2, implies the collision avoidance property **(S2)** when performing a task. Also, it can be used to prove that the satisfies the application specific task completion properties **(S1)**. However, note that the safe motion plan and safe task completion guarantees are satisfied under the assumption that the motion primitives satisfy the desired safety property that it always moves through the intermediate locations (see Section 7.1.1). This property may not hold when the motion primitives are implemented using third-party libraries or other untrusted techniques. In Chapter 8, we describe how this assumption about motion primitives is guaranteed using *runtime assurance* which in turn helps ensure the Theorem 7.2.2 for the robotics software stack.

7.3 VERIFICATION OF DMR SYSTEMS

In this section, we describe our approach for verifying that a DMR system ($\mathcal{M}$) satisfies specification Φ. As explained in Section 7.2.4, for the robots in the system to successfully follow their computed trajectories, the plan executor (PE) processes must step *almost-synchronously* with symmetric period τ. Hence, the PE processes across robots are implemented as periodic processes. All the other processes in the software stack, e.g., TP, MRMP, and SI are event-driven and are composed asynchronously. We call the DMR system as a *mixed synchronous* system as it is a composition of asynchronously composed processes and *almost-synchronously* composed processes.

7.3.1 *Formal Model of DMR system*

We model the DMR mixed synchronous system as a tuple $(k, \mathcal{S}, \mathcal{I}, \mathcal{P}_{sp}, \mathcal{P}_{as}, \vec{\chi}, \tau, \delta)$ where:

- k is the number of robots in the system.

- $\mathcal{S}$ is the set of discrete states of the system which is a product of the local states of all the processes.

- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states of the system.

- $\mathcal{P}_{sp} = \{\mathcal{P}^1_{sp}, \mathcal{P}^2_{sp}, \ldots, \mathcal{P}^k_{sp}\}$ is the set of process identifiers for the symmetric periodic (PE) processes. $\mathcal{P}^i_{sp}$ represents symmetric periodic process running on r_i.

- $\mathcal{P}_{as} = \{\mathcal{P}^1_{as}, \mathcal{P}^2_{as}, \ldots, \mathcal{P}^k_{as}\}$ is the set of process identifiers for the asynchronous processes. $\mathcal{P}^i_{as}$ represents composition of asynchronous process running on r_i. $\mathcal{P}^i_{as} = \text{TP}^i \parallel \text{MRMP}^i \parallel \text{SI}^i$.

- $\vec{\chi} = (\chi_1, \chi_2, \ldots, \chi_k)$ is a vector of real valued local clocks, each robot r_i has an associated local clock χ_i.

- $\vec{\tau}$ is the common global process timetable for the periodic $\mathcal{P}_{sp}$ processes. The timetable $\vec{\tau}$ is an infinite vector $(\tau^1, \tau^2, \tau^3, \ldots)$ specifying the time instants according to local clock χ_i when the process $\mathcal{P}^i_{sp}$ executes (steps). In other words, $\mathcal{P}^i_{sp}$ makes its jth step when $\chi_i(t) = \tau^j$ where $\chi_i(t)$ is the value of the local clock χ_i at global reference time t. Also, since the $\mathcal{P}_{sp}$ processes step with a period of τ, $|\tau^{j+1} - \tau^j| = \tau$.

- $\delta \subseteq \mathcal{S} \times \Sigma_{MS} \times \mathcal{S}$ is the labeled transition relation for the *mixed synchronous* system. Σ_{MS} denote $(2^{\mathcal{P}_{sp}} \setminus \{\}) \sqcup \mathcal{P}_{as}$, the transition labels of the system.

Note that the periodic $\mathcal{P}_{sp}$ processes have the same timetable, but that does *not* mean that the processes step perfectly synchronously, since their local clocks may report different values at the same global time t.

Timed traces: A timed trace σ of the mixed synchronous system $\mathcal{M}_{MS}$ is an infinite sequence of the timestamped record of the execution of the system according to the global (ideal) time reference t and is of the form $\sigma : (s_0, t_0), \ldots (s_n, t_n) \ldots$ with $\forall i.\ i \geqslant 0$, $s_i \in \mathcal{S}$, $t_i \in \mathbb{R}_{\geqslant 0}$ and $t_i \leqslant t_{i+1}$ satisfying requirements:
Initiation: $s_0 \in \mathcal{I}$, and $\forall i.\ \chi_i(t_0) = 0, t_0 = 0$.
Consecution: for all $i \geqslant 0$, there is a transition of the form (s_i, a_i, s_{i+1}) in δ such that the label a_i is either one of the following:

1. The label a_i is an asynchronous process, $a_i \in \mathcal{P}_{as}$ and the transition represents process a_i stepping at time t_i.

2. The label a_i is a subset of symmetric periodic processes, $a_i \subseteq \mathcal{P}_{sp}$ and $\forall j.\mathcal{P}^j_{sp} \in a_i$, $\chi_j(t_i) = \tau^m$ for some $m \in \{0, 1, 2, \ldots\}$. $\chi_j(t_i)$ is the value of the local clock χ_j at current global reference time t_i. This transition represents a subset of symmetric periodic processes making a step whose local clock value at time t_i is equal to some timetable value. Moreover, $\mathcal{P}_{sp}$ processes step according to their timetables;

thus, if any process $\mathcal{P}^i_{sp} \in \mathcal{P}_{sp}$ makes its mth and lth steps at times t_j and t_k respectively, for $m < l$, then $\chi_i(t_j) = \tau^{m_i} < \tau^{l_i} = \chi_i(t_k)$.

7.3.2 *Mixed Synchronous Abstraction*

$\mathcal{M}_{MS}$ system described above can be modeled as a hybrid or timed system (due to the continuous dynamics of physical clocks), but the associated methods [81, 120] for verification tend to be less efficient for systems with huge discrete state space. Instead, we construct the discrete abstraction $\widehat{\mathcal{M}}_{MS}$ of $\mathcal{M}_{MS}$ that preserves the relevant timing semantics of the *mixed synchronous* systems. We extend the *approximate synchrony abstraction* (see Section 5.2) to create an untimed *mixed synchronous* abstraction of $\mathcal{M}_{MS}$.

We define $\widehat{\mathcal{M}}_{MS}$ as a tuple $(k, \mathcal{S}, \mathcal{I}, \mathcal{P}_{sp}, \mathcal{P}_{as}, \rho_\Delta, \delta^a)$ where ρ_Δ is a scheduler process that performs an asynchronous composition of all the processes while enforcing approximate synchrony condition with parameter Δ (computed using Theorem 5.2.1) only for the $\mathcal{P}_{sp}$ processes. The scheduler ρ_Δ maintains counter N_i of the number of steps taken by each process $\mathcal{P}^i_{sp}$ from the initial state. A configuration of $\widehat{\mathcal{M}}_{MS}$ is a pair (s, N) where $s \in \mathcal{S}$ and $N \in \mathbb{N}^k$ is the vector of step counts for the $\mathcal{P}_{sp}$ processes. The transition function δ^a for the abstract model $\widehat{\mathcal{M}}_{MS}$ can be defined as $((s, N), a_i, (s', N')) \in \delta^a$ iff $\delta(s, a_i, s')$ and one of following holds: (1) $N'_j = N_j + 1$ and ρ_Δ permits all $\mathcal{P}^j_{sp} \in a_i$ to make a step, (2) $a_i \in \mathcal{P}_{as}$ and a_i makes a step.

ρ_Δ scheduler enforces the mixed synchrony condition during exploration by allowing $\mathcal{P}_{sp}$ processes to step iff their step does not violate the approximate synchrony condition, and the $\mathcal{P}_{as}$ are always allowed to step.

Untimed traces: Traces of $\widehat{\mathcal{M}}_{MS}$ are (*untimed*) sequences of discrete (global) states $s_0, s_1, s_2, \ldots$, where $s_j \in \mathcal{S}$, $s_0 \in \mathcal{I}$, and for all j, $(s_j, a_j, s_{j+1}) \in \delta^a$.

Theorem 7.3.1: Soundness of Mixed-Synchronous Abstraction

The abstract model $\widehat{\mathcal{M}}_{MS}$ is a sound abstraction of the concrete model $\mathcal{M}_{MS}$. Hence, $\widehat{\mathcal{M}}_{MS} \models \Phi$ implies $\mathcal{M}_{MS} \models \Phi$.

Proof. Let $\mathtt{traces}(\mathcal{M})$ represent the set of all *untimed* traces of the system $\mathcal{M}$. The untiming logic for timed traces is as defined by Alur in [8]. $\hat{\mathcal{M}}$ is a sound abstraction of $\mathcal{M}$ if $\mathtt{traces}(\mathcal{M}) \subseteq \mathtt{traces}(\hat{\mathcal{M}})$ We derive the proof-sketch from Theorem 5.2.1 which proves that for a time-synchronized system $\mathcal{M}_{ps}$ with synchronization β, the approximate synchrony based abstract model $\hat{\mathcal{M}}_{ps}$ is a sound abstraction with parameter $\Delta = \left\lceil \frac{\beta}{\tau} \right\rceil$. Since the $\mathcal{P}_{as}$ are interleaved asynchronously in both $\mathcal{M}_{MS}$ and $\widehat{\mathcal{M}}_{MS}$ we can further prove that $\mathtt{traces}(\mathcal{M}_{MS}) \subseteq \mathtt{traces}(\widehat{\mathcal{M}}_{MS})$. ■

Note that mixed-synchronous abstraction is critical for the verification of DMR systems. Performing synchronous composition of all processes in the system is unsound and performing asynchronous composition can lead to false-positives due to over-approximation.

Implementation of the verification approach: The P explorer (model checker) supports directed search based on an external scheduler (as described in Chapter 4). We implemented the mixed synchrony scheduler (ρ_Δ) as an external scheduler that constraints the interleaving explored during verification. The model-checking algorithm that uses approximate synchrony scheduler is described in Chapter 5. Note that the key feature that comes to rescue is the ability of the P explorer to enable analysis of event-driven systems using external schedulers.

7.4 EVALUATION

In this chapter, we empirically evaluate the DRONA framework with the following goals:
(Goal 1) Show that safe plan-generator can be used for on-the-fly motion planning with a large number of robots and large workspace size.
(Goal 2) Show how time-synchronization error (Δ) effects optimal path computation.
(Goal 3) Demonstrate the advantages of using DRONA for building reliable DMR system by implementing and verifying the priority mail delivery system as a case study.
(Goal 4) Deploy the generated code from DRONA on the ROS [163] simulator (and real drone platforms) for various configurations to validate the reliability.
All the experiments were performed on a laptop with 2.5 GHz Intel i7 core processor with 16GB RAM.

Evaluation of safe plan generator: Recently, there is an increased interest towards using SMT solvers for motion plan synthesis [148, 169, 170]. The performance of the plan generator depends on the complexity of constraints generated, which varies based on the size of the workspace, the number of robots, their current trajectories, and the density of static obstacles. From our experiments, we found that the state-of-the-art solver Z3 [45] does not scale for plan generation in the context of multi-robot systems. Generating a motion plan with a workspace of size 64x64 and 16 robots takes 2 min 18 secs (see Table 7.1).

We implemented the plan-generator using a publicly available A* implementation [15] and encoded the path constraints into A* search. In our evaluation of A* based plan generator, we increase the number of robots from 4 to 128 and consider 2-D grids of sizes 16x16 to 256x256 (our motion planner supports 3-D workspaces, simulation video at [64]). We generated random workspaces of varying size such that obstacles occupy 20% of the grid locations. We simulated a system with n robots and created an environment that pumps in a sequence of task requests with random goal location. We measured the amount of time it takes for each robot to compute

|R|	Time in seconds		
	Grid Size		
	16x16	32x32	64x64
4	0.66	3.5	15.4
8	0.9	8.5	33.55
16	-	44.6	138

Table 7.1: Performance of SMT-based plan-generator

its trajectory. Table 7.2 reports the average computation time over 300 invocations of plan-generator for different configurations.

|R|	Computation time in seconds				
	Grid Size				
	16x16	32x32	64x64	128x128	256x256
4	0.0174	0.0179	0.0215	0.0518	0.1485
8	0.0179	0.0184	0.0249	0.0837	0.2651
16	0.0187	0.0206	0.0318	0.0884	0.3038
32	-	0.0247	0.0435	0.1007	0.3186
64	-	-	0.0666	0.1538	0.3882
128	-	-	-	0.2293	0.5159

Table 7.2: Performance of A* based plan-generator

The results show that our plan generator that takes into account time-synchronized clocks is scalable for large grid sizes and robots. Hence, it can be used for generating plans on-the-fly in a decentralized fashion with formal guarantees.

Effect of Δ on planning: The approximate-synchrony parameter Δ represents the clock skew (and thus, step skew) in the system and effects the window of locations avoided by robots when computing trajectory. In other words, it affects how conservative a robot is when computing the trajectory. Hence, the optimal path for a robot may change based on the value of Δ. A simulation video to demonstrate this scenario is available at [64].

Building multi-robot surveillance system: We implemented the multi-robot surveillance system software stack (Figure 6.2) in P. We used the mixed synchronous discrete abstraction (Section 7.3.2) for verifying that the implementation always satisfies the properties **(S1)** to **(S3)**(Section 6.1). These specifications were implemented as P monitors. During the process of implementing the software stack, we found many critical bugs that would have been hard to find otherwise using traditional simulation-based

approach. For example, there was a bug (race condition) in the coordination protocol, which led to the case where a robot computes its trajectory using an older trajectory of other robots, causing a collision. This race condition could not be reproduced with 2 hours of random simulations but was caught in a few seconds using the model-checker. We also deployed the generated code on to the drone platform for conducting simple drone missions, and the videos are available on the DRONA website [64].

Evaluation of the Mixed-Synchronous abstraction-based Verification: We performed an analysis of the application in two phases:
(1) Stratified random sampling: To catch shallow bugs in our implementation, we first performed stratified sampling of executions (Section 4.3). We were able to find most of the bugs in our implementation during this mode of testing. Note that this is similar to performing random simulations but much more scalable as we use a parallel model checker for exploration.

|R|	Max depth explored in 10 hours		
	Grid Size		
	8x8	16x16	32x32
2	✓	✓	✓
4	✓	✓	✓
8	✓	✓	(78)

Table 7.3: Scalability of verification approach

(2) Deterministic exploration: Sampling-based approaches fail to provide coverage guarantees, for that, we performed deterministic enumeration (with state caching) of all possible executions in the system with max depth 100 and time budget of 10 hours. Table 7.3 shows the coverage results for various grid sizes and the number of robots. ✓represents that P explored all possible executions till depth 100 and (n) represents that P explored all possible executions till the depth n in the given time budget.

Rigorous Simulations: We also implemented a ROS simulator that supports 3-D simulation of the code generated from the DRONA framework. Figure 7.2 presents a snapshot of our multi-robot simulator. Simulation videos for various configurations are available at [64]. To validate the reliability of code generated by DRONA, we added runtime assertions into the generated C code and ran the simulations for 128 robots with random task generator for 12 hours. We did not find any bug during this stress testing, confirming that the verified code generated from the DRONA framework is reliable.

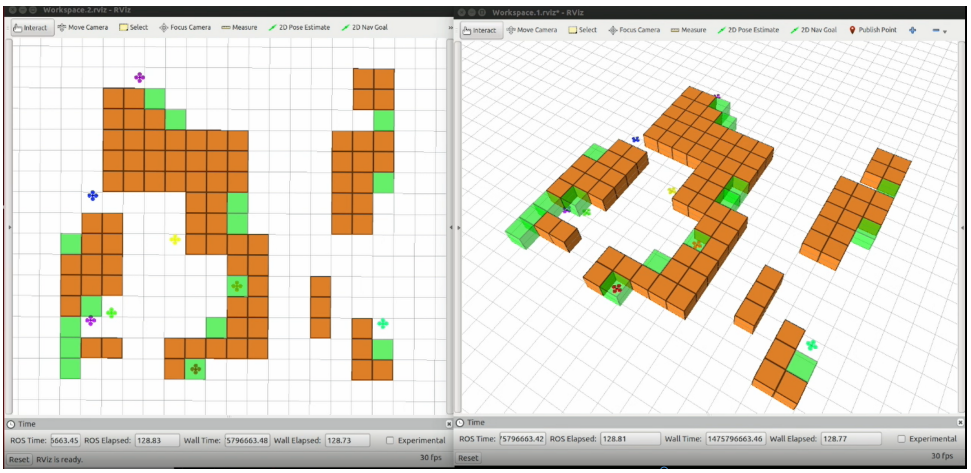


Figure 7.2: Multi-Robot ROS Simulator

7.5 RELATED WORK

We have discussed the related work in the area of building safe robotics systems, and have situated it with regards to the DRONA framework (see Section 6.4). In this section, we consider our other contribution of a novel decentralized reactive multi-robot motion planner and present the corresponding related work.

The problem of synthesizing collision-free trajectories for multi-robot systems in a scenario where the robots are preassigned a set of tasks has been addressed in several prior works. It can be categorized as follows: **(1)** ***Centralized motion planning*** (e.g. [72, 169, 170, 192]) where a central server, given a set of tasks and robots in the system, computes the collision-free trajectory for each robot offline, **(2)** ***Decentralized prioritized planning*** (e.g. [33, 91, 194]) where given a fixed set of tasks, the robots in the system coordinate with each other asynchronously for computing the trajectories. These papers empirically show that decentralized approaches can converge faster than the centralized approach. In this chapter, we presented a decentralized motion planning that can handle dynamically generated tasks and are robust against "almost synchrony". Recently, there is increased interest in using temporal logic formalism for synthesizing reactive motion plans [46, 116, 201]. This approach, in principle, can be extended and applied to solve a DMR problem. However, the problem with automated synthesis is that the algorithms scale poorly both with the number of robots and the size of the workspace. Also, they resolve collisions only locally and therefore cannot always guarantee that the resulting motion plan will be deadlock-free and that the robot will eventually reach its destination.

7.6 SUMMARY

In this chapter, we presented the DRONA framework for building reliable robotics systems. We implemented the reactive DMR software stack in P and used the abstraction based model-checking approach to find bugs in our implementation which rigorous random simulations failed to find. The multi-robot motion planner (MRMP) implemented as part of the DMR stack is provably correct and scales efficiently for large number of robots and large workspaces. MRMP is the first to take into account the time-synchronization error in a distributed multi-robot system when generating safe motion plans. We deployed the reliable DMR software stack on actual drone platform to perform several experiments and demos, the videos are available on the DRONA webpage.

8

GUARANTEEING SAFETY USING RUNTIME ASSURANCE

Exploring the unknown requires tolerating uncertainty

— Brian Greene

In Chapter 7, we described how the DRONA framework can be used for building safe robotics systems, in particular, we address the first challenge (Section 6.2) of programming safe reactive event-driven robotics software stack and verifying that the implementation satisfies the desired correctness specifications. However, these guarantees are provided by the DRONA framework under the assumption that the *untrusted* components (colored blocks in Figure 6.2) in the software stack satisfy the desired specification. This leads to a gap between the guarantees provided by the design-time verification and the actual behavior of the robot at runtime. One approach to bridging this gap is to leverage techniques for *run-time assurance*, where the results of design-time verification are used to build a system that monitors itself and its environment at run time; and, when needed, switches to a provably-safe operating mode, potentially at lower performance and sacrificing certain non-critical objectives. In this chapter, we seek to address the second challenge (Section 6.2) of building safe robotics systems in the presence of untrusted components by extending DRONA with runtime assurance capabilities.

We refer to the runtime assurance component of the DRONA tool chain as SOTER [58].

Runtime Assurance Architecture: A prominent example of a Run-Time Assurance (RTA) framework is the *Simplex Architecture* [181], which has been used for building provably-correct safety-critical avionics [171, 180], robotics [157] and cyber-physical systems [19, 20, 40]. The typical RTA architecture based on Simplex [181] (see Figure 8.1) comprises three sub-components: **(1)** The *advanced controller* (AC) that controls the robot under nominal operating conditions, and is designed to achieve *high-performance* with respect to specialized metrics (e.g., fuel economy, time), but it is not provably safe, **(2)** The *safe controller* (SC) that can be pre-certified to keep the robot within a region of safe operation for the plant/robot, usually at the cost of lower performance, and **(3)** The *decision module* (DM) which is pre-certified (or automatically synthesized

to be correct) to periodically monitor the state of the plant and the environment to determine when to switch from AC to SC so that the system is guaranteed to stay within the safe region. When AC is in control of the system, DM monitors (samples) the system state every Δ period to check whether the system can violate the desired safety specification (ϕ) in time Δ. If so, then DM switches control to SC.

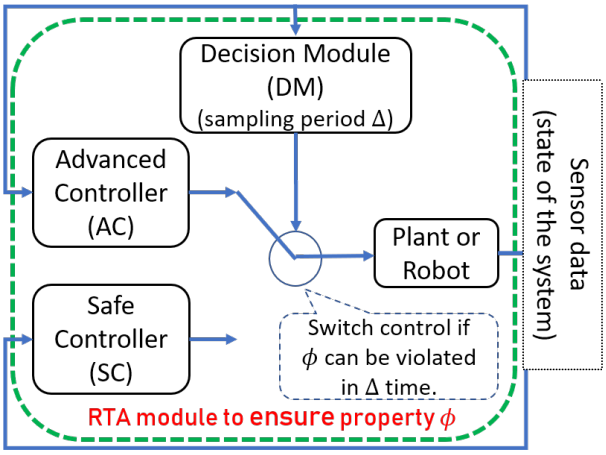


Figure 8.1: RTA Architecture

This Simplex-based RTA architecture is a very useful high-level framework, but there are several limitations of its existing instantiations. First, existing techniques either apply RTA [27, 156, 171] to a single untrusted component in the system or wrap the large monolithic system into a single instance of Simplex which makes the design and verification of the corresponding SC and DM difficult or infeasible. Second, most prior applications of RTA do not provide high-level *programming language support* for constructing provably-safe RTA systems in a *modular* fashion while designing for *timing and communication behavior* of such systems. In order to ease the construction of RTA systems, there is a need for a general programming framework for building provably-safe robotic software systems with run-time assurance that also considers implementation aspects such as timing and communication. Finally, existing techniques do not provide a principled and safe way for DM to switch back from SC to AC to keep performance penalties to a minimum while retaining strong safety guarantees.

In this chapter, we address these limitations with SOTER, an extension of DRONA with runtime assurance capabilities. We extended the P language (the programming language used in DRONA) with primitives to implement periodic processes, termed *nodes*, that interact with each other using a publish-subscribe model of communication (which is popular in robotics, e.g., in Robot Operating System, ROS [163]). An RTA module in SOTER consists of an advanced controller node, a safe controller node, and a safety specification; if the module is well-formed, then the framework provides a guarantee that the system satisfies the safety specification. SOTER allows programmers to declaratively construct an RTA module with specified timing behavior, combining

provably-safe operation with the feature of using AC whenever safe to achieve good performance. SOTER provides a provably-safe way for DM to switch back from SC to AC, thus extending the traditional RTA framework and providing higher performance. Our evaluation demonstrates that DRONA is effective at achieving this blend of safety and performance.

Crucially, SOTER supports compositional construction of the overall RTA system. The extended SOTER language framework includes constructs for decomposing the design and verification of the overall RTA system into that for individual RTA modules while retaining guarantees of safety for the overall composite system. The compiler generates the DM node that implements the switching logic, and which also generates C code to be executed on common robotics software platforms such as ROS [163] and MavLink [151].

We show that SOTER can be used to build a complex robotics software stack consisting of both third-party untrusted components and complex machine learning modules, and still provide system-wide correctness guarantees. The generated code for the robotics software has been tested both on an actual drone platform (the 3DR [2] drone) and in simulation (using the ROS/Gazebo [114] and OpenAI Gym [29]). Our results demonstrate that the RTA-protected software stack built using SOTER can ensure the safety of the drone both when using unsafe third-party controllers and in the presence of bugs introduced using fault injection in the advanced controller.
In summary, we make the following novel contributions in this chapter:

1. A programming framework for a Simplex-based run-time assurance system that provides language primitives for the modular design of safe robotics systems (Section 8.2);

2. A theoretical formalism based on computing reachable sets that keep the system provably safe while maintaining smooth switching behavior from advanced to a safe controller *and vice-versa* (Section 8.3);

3. A framework for the modular design of run-time assurance (Section 8.4), and

4. Experimental results in simulation and on real drone platforms demonstrating how SOTER can be used for guaranteeing the correctness of a system even in the presence of untrusted or unverified components (Section 8.5).

8.1 OVERVIEW

We illustrate the runtime assurance extensions to DRONA framework by using our case study of an autonomous drone surveillance system.

8.1.1 *Case Study: Drone Surveillance System*

We revisit the drone surveillance case study from Section 6.1, we would like the system to satisfy two safety invariants: **(1)** *Obstacle Avoidance* ($\phi_{\texttt{obs}}$): The drone must never collide with any obstacle. **(2)** *Battery Safety* ($\phi_{\texttt{bat}}$): The drone must never crash because of low battery. Instead, when the battery is low it must prioritize landing safely. $\phi_{\texttt{obs}}$ can be further decomposed into two parts $\phi_{\texttt{obs}} := \phi_{\texttt{plan}} \wedge \phi_{\texttt{mpr}}$; **(a)** *Safe Motion Planner* ($\phi_{\texttt{plan}}$): The motion planner must always generate a motion-plan such that the reference trajectory does not collide with any obstacle, **(b)** *Safe Motion Primitives* ($\phi_{\texttt{mpr}}$): When tracking the reference trajectory between any two waypoints generated by the motion planner, the controls generated by the motion primitives must ensure that the drone closely follows the trajectory and avoids collisions.

Challenges and Motivation. As described in Section 6.1, when implementing the software stack, the programmer may use several uncertified components. For example, implementing an on-the-fly motion planner may involve solving an optimization problem or using an efficient search technique that relies on a solver or a third-party library (e.g., OMPL [184]). Similarly, motion primitives are either designed using machine-learning techniques like Reinforcement Learning [109], or optimized for specific tasks without considering safety, or are off-the-shelf controllers provided by third parties [151]. Ultimately, in the presence of such uncertified or hard to verify components, it is challenging to provide formal guarantees of safety at design time.

In practice, for complex systems, it can be extremely difficult to design a component that is both safe and high-performance. The AC, in general, is any program or component designed for high-performance under nominal conditions using either third-party libraries or machine-learning techniques. We treat them as unsafe since they often exhibit unsafe behavior in off-nominal conditions and uncertain environments, and even when they do not, it is hard to be sure since their complexity makes verification or exhaustive testing prohibitively expensive. Furthermore, the trend in robotics is towards *advanced*, data-driven controllers, such as those based on neural networks (NN), that usually do not come with safety guarantees. Our approach of integrating RTA into a programming framework is motivated by the need to enable the use of such advanced controllers (e.g., designed using NN or optimized for performance) while retaining strong guarantees of safety.

8.1.2 *Extending the* P *Language*

The Robot Operating System (ROS [163]) is an open-source meta-operating system considered as the de facto standard for robot software development. In most cases, a ROS programmer implements the system as a collection of periodic processes that communicate using the publish-subscribe model of communication. We extended the P language based on a similar publish-subscribe model of communication. We introduce

periodic *nodes* (processes) in P that communicate with each other by publishing on and subscribing to message topics. A node periodically listens to data published on specific topics, performs computation, and publishes computed results on certain other topics. A topic is an abstraction of a communication channel.

Topics: Listing 8.1 declares the topic `targetWaypoint` that can be used to communicate messages of type `coord` (coordinates in 3D space). In SOTER, a node communicates with other nodes in the system by publishing messages on a topic (e.g., `targetWaypoint`) and the target nodes can consume these messages by subscribing to it.

```
1 type coord = (x: float, y: float, z: float);
2 topic NextWaypoint : coord;
3 ...
4 node MotionPrimitive
5 period 10;
6 subscribes LocalPosition, NextWaypoint;
7 publishes  Control;
8 { /* body */ }
```

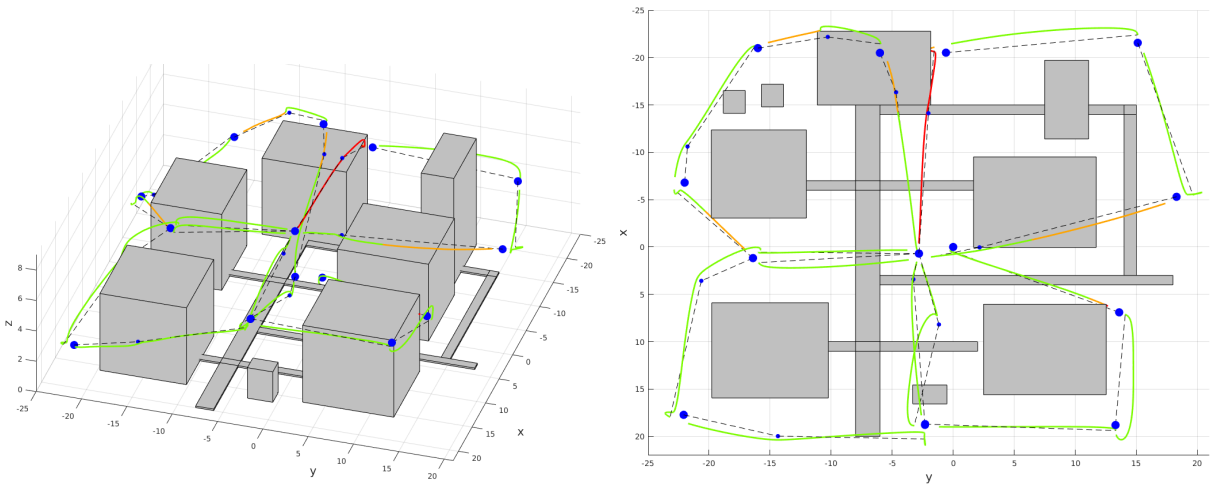
Listing 8.1: Declaration of topics and nodes in SOTER

Nodes: Listing 8.1 also declares a node `MotionPrimitive` that subscribes to topics `localPosition` and `targetWaypoint`. Each node has a separate local buffer associated with each subscribed topic. The publish operation on a topic adds the message into the corresponding local buffer of all the nodes that have subscribed to that topic. The `MotionPrimitive` node runs periodically every 10 ms. It reads messages from the subscribed topics, performs local computations, and then publishes the control action on the output topic. For the exposition, we ignore the syntactic details of the node body; it can be any sequential function written in P that performs the required read $\rightarrow$ compute $\rightarrow$ publish step.

8.1.3 *Guaranteeing Safety using Runtime Assurance*

In practice, the motion primitives (e.g., `MotionPrimitive` node in Listing 8.1) might generate control actions to traverse the reference trajectory from current position to the target waypoint using a low-level controller provided by the third-party robot manufacturer (e.g., [151]). These low-level controllers generally use approximate models of the dynamics of the robot and are optimized for performance rather than safety, making them unsafe.

In Section 6.2.2, we presented experiments that demonstrate unsafe behavior of a drone under the influence of untrusted motion primitives provide by third-party or are built using machine-learning techniques. To further emphasize the uncertainties involved when using untrusted component, we online-monitored trajectories taken by the drone during a surveillance mission in the city workspace (see Figure 6.1).



(a) 3-D View (b) Top View

Figure 8.2: Online monitoring of obstacle avoidance property during surveillance mission. green: property satisfied, orange : system very close to violating the property, red: property violated.

We consider an obstacle avoidance scenario, where the drone must never get closer than 0.5m to any obstacle in the workspace during its flight. We online monitored this requirement on all the trajectories generated by the drone during the surveillance task. Figure 8.2 shows two views of a faulty trajectory of the drone. Note how online monitoring detects a specification violation (red trace), meaning that the drone gets too close ($<$ 0.5m) to an obstacle. Also, observe that the robot robustly satisfy the specification in most of the trajectory (orange and green).

This motivates the need for a RTA system that guarantees safety by switching to a safe controller in case of danger but also maximizes the use of the untrusted but performant controller under nominal conditions.

Runtime Assurance module: Figure 8.3 illustrates the behavior of a SOTER based RTA-protected motion primitive module. We want the drone to move from its current location w_i to the target location w_f, and the desired safety property is that the drone must always remain inside the region ϕ_{safe} (outermost tube). Initially, the untrusted AC node (e.g., `MotionPrimitive`) is in control of the drone (red trajectory), and since it is not certified for correctness, it may generate controls action that tries to push the drone outside the ϕ_{safe} region.

If AC is wrapped inside an RTA module (see Figure 8.1) then DM must detect this imminent danger and switch to SC (blue trajectory) with enough time for SC to gain control over the drone. SC must be certified to keep the drone inside ϕ_{safe} and also move it to a state in ϕ_{safer} where DM evaluates that it is safe enough to return control

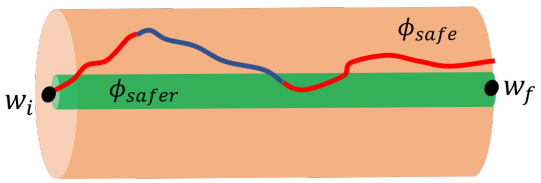


Figure 8.3: An RTA-protected Motion Primitive

to AC. The novel aspect of an RTA module formalized is that *it also allows control to return to* AC *to maximize performance*.

```
1  type State = ..;
2  ...
3  fun PhiSafer_MPr (s : State) : bool { ... }
4  fun TTF2D_MPr (s : State) : bool { ... }
5  ...
6  node MotionPrimitiveSC period 60;
7  subscribes LocalPosition, LocalVelocity, NextWaypoint;
8  publishes  Control;
9  { /* body */ }
10
11 rta SafeMotionPrimitive = { MotionPrimitive,
       MotionPrimitiveSC, 150, PhiSafer_MPr, TTF2D_MPr};
```

Listing 8.2: Declaration of an RTA module

Listing 8.2 presents the declaration of an RTA module consisting of `MotionPrimitive` (from Listing 8.1) and `MotionPrimitiveSC` as AC and SC nodes. The compiler checks that the declared RTA module `SafeMotionPrimitive` is well-formed (Section 8.3) and then generates the DM and the other glue code that together guarantees the ϕ_{safe} property. Details about other components of the module declaration are provided in Section 8.3.

Compositional RTA System. A large system is generally built by composing multiple components together. When the system-level specification is decomposed into a collection of simpler component-level specifications, one can scale provable guarantees to large, real-world systems.

SOTER enables building a *reliable* version (Figure 8.4) of the software stack with *runtime assurance* of the safety invariant: $\phi_{plan} \wedge \phi_{mpr} \wedge \phi_{bat}$. We decompose the stack into three components: **(1)** An RTA-protected motion planner that guarantees ϕ_{plan}, **(2)** A battery-safety RTA module that guarantees ϕ_{bat}, and **(3)** An RTA-protected motion primitive module that guarantees ϕ_{mpr}. Our theory of well-formed RTA modules (Theorem 8.3.1) ensures that if the constructed modules are well-formed, then they satisfy the desired safety invariant and their composition (Theorem 8.4.1) helps prove that the system-level specification is satisfied.

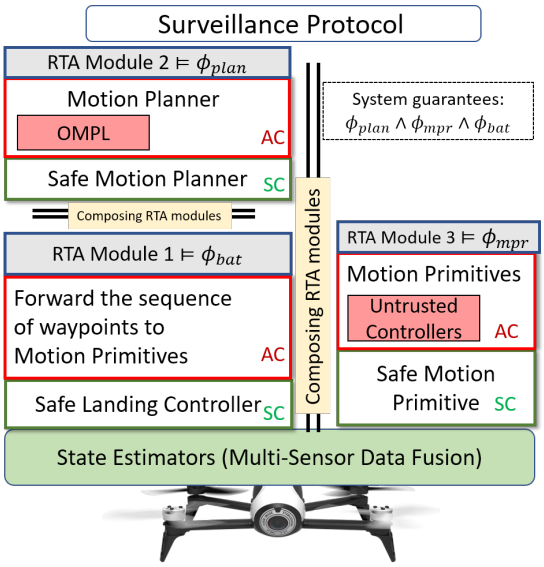


Figure 8.4: An RTA Protected Software Stack for Drone Surveillance

8.2 RUNTIME ASSURANCE (RTA) MODULE

In this section, we formalize the SOTER runtime assurance module and present the well-formedness conditions required for its correctness. We conclude by informally describing the behavior of a system protected by an RTA module.

8.2.1 *Programming Model*

Recollect that a program in SOTER is a collection of periodic nodes communicating with each other by publishing on and subscribing to message topics.

Topic. Formally, a topic is a tuple (e, v) consisting of a unique name $e \in \mathcal{T}$, where $\mathcal{T}$ is the universe of all topic names, and a value $v \in \mathcal{V}$, where $\mathcal{V}$ is the universe of all possible values that can be communicated using topic e. For simplicity of presentation: (1) we assume that all topics share the same set $\mathcal{V}$ of possible values and (2) instead of modeling the local buffers associated with each subscribed topic of a node; we model the communication between nodes using the global value associated with each topic.

Let $\mathcal{N}$ represent the set of names of all the nodes. We sometimes refer to a node by its unique name, for example, when $N_{ac} \in \mathcal{N}$ and we say "node N_{ac}", we are referring to a node with name N_{ac}. Let $\mathcal{L}$ represent the set of all possible values the local state of any node could have during its execution. A *valuation* of a set $X \subseteq \mathcal{T}$ of topic names is a map from each topic name $x \in X$ to the value v stored at topic (x, v). Let $\texttt{Vals}(X)$ represent the valuations of set X.

Node. A node in SOTER is a tuple (N, I, O, T, C) where:

1. $N \in \mathcal{N}$ is the unique name of the node.

2. $I \subseteq \mathcal{T}$ is the set of names of all topics subscribed to by the node (inputs).
3. $O \subseteq \mathcal{T}$ is the set of names of all topics on which the node publishes (output). The output topics are disjoint from the set of input topics ($I \cap O = \emptyset$).
4. $T \subseteq \mathcal{L} \times (I \to \mathcal{V}) \times \mathcal{L} \times (O \to \mathcal{V})$ is the transition relation of the node. If $(l, \texttt{Vals}(I), l', \texttt{Vals}(O)) \in T$, then on the *input* (subscribed) topics valuation of $\texttt{Vals}(I)$, the local state of the node moves from l to l' and publishes on the *output* topics to update its valuation to $\texttt{Vals}(O)$.
5. $C = \{(N, t_0), (N, t_1), \dots\}$ is the time-table representing the times $t_0, t_1, \dots$ at which the node N takes a step.

Intuitively, a node is a periodic input-output state-transition system: at every time instant in its calendar, the node reads the values in its input topics, updates its local state, and publishes values on its output topics. Note that we are using the timeout-based discrete event simulation [66] to model the periodic real-time process as a standard transition system (more details in Section 8.4). Each node specifies, using a time-table, the fixed times at which it should be scheduled. For a periodic node with period δ, the calendar will have entries $(N, t_0), (N, t_1), \dots$ such that $t_{i+1} - t_i = \delta$ for all i. We refer to the components of a node with name $N \in \mathcal{N}$ as $I(N), O(N), T(N)$ and $C(N)$ respectively. We use $\delta(N)$ to refer to the period δ of node N.

8.2.2 *Runtime Assurance Module*

Let $\mathcal{S}$ represent the state space of the system, i.e., the set of all possible configurations of the system (formally defined in Section 8.4). We assume that the desired safety property is given in the form of a subset $\phi_{safe} \subseteq \mathcal{S}$ (*safe states*). The goal is to ensure using an RTA module that the system always stays inside the safe set ϕ_{safe}.

RTA Module. An RTA module is represented as a tuple $(N_{ac}, N_{sc}, N_{dm}, \Delta, \phi_{safe}, \phi_{safer})$ where:

1. $N_{ac} \in \mathcal{N}$ is the advanced controller (AC) node,
2. $N_{sc} \in \mathcal{N}$ is the safe controller (SC) node,
3. $N_{dm} \in \mathcal{N}$ is the decision module (DM) node,
4. $\Delta \in \mathbb{R}^+$ represents the period of DM ($\delta(N_{sc}) = \Delta$),
5. $\phi_{safe} \subseteq \mathcal{S}$ is the desired safety property.
6. $\phi_{safer} \subseteq \phi_{safe}$ is a stronger safety property.

```
1   if (mode=SC ∧ s_t ∈ φ_safer)
2     mode = AC /*switch to AC*/
3   else if (mode=AC ∧ Reach_M(s_t, *, 2Δ) ⊈ φ_safe)
4     mode = SC /*switch to SC*/
5   else
6     mode = mode /* no mode switch */
```

Listing 8.3: Decision Module Switching Logic for Module M

Given an RTA module M, Listing 8.3 presents the switching logic that sets the *mode* of the RTA module given the current state s_t of the system. The DM node evaluates this switching logic once every Δ time unit. When it runs, it first reads the current state s_t and sets mode based on it. Note that the set ϕ_{safer} determines when it is safe to switch from N_{sc} to N_{ac}. $Reach_M(s, *, t) \subseteq \mathcal{S}$ represents the set of all states reachable in time $[0, t]$ starting from the state s, using *any non-deterministic controller*. We formally define Reach in Section 8.4, informally, $Reach_M(s_t, *, 2\Delta) \not\subseteq \phi_{safe}$ checks that the system will remain inside ϕ_{safe} in the next 2Δ time. This 2Δ look ahead is used to determine when it is necessary to switch to using N_{sc}, in order to ensure that the N_{sc} ($\delta(N_{sc}) \leqslant \Delta$) will be executed at least once before the system leaves ϕ_{safe}. The SOTER compiler automatically generates a unique DM node (N_{dm}) for each primitive RTA module declaration.

For an RTA module $(N_{ac}, N_{sc}, N_{dm}, \Delta, \phi_{safe}, \phi_{safer})$, DM is the node $(N_{dm}, I_{dm}, \emptyset, T_{dm}, C_{dm})$ where:

1. The local state is a binary variable *mode* : $\{AC, SC\}$.
2. Topics subscribed by DM include the topics subscribed by either of the nodes; i.e., $I(N_{ac}) \subseteq I_{dm}$ and $I(N_{sc}) \subseteq I_{dm}$.
3. DM does not publish on any topic. But it updates a global data structure that controls the outputs of AC and SC nodes (more details in Section 8.4).
4. If $(mode, \texttt{Vals}(I_{dm}), mode', \emptyset) \in T_{dm}$, then the local state moves from mode to mode′ based on the logic in Listing 8.3.
5. $C_{dm} = \{(N_{dm}, t_0), (N_{dm}, t_1), \dots\}$ where $\forall_i |t_i - t_{i+1}| = \Delta$ represents the time-table of the node.

We are implicitly assuming that the topics I_{dm} read by the DM contain enough information to evaluate ϕ_{safe}, ϕ_{safer}, and perform the reachability computation described in Section 8.4. Given a declaration of the RTA module (Listing 8.2), the SOTER compiler can automatically generate its DM.

8.3 CORRECTNESS OF AN RTA MODULE

The goal of an RTA module is to ensure that the system always stays inside the safe set ϕ_{safe}. We need an RTA module to satisfy some additional conditions to prove its safety.

An RTA module $M = (N_{ac}, N_{sc}, N_{dm}, \Delta, \phi_{safe}, \phi_{safer})$ is said to be *well-formed* if its components satisfy the following properties:

(P1a) The maximum period of N_{ac} and N_{sc} is Δ, i.e., $\delta(N_{dm}) = \Delta$, $\delta(N_{ac}) \leqslant \Delta$, and $\delta(N_{sc}) \leqslant \Delta$.

(P1b) The output topics of the N_{ac} and N_{sc} nodes must be same, i.e., $O(N_{ac}) = O(N_{sc})$.

The safe controller, N_{sc}, must satisfy the following properties:

(P2a) (*Safety*) $\mathtt{Reach}_M(\phi_{safe}, N_{sc}, \infty) \subseteq \phi_{safe}$. This property ensures that if the system is in ϕ_{safe}, then it will remain in that region as long as we use N_{sc}.

(P2b) (*Liveness*) For every state $s \in \phi_{safe}$, there exists a time T such that for all $s' \in \mathtt{Reach}_M(s, N_{sc}, T)$, we have $\mathtt{Reach}_M(s', N_{sc}, \Delta) \subseteq \phi_{safer}$. In words, from every state in ϕ_{safe}, after some finite time, the system is guaranteed to stay in ϕ_{safer} for at least Δ time.

(P3) $\mathtt{Reach}_M(\phi_{safer}, *, 2\Delta) \subseteq \phi_{safe}$. This condition says that irrespective of the controller if the system starts from a state in ϕ_{safer}, it remains in ϕ_{safe} for 2Δ time units. Note that this condition is stronger than the condition $\phi_{safer} \subseteq \phi_{safe}$.

Theorem 8.3.1: Runtime Assurance

For a well-formed RTA *module* M, *let* $\phi_{Inv}(\mathtt{mode}, s)$ *denote the predicate* $(\mathtt{mode{=}SC} \wedge s \in \phi_{safe}) \vee (\mathtt{mode{=}AC} \wedge \mathtt{Reach}_M(s, *, \Delta) \subseteq \phi_{safe})$.
If the initial state satisfies the invariant ϕ_{Inv}, *then every state* s_t *reachable from* s *will also satisfy the invariant* ϕ_{Inv}.

Proof. Let $(\mathtt{mode}, s)$ be the initial mode and initial state of the system. We know that the invariant holds at this state. Since the initial mode is SC, then, by assumption, $s \in \phi_{safe}$. We need to prove that all states s_t reachable from s also satisfy the invariant. If there is no mode change, then invariant is satisfied by Property **(P2a)**. Hence, assume there are mode switches. We prove that in every time interval between two consecutive executions of the DM, the invariant holds. So, consider time T when the DM executes. (Case1) We first prove that as long as there is no mode switch, this claim is valid. The mode at time T is SC, and there is no mode switch at this time. Property **(P2a)** implies that all future states satisfy the invariant.

(Case2) The mode at time T is SC, and there is a mode switch to the AC at this time. Then, the current state s_T at time T satisfies the condition $s_T \in \phi_{safer}$. By Property **(P3)**, we know that $Reach_M(s_T, *, 2\Delta) \subseteq \phi_{safe}$, and hence, it follows that $Reach_M(s_T, *, \Delta) \subseteq \phi_{safe}$, and hence the invariant ϕ_{Inv} holds at time T. In fact, irrespective of what actions AC applies to the plant, Property **(P3)** guarantees that the invariant holds for the interval $[T, T + \Delta]$. Now, it follows from Property **(P1)** that the DM executes again at or before the time instant $T + \Delta$, and hence the invariant holds until the next execution of DM.
(Case3) The current mode at time T is AC, and there is a mode switch to SC at this time. Then, the current state s_T at time T satisfies the condition $Reach_M(s_T, *, 2\Delta) \not\subseteq \phi_{safe}$. Since the mode at time $T - \epsilon$ was still AC, and by the inductive hypothesis, we know that the invariant held at that time; therefore, we know that $Reach_M(s_{T-\epsilon}, *, \Delta) \subseteq \phi_{safe}$. Therefore, for the period $[T - \epsilon, T - \epsilon + \Delta]$, we know that the reached state is in ϕ_{safe} and the invariant holds. Moreover, SC gets a chance to execute in this interval at least once, and hence, from that time point onwards, Property **(P2a)** guarantees that the invariant holds.
(Case4) The current mode at time T is AC, and there is a no mode switch. Since there is no mode switch at T, it implies that $Reach_M(s_T, *, 2\Delta) \subseteq \phi_{safe}$ and hence for the next Δ time units, we are guaranteed that $Reach_M(s_T, *, \Delta) \subseteq \phi_{safe}$ holds. ■

The invariant established in Theorem 8.3.1 ensures that if the assumptions of the theorem are satisfied, then all reachable states are always contained in ϕ_{safe}.

Remark 8.3.1: Guarantee switching and avoid oscillation

The liveness property **(P2b)** *guarantees that the system will definitely switch from* N_{sc} *to* N_{ac} *(to maximize performance). Property* **(P3)** *ensures that the system stays in the* AC *mode for some time and not switch back immediately to the* SC *mode. Note that property* **(P2b)** *is not needed for Theorem 8.3.1.*

Remark 8.3.2: AC is a black-box

Our well-formedness check does not involve proving anything about N_{ac}. **(P1a)** *and* **(P1b)** *require that* N_{ac} *samples at most as fast as* N_{dm} *and generates the same outputs as* N_{sc}*, this is for smooth transitioning between* N_{ac} *and* N_{sc}*. We only need to reason about* N_{sc}*, and we need to reason about all possible controller actions (when reasoning with* $Reach_M(s, *, \Delta)$*). The latter is a worst-case analysis and includes* N_{ac}*'s behavior. One could restrict behaviors to* $N_{ac} \cup N_{sc}$ *if we wanted to be more precise, but then* N_{ac} *would not be a black-box anymore.*

Our formalism makes no assumptions about the code (behavior) of the AC node, except that we do need to know the set of all possible output actions (required for doing worst-case reachability analysis). Theorem 3.1 ensures safety as long as all

output actions generated by the code AC (like in Listing 8.1) belong to the assumed set of all possible actions.

Definition 8.3.1: Regions or Set of States for an RTA Module

Let $R(\phi, t) = \{s \mid s \in \phi \wedge \text{Reach}_M(s, *, t) \subseteq \phi\}$. *For example,* $R(\phi_{safe}, \Delta)$ *represents the region or set of states in* ϕ_{safe} *from which all reachable states in time* Δ *are still in* ϕ_{safe}.

Regions of operation of a well-formed RTA module. We informally describe the behavior of an RTA protected module by organizing the state space of the system into different regions of operation (Figure 8.5). R1 represents the unsafe region of operation for the system. Regions R2-R5 represent the safe region, and R3-R5 are the recoverable regions of the state space. The region R3\R4 represents the *switching control* region (from AC to SC) as the time to escape ϕ_{safe} for the states in this region is less than 2Δ.

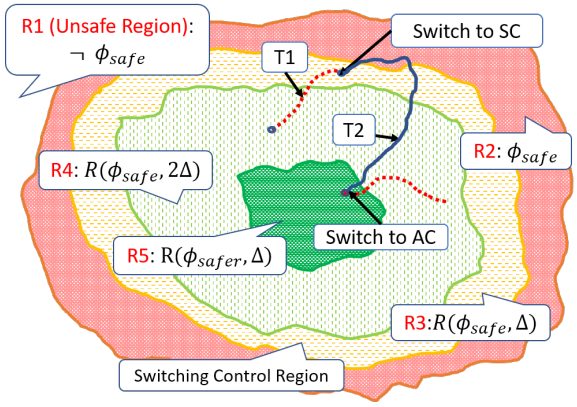


Figure 8.5: Regions of Operation for an RTA Module.

As the DM is guaranteed to sample the state of the system at least once in Δ time (property **(P1a)**), the DM is guaranteed to switch control from AC to SC if the system remains in the switching control region for at least Δ time, which is the case before system can leave region R3. Consider the case where T1 represents a trajectory of the system under the influence of AC when the system is in the switching control region the DM detects the imminent danger and switches control to SC. **(P1a)** ensures that N_{sc} takes control before the system escapes ϕ_{safe} in the next Δ time. Property **(P2a)** ensures that the resultant trajectory T2 of the system remains inside the safe region and Property **(P2b)** ensures that the system eventually enters region R5 where the control can be returned to AC for maximizing the performance of the system. Property **(P3)** ensures that the switch to AC is safe and the system will remain in AC mode for at least Δ time.

Remark 8.3.3: Choosing ϕ_{safer} and Δ

The value of Δ is critical for ensuring safe switching from AC *to* SC*. It also determines how conservatively the system behaves: for example, large value of Δ implies a large distance between boundaries of region R4 and R5 during which* SC *(conservative) is in control. Small values of Δ and a larger R5 region (ϕ_{safer}) can help maximize the use of* AC *but might increase the chances of switching between* AC *and* SC *as the region between the boundaries of R4 and R5 is too small. Currently, we let the programmer choose these values and leave the problem of automatically finding the optimal values as future work.*

From Theory to Practice. We are assuming here that the checks in Property **(P2)** and Property **(P3)** can be performed. The popular approach in control theory is to use reachability analysis when designing an N_{sc} that always keeps the system within a set of safe states. We used existing tools like FastTrack [100] and the Level-Set Toolbox [81].

First, consider the problem of synthesizing the safe controller N_{sc} for a given safe set ϕ_{safe}. N_{sc} can be synthesized using pre-existing safe control synthesis techniques. For example, for motion primitives, we can use a framework like FaSTrack [100] for the synthesis of low-level N_{sc}. Next, we note that the DM needs to reason about the reachable set of states for a system when either the controller is fixed to N_{sc} or is nondeterministic. Again, there are several tools and techniques for performing reachability computations [81]. One particular concept that SOTER requires here is the notion of *time to failure less than* 2Δ ($\texttt{ttf}_{2\Delta}$). The function $\texttt{ttf}_{2\Delta} : \mathcal{S} \times 2^{\mathcal{S}} \to \mathbb{B}$, given a state $s \in \mathcal{S}$ and a predicate $\phi \subseteq \mathcal{S}$ returns *true* if starting from s, the minimum time after which ϕ may not hold is less than or equal to 2Δ. The check $\texttt{Reach}(s_t, *, 2\Delta) \not\subseteq \phi_{safe}$ in Listing 8.3 can be equivalently described using the $\texttt{ttf}_{2\Delta}$ function as $\texttt{ttf}_{2\Delta}(s_t, \phi_{safe})$. Let us revisit the boolean functions `PhiSafer_MPr` and `TTF2D_MPr` from Listing 8.2, these functions correspond to the checks $s_t \in \phi_{safer}$ and $\texttt{ttf}_{2\Delta}(s_t, \phi_{safe})$ respectively.

8.4 OPERATIONAL SEMANTICS OF AN RTA MODULE

Definition 8.4.1: Composable RTA Modules

A set of RTA *modules $S = \{M_0, M_1, \ldots, M_n\}$ are composable if:*
(1) *Nodes in all modules are disjoint, i.e., if N^i_{ac}, N^i_{sc}, and N^i_{dm} represent the* AC*,* SC *and* DM *nodes of a module M_i then, for all i, j s.t. $i \neq j$, $\{N^i_{ac}, N^i_{sc}, N^i_{dm}\} \cap \{N^j_{ac}, N^j_{sc}, N^j_{dm}\} = \emptyset$.*
(2) *Outputs of all modules are disjoint, i.e., for all i, j s.t. $i \neq j$, $O(M_i) \cap O(M_j) = \emptyset$.*

Note that the only constraint for composition is that the outputs (no constraints on inputs) must be disjoint (as discussed in the traditional compositional frameworks like I/O Automata and Reactive Modules [9, 129]).

An RTA system is a set of composable RTA modules. If RTA modules P and Q are composable then their composition $P \parallel Q$ is an RTA system consisting of the two modules $\{P, Q\}$. Also, composition of two RTA systems S1 and S2 is an RTA system $S1 \cup S2$, if all modules in $S1 \cup S2$ are composable.

Theorem 8.4.1: Compositional RTA System

Let $S = \{M_0, \ldots M_n\}$ *be an* RTA *system. If for all* i, M_i *is a well-formed* RTA *module satisfying the safety invariant* ϕ^i_{Inv} *then, S satisfies the invariant* $\bigwedge_i \phi^i_{Inv}$.

Proof. Note that this theorem follows from the fact that composition restricts the environment. Since we are guaranteed output disjointness during composition, the composition of two modules is guaranteed to be language intersection. The proof for such composition theorem is described in details in [9, 129]. ■

Theorem 8.4.1 plays a vital role in building a reliable robotics software stack. The software stack is decomposed such that each component is protected by an RTA module, individually satisfying the respective safety invariant, and their composition satisfies the system-level specification.

Attributes of an RTA system. Given an RTA system $S = \{M_0, \ldots, M_n\}$, its attributes (used for defining the operational semantics) can be inferred as follows[1]:

1. $ACNodes \in \mathcal{N} \to \mathcal{N}$ is a map that binds a DM node n to the particular AC node $ACNodes[n]$ it controls, i.e., if $M_i \in S$ then $(N^i_{dm}, N^i_{ac}) \in ACNodes$.
2. $SCNodes \in \mathcal{N} \to \mathcal{N}$ is a map that binds a DM node n to the particular SC node $SCNodes[n]$ it controls, i.e., if $M_i \in S$ then $(N^i_{dm}, N^i_{sc}) \in SCNodes$.
3. $Nodes \subseteq \mathcal{N}$ represents the set of all nodes in the RTA system, $Nodes = \mathtt{dom}(ACNodes) \cup \mathtt{codom}(ACNodes) \cup \mathtt{codom}(SCNodes)$.
4. $OS \subseteq \mathcal{T}$ represents the set of outputs of the RTA system, $OS = \bigcup_{n \in Nodes} O(n)$.
5. $IS \subseteq \mathcal{T}$ represents the set of inputs of the RTA system (inputs from the environment), $IS = \bigcup_{n \in Nodes} I(n) \setminus OS$.
6. CS represents the calendar or time-table of the RTA system, $CS = \bigcup_{n \in Nodes} C(n)$.

We refer to the attributes of an RTA system S as $ACNodes(S)$, $SCNodes(S)$, $Nodes(S)$, $OS(S)$, $IS(S)$, and $CS(S)$ respectively.

1 Recollect that $\mathtt{dom}(X)$ refers to the domain of map X and $\mathtt{codom}(X)$ refers to the codomain of X.

Note that the semantics of an RTA module is the semantics of an RTA system where the system is a singleton set. We use the timeout-based discrete event simulation model [66] for modeling the semantics of an RTA system. The calendar CS stores the future times at which nodes in the RTA system must step. Using a variable ct to store the current time and FN to store the enabled nodes, we can model the real-time system as a discrete transition system.

Configuration. The configuration of an RTA system is a tuple $(L, OE, ct, FN, Topics)$ where:

1. $L \in Nodes \to \mathcal{L}$ represents a map from a node to the local state of that node.
2. $OE \in \mathcal{N} \to \mathbb{B}$ represents a map from a node to a boolean value indicating whether the output of the node is enabled or disabled. This is used for deciding whether AC or SC should be in control. The domain of OE is $\texttt{codom}(ACNodes) \cup \texttt{codom}(SCNodes)$.
3. $ct \in \mathbb{R}$ represents the current time.
4. $FN \subseteq \mathcal{N}$ represents the set of nodes that are remaining to be fired at time ct.
5. $Topics \in \mathcal{T} \to \mathcal{V}$ is a map from a topic name to the value stored at that topic, it represents the globally visible topics. If $X \subseteq \mathcal{T}$ then $Topics[X]$ represents a map from each $x \in X$ to $Topics[x]$.

The initial configuration of any RTA system is represented as $(L_0, OE_0, ct_0, FN_0, Topics_0)$ where: **(1)** L_0 maps each node in its domain to default local state value l_0, if the node is a DM then $mode = SC$; **(2)** OE_0 maps each SC node to `true` and AC node to `false` (this is to ensure that each RTA module starts in SC mode); **(3)** $ct_0 = 0$; **(4)** $FN_0 = \emptyset$; and **(5)** $Topics_0$ maps each topic name to its default value $v \in \mathcal{V}$.

We represent the operational semantics of a RTA system as a transition relation over its configurations (Figure 8.6). There are two types of transitions: **(1)** discrete transitions that are instantaneous and hence does not change the current time, and **(2)** time-progress transitions that advance the time when no discrete transition is enabled.

ENVIRONMENT-INPUT transitions are triggered by the environment and can happen at any time. It updates any of the input topics $e \in IS$ of the module to (e, v).

DISCRETE-TIME-PROGRESS-STEP represents the time-progress transitions that can be executed when no discrete transitions are enabled **(dt1)**. It updates ct to the next time at which a discrete transition must be executed **(dt2)**. FN is updated to the set of nodes that are enabled and must be executed **(dt3)** at the current time.

DM-STEP and AC-OR-SC-STEP are the discrete transitions of the system. DM-STEP represents the transition of any of the DM nodes in the module. The important operation performed by this transition is to enable or disable the outputs of the AC

ITE(x, y, z) represents if x then y else z

(ENVIRONMENT-INPUT)

$$\frac{e \in \text{IS} \qquad v \in \mathcal{V}}{(L, OE, ct, FN, Topics) \rightarrow (L, OE, ct, FN, Topics[e \mapsto v])}$$

(DISCRETE-TIME-PROGRESS-STEP)

$$\frac{FN = \emptyset^{(dt1)} \; ct' = \min(\{t \mid (x,t) \in CS, t > ct\})^{(dt2)} \quad FN' = \{n \mid (n, ct') \in CS\}^{(dt3)}}{(L, OE, ct, FN, Topics) \rightarrow (L, OE, ct', FN', Topics)}$$

(DM-STEP)

$$\frac{\begin{array}{c} dm \in FN \qquad FN' = FN \setminus \{dm\} \\ dm \in \texttt{dom}(ACNodes) \qquad (l, \{(\texttt{STATE}, s_t)\}, l', \emptyset) \in T(dm) \qquad ac = ACNodes[dm] \\ sc = SCNodes[dm] \qquad \textbf{ITE}(l' = AC, en = \texttt{true}, en = \texttt{false})^{(dm1)} \end{array}}{\begin{array}{c} (L, OE, ct, FN, Topics) \rightarrow \\ (L[dm \mapsto l'], OE[ac \mapsto en, sc \mapsto \neg en]^{(dm2)}, ct, FN', Topics) \end{array}}$$

(AC-OR-SC-STEP)

$$\frac{\begin{array}{c} n \in FN \qquad FN' = FN \setminus \{n\} \\ n \notin \texttt{dom}(ACNodes) \qquad in = Topics[I(n)] \qquad (l, in, l', out) \in T(n) \\ \textbf{ITE}(OE[n], Topics' = out \cup Topics[\mathcal{T} \setminus \texttt{dom}(out)], Topics' = Topics)^{(n1)} \end{array}}{(L, OE, ct, FN, Topics) \rightarrow (L[n \mapsto l'], OE, ct, FN', Topics')}$$

Figure 8.6: Operational Semantics of SOTER

and SC node **(dm2)** based on its current *mode* **(dm1)**. Finally, AC-OR-SC-STEP represents the step of any AC or SC node in the module. Note that the node updates the output topics only if its output is enabled (based on $OE(n)$ **(n1)**).

Reachability. Note that the state space $\mathcal{S}$ of an RTA system is the set of all possible configurations. The set of all possible reachable states of an RTA system is a set of configurations that are reachable from the initial configuration using the transition system described in Figure 8.6. Since the environment transitions are nondeterministic, potentially many states are reachable even if the RTA modules are all deterministic.

Let $Reach_M(s, N_{sc}, t) \subseteq \mathcal{S}$ represent the set of all states of the RTA system S reachable in time $[0, t]$ starting from the state s, using only the controller SC node N_{sc} of the RTA module $M \in S$. In other words, instead of switching control between SC and AC of the RTA module M, the DM always keeps SC node in control. $Reach_M(s, *, t) \subseteq \mathcal{S}$ represents the set of all states of the RTA system S reachable in time $[0, t]$ starting from

the state s, using only a completely nondeterministic module instead of $M \in S$. In other words, instead of module M, a module that generates nondeterministic values on the output topics of M is used. The notation $\mathtt{Reach}$ is naturally extended to a set of states: $\mathtt{Reach}_M(\psi, x, t) = \bigcup_{s \in \psi} \mathtt{Reach}_M(s, x, t)$ is the set of all states reachable in time $[0, t]$ when starting from a state $s \in \psi$ using x. Note that, $\mathtt{Reach}_M(\psi, N_{sc}, t) \subseteq \mathtt{Reach}_M(\psi, *, t)$.

We note that the definition of DM for an RTA module M is sensitive to the choice of the environment for M. Consequently, every attribute of M (such as well-formedness) depends on the context in which M resides. We implicitly assume that all definitions of M are based on a completely nondeterministic context. All results hold for this interpretation, but they also hold for any more constrained environment.

8.5 EVALUATION

We empirically evaluate the SOTER framework by building an RTA-protected software stack (presented in Figure 8.4) that satisfies the safety invariant: $\phi_{plan} \wedge \phi_{mpr} \wedge \phi_{bat}$. The goal of our evaluation is twofold:

(Goal 1) Demonstrate how the SOTER runtime assurance framework can be used for building the software stack compositionally, where each component is guaranteed to satisfy the component-level safety specification. Further, we show how the programmable switching feature of an RTA module can help maximize its performance.

(Goal 2) Empirically validate using rigorous simulations that an RTA-protected software stack can ensure the safety of the drone in the presence of third-party (or machine learning) components, where otherwise, the drone could have crashed.

Implementation and Experimental Setup. We extended the DRONA tool chain (see Section 6.3) with the SOTER runtime assurance component. This involved extending the P language with capabilities to implement periodic *nodes* and RTA modules. The SOTER compiler first checks that all the constructed RTA modules in the program are well-formed and then converts the source-level syntax into C code (extending the P code generator). This code contains statically-defined C array-of-structs and functions for the topics, nodes, and functions declarations. The OE that controls the output of each node is implemented as a shared-global data-structure updated by all the DM in the program. The DRONA runtime implements periodic behavior of each node using OS timers for our experiments, deploying the generated code on a real-time operating system is future work. Since a SOTER program is a multi-rate periodic system, we use a bounded-asynchronous scheduler [76] to explore only those schedules that satisfy the bounded-asynchrony semantics. In this case as well, we leveraged the capability of the P explorer to encode exploration strategies as external scheduler (Chapter 4). When performing systematic testing of the robotics software stack the third-party (untrusted) components that are not implemented in DRONA are replaced by their abstractions implemented in DRONA.

For our experiments on the real drone hardware, we use a 3DR Iris [2] drone that comes with the open-source Pixhawk PX4 [151] autopilot. The simulation results were done in the Gazebo [114] simulator environment that has high fidelity models of Iris drone. For our simulations, we execute the PX4 firmware in the loop.

The videos and other details corresponding to our experiments on real drones are available on https://drona-org.github.io/Drona/.

8.5.1 RTA-*Protected Safe Motion Primitives*

A drone navigates in the 3D space by tracking trajectories between waypoints computed by the motion planner. Given the next waypoint, an appropriate motion primitive is used to track the reference trajectory. Informally, a motion primitive consists of a pre-computed control law (sequence of control actions) that regulates the state of the drone as a function of time.

Failure in the presence of untrusted components. For our experiments in Figure 6.4 and Figure 8.2, we used the motion primitives provided by the PX4 autopilot [151] as our advanced controller and found that it can lead to failures or collision.

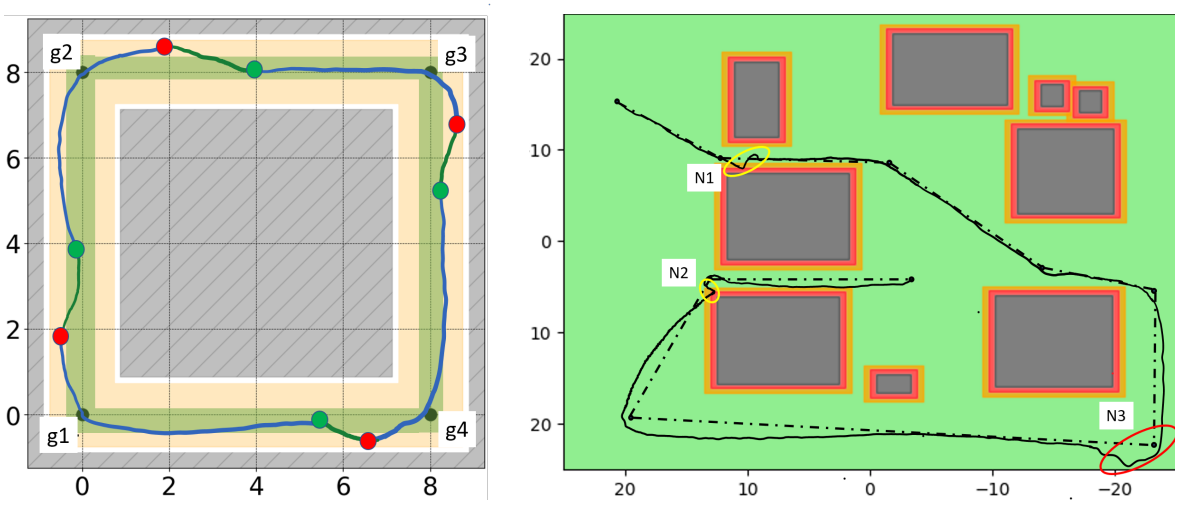
To achieve RTA-protected motion primitive, there are three essential steps: (1) Design of the safe controller N_{sc}; (2) Designing the $\texttt{ttf}_{2\Delta}$ function that controls switching from the AC to SC for the motion primitive; (3) Programming the switching from SC to AC and choosing an appropriate Δ and ϕ_{safer} so that the system is not too conservative.

When designing the N_{sc}, it must satisfy the Property **(P2)**, where ϕ_{safe} is the region not occupied by any obstacle. Techniques from control theory, like *reachability* [139] can be used for designing N_{sc}. We use the FaSTrack [100] tool for generating a correct-by-construction controller for the drone such that it satisfies all the properties required for a N_{sc}.

To design the switching condition from AC to SC, we need to compute the $\texttt{ttf}$ function that checks $\texttt{Reach}(s_t, *, 2\Delta) \not\subseteq \phi_{safe}$ (Listing 8.3) where s_t is the current state. Consider the 2D representation of the workspace in Figure 8.7b. The obstacles (shown in grey) represent the ϕ_{unsafe} region, and any region outside is ϕ_{safe}. Note that, N_{sc} can guarantee safety for all locations in ϕ_{safe} **(P2)**. We can use the level set toolbox [139] to compute the backward reachable set from ϕ_{safe} in 2Δ (shown in yellow), i.e., the set of states from where the drone can leave ϕ_{safe} (collide with an obstacle) in 2Δ. In order to maximize the performance of the system, the RTA module must switch from SC to AC after the system has recovered. In our experiments, we choose $\phi_{safer} = \texttt{R}(arg2, \phi_{safe})2\Delta$ (shown in green). N_{sc} is designed such that given ϕ_{safer}, Property **(P2b)** holds. DM transfers control to AC when it detects that the drone is in ϕ_{safer}, which is the backward reachable set from ϕ_{safe} in 2Δ time.

Choosing the period Δ is an important design decision. Choosing a large Δ can lead to overly-conservative $\texttt{ttf}_{2\Delta}(s_t, \phi_{safe})$ and ϕ_{safer}. In other words, a large Δ pushes the

switching boundaries further away from the obstacle. In which case, a large part of the workspace is covered by red or yellow region where SC (conservative controller) is in control.



(a) Example trajectory demonstrating RTA-enabled safety of the drone (collision avoidance) when flying in a workspace surrounded by obstacles (also see Figure 6.4a). Red dots are points where RTA module switched control from AC to SC to safeguard the system. Green dots are points where the drone had recovered, and the control is returned to the AC.

(b) Example trajectory demonstrating RTA-enabled safety (collision avoidance) during Surveillance Mission. Regions N1, N2 represent cases where the AC takes the drone too close to the obstacle which leads to switching control to SC that bring the drone away from the obstacle into the green region

Figure 8.7: Evaluation of RTA-Protected Motion Primitives

We implemented the safe motion primitive as a RTA module using the components described above. Figure 8.7a presents one of the interesting trajectories where the SC takes control multiple times and ensures the overall correctness of the mission. The green tube inside the yellow tube represents the ϕ_{safer} region. The red dots represent the points where the DM switches control to SC, and the green dots represent the points where the DM returns control back to the AC for optimizing performance. The average time taken by the drone to go from g_1 to g_4 is 10 secs when only the unsafe N_{ac} is in control (can lead to collisions), it is 14 secs when using the RTA protected safe motion primitive, and 24 secs when only using the safe controller. Hence, using RTA provides a "safe" middle ground without sacrificing performance too much.

Figure 8.7b presents the 2D representation of our workspace in Gazebo (Figure 6.1b). The dotted lines represent one of the reference trajectories of the drone during the surveillance mission. The trajectory in solid shows the trajectory of the drone when using the RTA-protected software stack consisting of the safe motion primitive. At N1 and N2, the N_{sc} takes control and pushes the drone back into ϕ_{safer} (green);

and returns control back to N_{ac}. We observe that the N_{ac} is in control for most of the surveillance mission even in cases when the drone deviates from the reference trajectory (N3) but is still safe.

8.5.2 RTA-*Protected Battery Safety*

We want our software stack to provide the battery-safety guarantee, that prioritizes safely landing the drone when the battery charge falls below a threshold level. We first augment the state of the drone with the current battery charge, b_t. N_{ac} is a node that receives the current motion plan from the planner and forwards it to the motion primitives module. N_{sc} is a certified planner that safely lands the drone from its current position. The set of all safe states for the battery safety is given by, $\phi_{safe} := b_t > 0$, i.e., the drone is safe as long as the battery does not run out of charge. We define $\phi_{safer} := b_t > 85\%$, i.e., the battery charge is greater than 85%. Since the battery discharges at a slower rate compared to changes in the position of the drone, we define a larger Δ for the battery RTA compared to the motion primitive RTA.

To design the $\texttt{ttf}_{2\Delta}$, we first define two terms: (1) Maximum battery charge required to land T_{max}; and (2) Maximum battery discharge in 2Δ, $cost^*$. In general, T_{max} depends on the current position of the drone. However, we approximate T_{max} as the battery required to land from the maximum height attained by the drone safely. Although conservative, it is easy to compute and can be done offline. To find $cost^*$, we first define a function $cost$, which given the low-level control to the drone and a time period, returns the amount of battery the drone discharges by applying that control for the given time period. Then, $cost^* = \max_u cost(u, 2\Delta)$ is the maximum discharge that occurs in time 2Δ across all possible controls, u. We can now define $\texttt{ttf}_{2\Delta}(b_t, \phi_{safe}) = b_t - cost^* < T_{max}$. It guarantees that DM switches control to SC if the current battery level may not be sufficient to safely land if AC were to apply the worst possible control. DM returns control to N_{ac} once the drone is sufficiently charged. This is defined by ϕ_{safer}, which is chosen to assert that the battery has at least 85% charge before DM can hand control back to AC. The resultant RTA module is well-formed and satisfies the battery safety property ϕ_{bat}.

We implemented the battery safety RTA module with the components defined above. Figure 8.8 shows a trajectory, where the battery falls below the safety threshold, causing DM to transfer control to N_{sc}, which lands the drone.

8.5.3 RTA *for Safe Motion Planner*

We implemented the motion-planner for our surveillance application using the RRT* [110] algorithm from OMPL. OMPL [184] is a third-party motion-planning library that implements many state-of-the-art sampling-based motion planning algo-

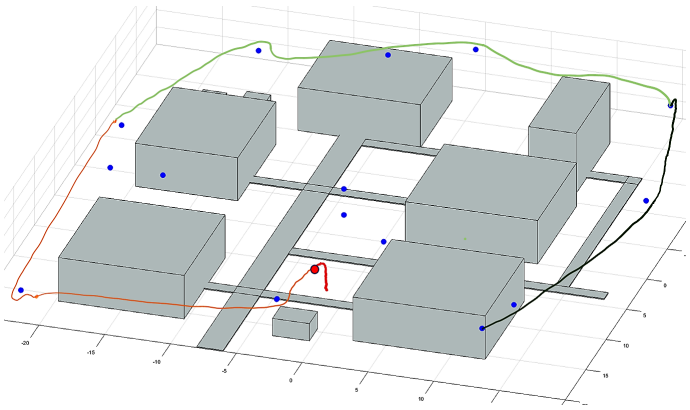


Figure 8.8: Guaranteeing Battery Safety (ϕ_{bat}) using Runtime Assurance

rithms. We injected bugs into the implementation of RRT* such that in some cases, the generated motion plan can collide with obstacles. We wrapped the motion-planner inside an RTA module to ensure that the waypoints generated by motion plan do not collide with an obstacle (violating ϕ_{plan}).

8.5.4 RTA *for Safe Exploration*

When operating in environments which are unknown *a-priori*, a robot faces the challenge of exploring the environment safely and still accomplishing the desired goal. A large body of research, classified as safe exploration [141], focuses on developing techniques to explore the environment safely.

As a case study, we use the RTA approach for decomposing the problem of optimized exploration from the problem of providing a safety guarantee for a robot working in a previously unknown environment. In the previous experiments, the motion planner was aware of the static obstacles in the system. To design the RTA module to safely explore unknown environments, we need to (1) Design RTA components N_{ac}, N_{sc} and ϕ_{safe}, (2) Design the switching condition $ttf_{2\Delta}$ for switching from AC to SC, and (3) Programming the switching from SC to AC and choosing an appropriate Δ and ϕ_{safer}.

In this experiment, N_{ac} is a motion planner designed to explore the environment optimally with the minimum number of steps, and the N_{sc} is responsible for bringing the system to a *a-priori* known part of the environment. ϕ_{safe} is the entire state space outside the obstacles.

If the environment was known *a-priori*, we could have used the reachability based technique proposed in Section 8.5.1. However, in the absence of full knowledge of the environment, we approximate $ttf_{2\Delta} := \{s : s \in \mathcal{S} \text{ s.t. } s + v_{max} \cdot 2 \cdot \Delta \notin \phi_{safe}\}$ where v_{max} is the maximum velocity attainable by the quadrotor in x, y, or z direction. Intuitively, it checks if a state would leave ϕ_{safe} in 2Δ if it were moving with its highest velocity. This function is more conservative compared to $ttf_{2\Delta}$ proposed in

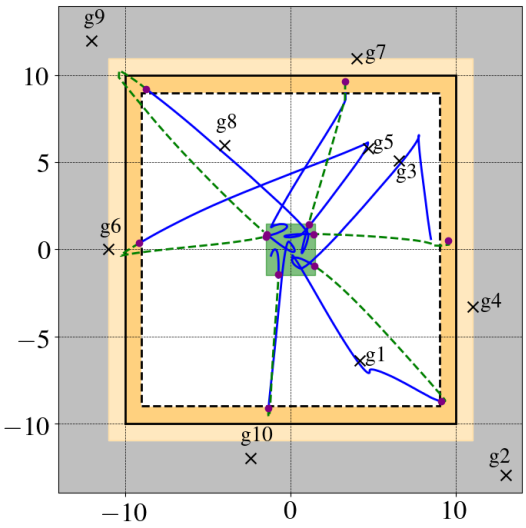


Figure 8.9: Safe exploration using RTA module

Section 5.1 computed using reachability. However, this is fast to compute and can be computed on the fly, making it particularly attractive to be used in a partially observable environment.

Since the environment is unknown, we have to be conservative about our set ϕ_{safer}. In our experiments, ϕ_{safer} is a predefined known area of the state space. The switching from SC to AC occurs at the boundary of the set $R(\phi_{safer}, \Delta)$. Similar to Section 8.5.1, Δ should be chosen to avoid overly-conservative $ttf_{2\Delta}$ and $R(\phi_{safer}, \Delta)$.

We used our RTA module to safely explore an environment (Figure 8.9) by avoiding collision with the surrounding wall in gray whose location is unknown *a-priori*. ϕ_{safe} is the entire workspace contained within the gray wall, $R(\phi_{safer}, \Delta)$ is the green square at the center of the workspace. Additionally, Δ is chosen such that $R(arg2, \phi_{safe})\Delta$ is the square with the black boundary and $R(\phi_{safe}, 2\Delta)$ is the square with the dashed black boundary.

In our experiment, the exploring motion planner generates goal points $g_1 - g_{10}$ (black crosses in Figure 8.9) for the drone to traverse, sequentially. For each goal point, g_i, N_{ac} plans a path from the current position of the quadcopter, $\mathbf{x}_t$ to the g_i. However, during exploration when g_i satisfies $ttf_{2\Delta}$, our RTA module detects the wall at runtime, switches to SC (shown by dot in Figure 8.9) when the trajectory leaves $R(\phi_{safe}, 2\Delta)$ while still inside $R(\phi_{safe}, \Delta)$. N_{sc} brings the trajectory back to ϕ_{safer} (shown by the broken trajectory). Once inside $R(\phi_{safer}, \Delta)$, the DM hands back control to the N_{ac} (shown by dot) and the exploration process begins again.

8.5.5 *Rigorous Simulation*

To demonstrate that runtime assurance helps build robust robotics systems, we conducted rigorous stress testing of the RTA-protected drone software stack. We conducted software in the loop simulations for **104** hours, where an autonomous drone is tasked to visit randomly generated surveillance points in the Gazebo workspace (Figure 6.1) repeatedly. In total, the drone flew for approximately 1505K meters in the 104 hours of simulation. We found that there were 109 disengagements; these are cases where one of the SC nodes took control from AC and avoided a potential failure. There were 34 crashes during the experiments, and we found that in most of these cases the potential danger was detected by the DM node, but the SC node was not scheduled in time for the system to recover. Further study is required to analysis the root cause of these failures, but we believe that some of these crashes can be avoided by running the software stack on a real-time operating system. We also found that as the RTA module is designed to return the control to AC after recovering the system, during our simulations, AC nodes were in control for $> 96\%$ of the time. Thus, safety is ensured without sacrificing the overall performance, and the optimal controller (AC) is in control for the most part of the mission.

Evaluation Summary. We used the theory of well-formed RTA module to construct three RTA modules: motion primitives, battery safety, and motion planner. We leverage Theorem 8.3.1 to ensure that the modules individually satisfy the safety invariants $\phi_{\mathtt{mpr}}$, $\phi_{\mathtt{bat}}$, and $\phi_{\mathtt{plan}}$ respectively. The RTA-protected software stack (Figure 8.4) is a composition of the three modules and using Theorem 8.4.1 we can guarantee that the system satisfies the desired safety invariant $\phi_{\mathtt{plan}} \wedge \phi_{\mathtt{mpr}} \wedge \phi_{\mathtt{bat}}$.

8.6 RELATED WORK

Runtime verification has been applied to robotics [48, 59, 103, 106, 124, 130, 155] where online monitors are used to check the correctness (safety) of the robot at runtime. We refer the reader to the articles [40] presenting detailed survey of runtime verification and assurance techniques applied for safety of robotics and cyber-physical systems.

More recently, Schierman *et al.* [171] investigated how the RTA framework can be used at different levels of the software stack of an unmanned aircraft system. In a more recent work [156], Schierman proposed a component-based simplex architecture (CBSA) that combines assume-guarantee contracts with RTA for assuring the runtime safety of component-based cyber-physical systems. Note that most prior applications of RTA do not provide high-level programming language support for constructing provably-safe RTA systems in a compositional fashion while designing for *timing and communication behavior* of such systems. They are all instances of using RTA as a *design methodology* for building reliable systems in the presence of untrusted components. We

take inspiration from them, and integrated these design methodologies into a practical programming framework.

In [20], the authors apply simplex approach for sandboxing cyber-physical systems and present automatic reachability based approaches for inferring switching conditions. The idea of using an advanced controller (AC) under nominal conditions; while at the boundaries, using optimal safe control (SC) to maintain safety has also been used in [6] for operating quadrotors in the real world. In [16] the authors use a switching architecture ([17]) to switch between a nominal safety model and learned performance model to synthesize policies for a quadrotor to follow a trajectory (more examples in the survey articles [40]). Our rigorous simulation results demonstrates that

8.7 SUMMARY

In this chapter, we have presented a new run-time assurance (RTA) framework for programming safe robotics systems. In contrast with other RTA frameworks, SOTER provides (1) a programming language for modular implementation of safe robotics systems by combining each advanced controller with a safe counterpart; (2) theoretical results showing how to safely switch between advanced and safe controllers, and (3) experimental results demonstrating SOTER on drone platforms in both simulation and in hardware.

Part IV

CONCLUSION

9

CONCLUSION

In this chapter, we conclude this dissertation by reflecting on the contributions of this thesis and the lessons learned. We also discuss directions for future work.

9.1 CLOSING THOUGHTS

In this thesis, we considered the challenge of building a programming framework that enables the developers to build safe event-driven asynchronous systems.

As a first contribution, we presented MODP, a programming framework that enables assume-guarantee style compositional reasoning of event-driven asynchronous systems. Chapter 2 presented the novel theory of compositional refinement supported by the MODP module system and Chapter 3 demonstrated the efficacy of using the theory in practice for building a reliable distributed systems software stack. Our results showed that compositional reasoning can help scale systematic testing to large distributed systems. MODP is now being used for the compositional model-checking of distributed services inside Amazon Web Services (AWS).

The second contribution of this thesis is the new approaches for scalable analysis of event-driven asynchronous systems. In Chapter 4, we presented *delaying explorer*, a programmable search prioritization technique for systematic testing of asynchronous programs. Our results showed that delaying explorers beat most of the popular approaches for concurrency testing and also led to the observation that no unique search strategy wins (in terms of finding bugs faster) for all our benchmarks. This was the inspiration for the P# [47] tool for using a portfolio approach where a collection fo different search strategies are executed in parallel, each targetting a different part of the search space. Developers use P# inside Microsoft Azure for implementing and testing some of the core distributed services.

Next, we introduced *approximate synchrony* (Chapter 5), a sound abstraction for verification of almost-synchronous systems. We presented an iterative algorithm for computing this abstraction using model-checking. Using approximate synchrony, we verified the correctness of the IEEE 1588 protocol and also in the process, found a liveness bug that was well appreciated by the standards committee.

Finally, we considered the problem of building autonomous robotics systems with formal guarantees of correctness. We presented two frameworks, DRONA (Chapter 7) for programming distributed mobile robotics systems and SOTER (Chapter 8) that uses runtime assurance for guaranteeing safety of robotics systems in the presence of untrusted software components. We implemented an autonomous drone software stack using these frameworks and presented results both in simulation and on actual drone platforms.

We share some of the lessons learned when trying to get MODP adopted for building software both in academia and in industry (Microsoft and Amazon).

- *"Connecting specifications to executable code is important"*: Even when the goal is less ambitious than full proof, it is still essential to have a connection between high-level models/specifications and the executable code, and keep them in synchrony.
- *"It is not just about finding bugs"*: Modeling and coding proceed together, verification and testing tools must run every time code is checked in. Hence, these frameworks must be designed with the goal of being integrated into continuous integration or a build system.
- *"Nondeterminism is pervasive in concurrent and distributed systems"*: In order to avoid the problematic sources of nondeterminism in systems considered in this thesis, they must be designed in a principled way from the start with formal methods used in design not just in verification.

9.2 FUTURE WORK

We conclude with a discussion of future research directions influenced by the work presented in this thesis.

More applications. We presented three programming frameworks in this thesis: MODP for compositional programming of asynchronous systems, DRONA for programming distributed mobile robotics systems, and SOTER that integrates runtime assurance into DRONA for safe autonomy. An important subject of future work is to build more real-world applications using these frameworks and further evaluate its efficacy; this will, in turn, open up other directions of research.

A unified framework that supports both model-checking and deductive verification. The Modular P (MODP) programming framework presented in this thesis supports a model-checking backend for systematic testing of complex asynchronous programs. These techniques are excellent for finding bugs in the protocol logic and perform high coverage testing for a finite test-harness but cannot prove correctness.

As a next step, we would like to build a verifier that can perform deductive verification of the high-level protocols implemented in MODP. The vision is to have a unified

framework that provides a high-level programming language with an automated reasoning backend (based on model-checking), a verifier for proving correctness (based on deductive verification), and finally, a compiler that generates executable code. We imagine a world were developers will model the protocol design using a high-level language (like P), write specifications, and use automated reasoning (model checking) to validate the design for a finite scenarios. This has low overhead to adoption as developers can use automated "push-button" tools that require limited expertise in formal methods. If the component is more critical, then an expert can re-use the models and specifications provided by the developer to do proofs using the backend verifier. Having a unified framework can also enable leveraging the model-checker as an aid to the deductive verifier, for falsifying the invariant or for synthesizing it by leveraging the recent advances on algorithmic program synthesis [179].

Achieving assured autonomy. In Chapter 8, we presented a programming framework that allows the programmers to build safe autonomous systems in the presence of untrusted or hard-to-verify components. As autonomous systems become a reality, their dependence on machine-learning and other data-driven techniques is bound to increase. The solution for building these systems with formal guarantees, as verifying such components is hard, is to use runtime assurance techniques. For future work, we are investigating the role a system like SOTER can play in the design and implementation of verified learning-based robotics, and more generally, for verified artificial intelligence [63, 178], where we believe runtime assurance will play a central role.

BIBLIOGRAPHY

[1] 802.15.4e 2012. "IEEE Standard for Local and metropolitan area networks-Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer." In: (2012).

[2] *3D Robotics*. https://3dr.com/. 2017.

[3] Martín Abadi and Leslie Lamport. "Conjoining Specifications." In: *ACM Trans. Program. Lang. Syst.* (1995).

[4] Tesnim Abdellatif, Saddek Bensalem, Jacques Combaz, Lavindra de Silva, and Felix Ingrand. "Rigorous design of robot software: A formal component-based approach." In: *Robotics and Autonomous Systems* 60.12 (2012), pp. 1563 –1578. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2012.09.005. URL: http://www.sciencedirect.com/science/article/pii/S0921889012001510.

[5] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-01092-5.

[6] A. K. Akametalu, J. F. Fisac, J. H. Gillula, S. Kaynama, M. N. Zeilinger, and C. J. Tomlin. "Reachability-based safe learning with Gaussian processes." In: *53rd IEEE Conference on Decision and Control*. 2014, pp. 1424–1431. DOI: 10.1109/CDC.2014.7039601.

[7] Akka. *Akka Programming Language*. http://akka.io/. 2017.

[8] Rajeev Alur and David L Dill. "A theory of timed automata." In: *Theoretical computer science* 126.2 (1994), pp. 183–235.

[9] Rajeev Alur and Thomas A. Henzinger. "Reactive Modules." English. In: *Formal Methods in System Design* 15.1 (1999), pp. 7–48. ISSN: 0925-9856.

[10] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, P-H Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. "The algorithmic analysis of hybrid systems." In: *Theoretical computer science* (1995).

[11] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. "MOCHA: Modularity in Model Checking." In: *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*. 1998, pp. 521–525.

[12] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. "Behavioral types in programming languages." In: *Foundations and Trends® in Programming Languages* 3.2-3 (2016), pp. 95–230.

[13] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. ISBN: 193435600X, 9781934356005.

[14] Thomas Arts, Laura M Castro, and John Hughes. "Testing erlang data types with quviq quickcheck." In: *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*. ACM. 2008, pp. 1–8.

[15] *Astar Algorithm Cpp Github*. https://github.com/justinhj/astar-algorithm-cpp.git. 2017.

[16] Anil Aswani, Patrick Bouffard, and Claire Tomlin. "Extensions of learning-based model predictive control for real-time application to a quadrotor helicopter." In: *2012 American Control Conference (ACC)*. IEEE. 2012, pp. 4661–4666.

[17] Anil Aswani, Humberto Gonzalez, S. Shankar Sastry, and Claire Tomlin. "Provably Safe and Robust Learning-based Model Predictive Control." In: *Automatica* 49.5 (May 2013), pp. 1216–1226. ISSN: 0005-1098. DOI: 10.1016/j.automatica.2013.02.003. URL: http://dx.doi.org/10.1016/j.automatica.2013.02.003.

[18] PaulC. Attie and NancyA. Lynch. "Dynamic Input/Output Automata: A Formal Model for Dynamic Systems." In: *CONCUR 2001*. Ed. by KimG. Larsen and Mogens Nielsen. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001.

[19] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. "The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety." In: *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. 2009, pp. 99–107. DOI: 10.1109/RTAS.2009.20.

[20] S. Bak, K. Manamcheri, S. Mitra, and M. Caccamo. "Sandboxing Controllers for Cyber-Physical Systems." In: *2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*. 2011, pp. 3–12. DOI: 10.1109/ICCPS.2011.25.

[21] Timothy D Barfoot. *State Estimation for Robotics*. Cambridge University Press, 2017.

[22] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. "Satisfiability Modulo Theories." In: *Handbook of Satisfiability*. Ed. by Armin Biere, Hans van Maaren, and Toby Walsh. Vol. 4. IOS Press, 2009. Chap. 8.

[23] Saddek Bensalem, Lavindra de Silva, Félix Ingrand, and Rongjie Yan. "A verifiable and correct-by-construction controller for robot functional levels." In: *arXiv preprint arXiv:1309.0442* (2013).

[24] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. *Orleans: Distributed Virtual Actors for Programmability and Scalability*. Tech. rep. 2014.

[25] Gérard Berry, S Ramesh, and RK Shyamasundar. "Communicating reactive processes." In: *Proceedings of POPL*. 1993.

[26] Colin Blundell, Dimitra Giannakopoulou, and Corina S. Păsăreanu. "Assume-guarantee Testing." In: *SIGSOFT Softw. Eng. Notes* (2006).

[27] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. "VeriPhy: Verified Controller Executables from Verified Cyber-physical System Models." In: *SIGPLAN Not.* 53.4 (June 2018), pp. 617–630. ISSN: 0362-1340. DOI: 10.1145/3296979.3192406. URL: http://doi.acm.org/10.1145/3296979.3192406.

[28] Edwin Brady. "State Machines All The Way Down An Architecture for Dependently Typed Applications." In: (2016).

[29] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG].

[30] David Broman, Patricia Derler, Ankush Desai, John C. Eidson, and Sanjit A. Seshia. "Endlessly Circulating Messages in IEEE 1588-2008 Systems." In: *Proceedings of the 8th International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication (ISPCS)*. 2014.

[31] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. "A randomized scheduler with probabilistic guarantees of finding bugs." In: *Proceedings of ASPLOS*. 2010.

[32] Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin. *Orleans: A Framework for Cloud Computing*. Tech. rep. 2010.

[33] Michal Cáp, Peter Novák, Martin Seleckỳ, Jan Faigl, and Jiff Vokffnek. "Asynchronous decentralized prioritized planning for coordination in multi-robot system." In: *International Conference on Intelligent Robots and Systems*. IEEE. 2013, pp. 3822–3829.

[34] P. Caspi, C. Mazuet, and N. R. Paligot. "About the Design of Distributed Control Systems: The Quasi-Synchronous Approach." In: *SAFECOMP*. 2001.

[35] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. "Foundations of session types." In: *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*. ACM. 2009, pp. 219–230.

[36] S. Chandra, B. Richards, and J. R. Larus. "Teapot: a domain-specific language for writing cache coherence protocols." In: *IEEE Transactions on Software Engineering* (1999).

[37] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective." In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. PODC '07. Portland, Oregon, USA: ACM, 2007, pp. 398–407.

[38] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. "Using Crash Hoare Logic for Certifying the FSCQ File System." In: *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. 2015.

[39] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. "Flow*: An Analyzer for Non-linear Hybrid Systems." In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 258–263.

[40] Matthew Clark, Xenofon Koutsoukos, Ratnesh Kumar, Insup Lee, George Pappas, Lee Pike, Joseph Porter, and Oleg Sokolsky. *Study on Run Time Assurance for Complex Cyber Physical Systems*. Tech. rep. ADA585474. Available at https://leepike.github.io/pubs/RTA-CPS.pdf. Air Force Research Lab, 2013.

[41] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. "Bounded partial-order reduction." In: *Proceedings of OOPSLA 2013*.

[42] James C. Corbett et al. "Spanner: Google's Globally-distributed Database." In: *Proceedings of OSDI 2012*.

[43] Véronique Cortier and Stéphanie Delaune. "A method for proving observational equivalence." In: *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*. IEEE. 2009, pp. 266–276.

[44] Conrado Daws and Sergio Yovine. "Two examples of verification of multirate timed automata with Kronos." In: *Proceedings of RTSS*. 1995.

[45] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008, pp. 337–340.

[46] Jonathan A. DeCastro, Javier Alonso-Mora, Vasu Raman, Daniela Rus, and Hadas Kress-Gazit. "Collision-Free Reactive Mission and Motion Planning for Multi-Robot Systems." In: *International Symposium on Robotics Research (ISRR)*. Sestri Levante, Italy, 2015.

[47] Pantazis Deligiannis, Alastair F Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. "Asynchronous programming, analysis and testing with state machines." In: *ACM SIGPLAN Notices*. Vol. 50. 6. ACM. 2015, pp. 154–164.

[48] Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. "Combining Model Checking and Runtime Verification for Safe Robotics." In: *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*. 2017, pp. 172–189. DOI: 10.1007/978-3-319-67531-2_11. URL: https://doi.org/10.1007/978-3-319-67531-2_11.

[49] Ankush Desai and Shaz Qadeer. "P: Modular and Safe Asynchronous Programming." In: *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*. 2017, pp. 3–7. DOI: 10.1007/978-3-319-67531-2_1. URL: https://doi.org/10.1007/978-3-319-67531-2_1.

[50] Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. "Systematic testing of asynchronous reactive systems." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 2015, pp. 73–83. DOI: 10.1145/2786805.2786861. URL: https://doi.org/10.1145/2786805.2786861.

[51] Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. "Programming Safe Robotics Systems: Challenges and Advances." In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II*. 2018, pp. 103–119. DOI: 10.1007/978-3-030-03421-4_8. URL: https://doi.org/10.1007/978-3-030-03421-4_8.

[52] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. "P: Safe Asynchronous Event-Driven Programming." In: *Proceedings of PLDI*. 2013.

[53] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. "P: safe asynchronous event-driven programming." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 321–332. DOI: 10.1145/2491956.2462184. URL: https://doi.org/10.1145/2491956.2462184.

[54] Ankush Desai, David Broman, John Eidson, Shaz Qadeer, and Sanjit A. Seshia. *Approximate Synchrony: An Abstraction for Distributed Time-Synchronized Systems*. Tech. rep. UCB/EECS-2014-136. University of California, Berkeley, 2014.

[55] Ankush Desai, Sanjit A. Seshia, Shaz Qadeer, David Broman, and John C. Eidson. "Approximate Synchrony: An Abstraction for Distributed Almost-Synchronous Systems." In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part*

II. 2015, pp. 429–448. DOI: 10.1007/978-3-319-21668-3_25. URL: https://doi.org/10.1007/978-3-319-21668-3_25.

[56] Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A. Seshia. "DRONA: a framework for safe distributed mobile robotics." In: *Proceedings of the 8th International Conference on Cyber-Physical Systems, ICCPS 2017, Pittsburgh, Pennsylvania, USA, April 18-20, 2017*. 2017, pp. 239–248. DOI: 10.1145/3055004.3055022. URL: https://doi.org/10.1145/3055004.3055022.

[57] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. "Compositional programming and testing of dynamic distributed systems." In: *PACMPL* 2.OOPSLA (2018), 159:1–159:30. DOI: 10.1145/3276529. URL: https://doi.org/10.1145/3276529.

[58] *SOTER: A Runtime Assurance Framework for Programming Safe Robotics Systems*. IEEE Computer Society, 2019.

[59] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. "Robust online monitoring of signal temporal logic." In: *Formal Methods in System Design* 51.1 (2017), pp. 5–30. ISSN: 1572-8102. DOI: 10.1007/s10703-017-0286-7. URL: https://doi.org/10.1007/s10703-017-0286-7.

[60] Mariangiola Dezani-Ciancaglini and Ugo De'Liguoro. "Sessions and session types: An overview." In: *International Workshop on Web Services and Formal Methods*. Springer. 2009, pp. 1–28.

[61] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. "On the Minimal Synchronism Needed for Distributed Consensus." In: *J. ACM* 34.1 (Jan. 1987), pp. 77–97. ISSN: 0004-5411. DOI: 10.1145/7531.7533. URL: http://doi.acm.org/10.1145/7531.7533.

[62] Tommaso Dreossi, Alexandre Donzé, and Sanjit A. Seshia. "Compositional Falsification of Cyber-Physical Systems with Machine Learning Components." In: *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*. 2017, pp. 357–372. DOI: 10.1007/978-3-319-57288-8_26. URL: https://doi.org/10.1007/978-3-319-57288-8_26.

[63] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia. "VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems." In: *31st International Conference on Computer Aided Verification (CAV)*. July 2019.

[64] *Drona Website*. https://drona-org.github.io/Drona/. 2019.

[65] Parasara Sridhar Duggirala, Sayan Mitra, Mahesh Viswanathan, and Matthew Potok. "C2E2: a verification tool for stateflow models." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2015, pp. 68–82.

[66] Bruno Dutertre and Maria Sorea. "Modeling and Verification of a Fault-Tolerant Real-Time Startup Protocol Using Calendar Automata." In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Ed. by Yassine Lakhnech and Sergio Yovine. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 199–214. ISBN: 978-3-540-30206-3.

[67] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the Presence of Partial Synchrony." In: *J. ACM* (1988).

[68] John Eidson and Kang Lee. "IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems." In: *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*. Ieee. 2002, pp. 98–105.

[69] Ásgeir Th. Eiríksson. "The Formal Design of 1M-gate ASICs." In: *Form. Methods Syst. Des.* (2000).

[70] Tayfun Elmas, Jacob Burnim, George Necula, and Koushik Sen. "CONCURRIT: A domain specific language for reproducing concurrency bugs." In: *Proceedings of PLDI*. 2013.

[71] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. "Delay-bounded scheduling." In: *Proceedings of POPL*. 2011.

[72] Michael Erdmann and Tomas Lozano-Perez. "On Multiple Moving Objects." In: *Algorithmica* 2 (1986), pp. 1419–1424.

[73] Georgios E Fainekos, Antoine Girard, Hadas Kress-Gazit, and George J Pappas. "Temporal logic motion planning for dynamic robots." In: *Automatica* (2009).

[74] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. "On the relationship between concurrent separation logic and assume-guarantee reasoning." In: *European Symposium on Programming*. Springer. 2007, pp. 173–188.

[75] C. Finucane, Gangyuan Jing, and H. Kress-Gazit. "LTLMoP: Experimenting with language, Temporal Logic and robot control." In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010.

[76] Jasmin Fisher, Thomas A. Henzinger, Maria Mateescu, and Nir Piterman. "Bounded Asynchrony: Concurrency for Modeling Cell-Cell Interactions." In: *Formal Methods in Systems Biology*. Ed. by Jasmin Fisher. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 17–32. ISBN: 978-3-540-68413-8.

[77] Jasmin Fisher, Thomas A. Henzinger, Dejan Nickovic, Nir Piterman, Anmol V. Singh, and Moshe Y. Vardi. "Dynamic Reactive Modules." In: *CONCUR 2011 – Concurrency Theory: 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*. Ed. by Joost-Pieter Katoen and Barbara König. 2011.

[78] Robert W Floyd. "Assigning meanings to programs." In: *Program Verification*. Springer, 1993, pp. 65–81.

[79] Cédric Fournet and Georges Gonthier. "The Reflexive CHAM and the Join-calculus." In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1996.

[80] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. "A Calculus of Mobile Agents." In: *Proceedings of the 7th International Conference on Concurrency Theory*. CONCUR '96. 1996.

[81] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. "SpaceEx: Scalable Verification of Hybrid Systems." In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 379–395.

[82] Ivan Gavran, Rupak Majumdar, and Indranil Saha. "ANTLAB: a multi-robot task server." In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.5s (2017), p. 190.

[83] Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. "Rely/guarantee reasoning for asynchronous programs." In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 42. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[84] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, 1996.

[85] Patrice Godefroid. "Model Checking for Programming Languages using Verisoft." In: *Proceedings of POPL*. 1997, pp. 174–186.

[86] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. ISBN: 0262035618, 9780262035613.

[87] Jim Gray. "Notes on Data Base Operating Systems." In: *Operating Systems, An Advanced Course*. London, UK, UK, 1978, pp. 393–481.

[88] Jim Gray and Leslie Lamport. "Consensus on Transaction Commit." In: *ACM Trans. Database Syst.* 31.1 (Mar. 2006), pp. 133–160.

[89] Jeremie Guiochet, Mathilde Machin, and Helene Waeselynck. "Safety-critical advanced robots: A survey." In: *Robotics and Autonomous Systems* 94 (2017), pp. 43 –52. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2017.04.004. URL: http://www.sciencedirect.com/science/article/pii/S0921889016300768.

[90] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. "Practical software model checking via dynamic interface reduction." In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. 2011, pp. 265–278.

[91] Yi Guo and L. E. Parker. "A distributed and optimal motion planning approach for multiple mobile robots." In: *International Conference on Robotics and Automation (ICRA)*. Vol. 3. 2002, pp. 2612–2619.

[92] Nicolas Halbwachs and Louis Mandel. "Simulation and Verification of Asynchronous Systems by Means of a Synchronous Model." In: *Proceedings of ACSD*. 2006.

[93] David Harel. "Statecharts: A Visual Formalism for Complex Systems." In: *Sci. Comput. Program.* (1987).

[94] P. E. Hart, N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." In: *IEEE Transaction on Systems Science and Cybernetics* (1968).

[95] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. "Ironclad Apps: End-to-end Security via Automated Full-system Verification." In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2014.

[96] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. "IronFleet: Proving Practical Distributed Systems Correct." In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles*. 2015.

[97] Matthew Hennessy and James Riely. "Resource Access Control in Systems of Mobile Agents." In: *Inf. Comput.* 173.1 (Feb. 2002), pp. 82–120. ISSN: 0890-5401. DOI: 10.1006/inco.2001.3089. URL: http://dx.doi.org/10.1006/inco.2001.3089.

[98] Thomas A Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. "Giotto: A time-triggered language for embedded programming." In: *International Workshop on Embedded Software*. Springer. 2001, pp. 166–184.

[99] Thomas A. Henzinger, Xiaojun Liu, Shaz Qadeer, and Sriram K. Rajamani. "Formal Specification and Verification of a Dataflow Processor Array." In: *Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided Design*. 1999.

[100] S. L. Herbert, M. Chen, S. Han, S. Bansal, J. F. Fisac, and C. J. Tomlin. "FaSTrack: A modular framework for fast and guaranteed safe motion planning." In: *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. 2017, pp. 1517–1522. DOI: 10.1109/CDC.2017.8263867.

[101] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Communications of the ACM* (1969).

[102] C. A. R. Hoare. "Communicating Sequential Processes." In: *Communications of the ACM* (1978).

[103] Andreas G. Hofmann and Brian Charles Williams. "Robust Execution of Temporally Flexible Plans for Bipedal Walking Devices." In: *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*. 2006, pp. 386–389. URL: http://www.aaai.org/Library/ICAPS/2006/icaps06-047.php.

[104] G. Holzmann. "The Model Checker SPIN." In: *IEEE Transactions on Software Engineering* (1997).

[105] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty Asynchronous Session Types." In: *J. ACM* 63.1 (Mar. 2016).

[106] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. "ROSRV: Runtime Verification for Robots." In: *Runtime Verification*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Cham: Springer International Publishing, 2014, pp. 247–254. ISBN: 978-3-319-11164-3.

[107] Xiaowan Huang, Anu Singh, and Scott A Smolka. "Using Integer Clocks to Verify the Timing-Sync Sensor Network Protocol." In: *Proceedings of NFM*. 2010.

[108] John Hughes, Benjamin C Pierce, Thomas Arts, and Ulf Norell. "Mysteries of dropbox: property-based testing of a distributed synchronization service." In: *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*. IEEE. 2016, pp. 135–145.

[109] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey." In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[110] Sertac Karaman and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning." In: *The International Journal of Robotics Research* 30.7 (2011), pp. 846–894. DOI: 10.1177/0278364911406761. eprint: https://doi.org/10.1177/0278364911406761. URL: https://doi.org/10.1177/0278364911406761.

[111] Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. "Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code." In: *Symposium on Networked Systems Design and Implementation*. 2007.

[112] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. "Mace: language support for building distributed systems." In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 2007, pp. 179–188.

[113] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel." In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. 2009.

[114] Nathan Koenig and Andrew Howard. "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator." In: *In IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2004, pp. 2149–2154.

[115] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. "Temporal logic based reactive mission and motion planning." In: *IEEE Transactions on Robotics* (2009).

[116] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. "Temporal-logic-based reactive mission and motion planning." In: *IEEE transactions on robotics* 6 (2009), pp. 1370–1381.

[117] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[118] Leslie Lamport. "The part-time parliament." In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169.

[119] Leslie Lamport. "Paxos Made Simple." In: *ACM SIGACT News* 32.4 (Dec. 2001).

[120] Kim G Larsen, Paul Pettersson, and Wang Yi. "UPPAAL in a nutshell." In: *International journal on software tools for technology transfer* 1.1-2 (1997), pp. 134–152.

[121] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. "A Framework for State-Space Exploration of Java-Based Actor Programs." In: *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*. 2009.

[122] Edward Ashford Lee and Sanjit A Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. MIT Press, 2017.

[123] K Rustan M Leino and Peter Müller. "A basis for verifying multi-threaded programs." In: *European Symposium on Programming*. Springer. 2009, pp. 378–393.

[124] Hui X. Li and Brian C. Williams. "Generative Planning for Hybrid Systems Based on Flow Tubes." In: *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*. 2008, pp. 206–213. URL: http://www.aaai.org/Library/ICAPS/2008/icaps08-026.php.

[125] Yixiao Lin and Sayan Mitra. "StarL: Towards a Unified Framework for Programming, Simulating and Verifying Distributed Robotic Systems." In: *Languages, Compilers and Tools for Embedded Systems (LCTES)*. 2015, 9:1–9:10.

[126] M. Lipinski, T. Wlostowski, J. Serrano, P. Alvarez, J.D. Gonzalez Cobas, A. Rubini, and P. Moreira. "Performance results of the first White Rabbit installation for CNGS time transfer." In: *Proceedings of ISPCS*. 2012.

[127] Yanhong A Liu, Scott D Stoller, Bo Lin, and Michael Gorbovitski. "From clarity to efficiency for distributed algorithms." In: *ACM SIGPLAN Notices*. Vol. 47. 10. ACM. 2012, pp. 395–410.

[128] Nancy A. Lynch and Mark R. Tuttle. "Hierarchical Correctness Proofs for Distributed Algorithms." In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 1987.

[129] Nancy A Lynch and Mark R Tuttle. "An introduction to input/output automata." In: (1988).

[130] Lola Masson, Jérémie Guiochet, Hélène Waeselynck, Kalou Cabrera, Sofia Cassel, and Martin Törngren. "Tuning Permissiveness of Active Safety Monitors for Autonomous Systems." In: *NASA Formal Methods*. Ed. by Aaron Dutle, César Muñoz, and Anthony Narkawicz. Cham: Springer International Publishing, 2018, pp. 333–348. ISBN: 978-3-319-77935-5.

[131] Caitie McCaffrey. "The Verification of a Distributed System." In: *Commun. ACM* 59.2 (Jan. 2016).

[132] Kenneth L. McMillan. "A methodology for hardware verification using compositional model checking." In: *Sci. Comput. Program.* 37.1-3 (2000), pp. 279–309.

[133] Kenneth Lauchlin McMillan. "Symbolic Model Checking: An Approach to the State Explosion Problem." PhD thesis. Pittsburgh, PA, USA, 1992.

[134] Kenneth Lauchlin McMillan. *SMV Model Checker*. http://www.kenmcmil.com/smv.html. 2017.

[135] Kenneth McMillan. "Modular specification and verification of a cache-coherent interface." In: *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc. 2016, pp. 109–116.

[136] Daniel Mellinger and Vijay Kumar. "Minimum snap trajectory generation and control for quadrotors." In: *International Conference on Robotics and Automation (ICRA)*. 2011, pp. 2520–2525.

[137] R. Milner. *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982. ISBN: 0387102353.

[138] Robin Milner, Joachim Parrow, and David Walker. "A Calculus of Mobile Processes, I." In: *Inf. Comput.* 100.1 (Sept. 1992).

[139] I. M. Mitchell, A. M. Bayen, and C. J. Tomlin. "A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games." In: *IEEE Transactions on Automatic Control* 50.7 (2005), pp. 947–957. ISSN: 0018-9286. DOI: 10.1109/TAC.2005.851439.

[140] Stefan Mitsch and André Platzer. "ModelPlex: verified runtime validation of verified cyber-physical system models." In: *Formal Methods in System Design* 49.1 (2016), pp. 33–74. ISSN: 1572-8102. DOI: 10.1007/s10703-016-0241-z. URL: https://doi.org/10.1007/s10703-016-0241-z.

[141] Teodor Mihai Moldovan and Pieter Abbeel. "Safe Exploration in Markov Decision Processes." In: *CoRR* abs/1205.4810 (2012). URL: http://arxiv.org/abs/1205.4810.

[142] Iulian Moraru, David G Andersen, and Michael Kaminsky. *EPaxos Code*. https://github.com/efficient/epaxos/. 2013.

[143] Iulian Moraru, David G. Andersen, and Michael Kaminsky. "There is More Consensus in Egalitarian Parliaments." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*. 2013.

[144] Madan Musuvathi and Shaz Qadeer. *Partial-order reduction for context-bounded state exploration*. Tech. rep. MSR-TR-2007-12. Microsoft Research, 2012.

[145] Madanlal Musuvathi and Shaz Qadeer. "Iterative Context Bounding for Systematic Testing of Multithreaded Programs." In: *Proceedings of PLDI*. 2007.

[146] Madanlal Musuvathi and Shaz Qadeer. "Iterative context bounding for systematic testing of multithreaded programs." In: *Proceedings of PLDI*. 2007.

[147] Santosh Nagarakatte, Sebastian Burckhardt, Milo M.K. Martin, and Madanlal Musuvathi. "Multicore Acceleration of Priority-based Schedulers for Concurrency Bug Detection." In: *Proceedings of PLDI 2012*.

[148] Srinivas Nedunuri, Sailesh Prabhu, Mark Moll, Swarat Chaudhuri, and Lydia E Kavraki. "SMT-based synthesis of integrated task and motion plans from plan outlines." In: *International Conference on Robotics and Automation (ICRA)*. IEEE. 2014, pp. 655–662.

[149] Peter W O'Hearn. "Resources, concurrency, and local reasoning." In: *Theoretical computer science* 375.1-3 (2007), pp. 271–307.

[150] P-GitHub. *The P Programming Langugage*. https://github.com/p-org/P. 2019.

[151] *PX4 Autopilot*. https://pixhawk.org/. 2017.

[152] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. "Ivy: Safety Verification by Interactive Generalization." In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. 2016.

[153] Yash Vardhan Pant, Houssam Abbas, Rhudii A. Quaye, and Rahul Mangharam. "Fly-by-logic: Control of Multi-drone Fleets with Temporal Logic Objectives." In: *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. ICCPS '18. Porto, Portugal: IEEE Press, 2018, pp. 186–197. ISBN: 978-1-5386-5301-2. DOI: 10.1109/ICCPS.2018.00026. URL: https://doi.org/10.1109/ICCPS.2018.00026.

[154] Radia Perlman. "An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN." In: *Proceedings of SIGCOMM*. 1985.

[155] Ola Pettersson. "Execution monitoring in robotics: A survey." In: *Robotics and Autonomous Systems* 53.2 (2005), pp. 73 –88. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2005.09.004. URL: http://www.sciencedirect.com/science/article/pii/S092188900500134X.

[156] Dung Phan, Junxing Yang, Matthew Clark, Radu Grosu, John D Schierman, Scott A Smolka, and Scott D Stoller. "A Component-Based Simplex Architecture for High-Assurance Cyber-Physical Systems." In: *arXiv preprint arXiv:1704.04759* (2017).

[157] Dung Phan, Junxing Yang, Radu Grosu, Scott A. Smolka, and Scott D. Stoller. "Collision avoidance for mobile robots with limited sensing and limited information about moving obstacles." In: *Formal Methods in System Design* 51.1 (2017), pp. 62–86. URL: https://doi.org/10.1007/s10703-016-0265-4.

[158] Benjamin C. Pierce and David N. Turner. "Proof, Language, and Interaction." In: ed. by Gordon Plotkin, Colin Stirling, and Mads Tofte. 2000. Chap. Pict: A Programming Language Based on the Pi-Calculus.

[159] Benjamin Pierce and Davide Sangiorgi. "Typing and Subtyping for Mobile Processes." In: *Mathematical Strustures In Computer Science*. 1996, pp. 376–385.

[160] Amir Pnueli. "The Temporal Logic of Programs." In: *Proceedings of FOCS*. 1977.

[161] Pony. *Pony Programming Langugage*. https://www.ponylang.org. 2017.

[162] Stephen Ponzio and Ray Strong. "Semisynchrony and real time." English. In: *Distributed Algorithms*. Ed. by Adrian Segall and Shmuel Zaks. Vol. 647. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 120–135. ISBN: 978-3-540-56188-0. DOI: 10.1007/3-540-56188-9_9. URL: http://dx.doi.org/10.1007/3-540-56188-9_9.

[163] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. "ROS: an open-source Robot Operating System." In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[164] Basant Rajan and RK Shyamasundar. "Multiclock Esterel: a reactive framework for asynchronous design." In: *IPDPS*. 2000.

[165] Robbert van Renesse and Fred B. Schneider. "Chain replication for supporting high throughput and availability." In: *Proc. 6th USENIX OSDI*. San Francisco, CA, Dec. 2004.

[166] James Riely and Matthew Hennessy. "A Typed Language for Distributed Mobile Processes (Extended Abstract)." In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. 1998.

[167] John Rushby. "Systematic formal verification for fault-tolerant time-triggered algorithms." In: *Software Engineering, IEEE Transactions on* (1999).

[168] Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George J Pappas, and Sanjit A Seshia. "Automated composition of motion primitives for multi-robot systems from safe LTL specifications." In: *IROS*. IEEE. 2014, pp. 1525–1532.

[169] Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George J Pappas, and Sanjit A Seshia. "Automated composition of motion primitives for multi-robot systems from safe LTL specifications." In: *International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2014, pp. 1525–1532.

[170] Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George J Pappas, and Sanjit A Seshia. "Implan: scalable incremental motion planning for multi-robot systems." In: *International Conference on Cyber-Physical Systems (ICCPS)*. IEEE. 2016, pp. 1–10.

[171] John D Schierman, Michael D DeVore, Nathan D Richards, Neha Gandhi, Jared K Cooper, Kenneth R Horneman, Scott Stoller, and Scott Smolka. *Runtime assurance framework development for highly adaptive flight control systems*. Tech. rep. Barron Associates, Inc. Charlottesville, 2015.

[172] Fred B. Schneider. "Implementing fault-tolerant services using the state machine approach: a tutorial." In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319.

[173] Koushik Sen. "Effective Random Testing of Concurrent Programs." In: *Proceedings of ASE, 2007*.

[174] Koushik Sen. "Race directed random testing of concurrent programs." In: *Proceedings of PLDI*. 2008, pp. 11–21.

[175] Koushik Sen and Gul Agha. "Automated Systematic Testing of Open Distributed Programs." In: *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*. 2006.

[176] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. "MultiSE: Multi-path symbolic execution using value summaries." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 842–853.

[177] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. "Programming and Proving with Distributed Protocols." In: *45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '18. ACM, 2018.

[178] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. "Towards Verified Artificial Intelligence." In: *ArXiv e-prints* (2016). arXiv: 1606.08514.

[179] Sanjit A. Seshia and Pramod Subramanyan. "UCLID5: Integrating Modeling, Verification, Synthesis, and Learning." In: *Proceedings of the 15th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. 2018.

[180] Danbing Seto, Enrique Ferriera, and Theodore Marz. *Case Study: Development of a Baseline Controller for Automatic Landing of an F-16 Aircraft Using Linear Matrix Inequalities (LMIs)*. Tech. rep. CMU/SEI-99-TR-020. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=13489.

[181] Lui Sha. "Using Simplicity to Control Complexity." In: *IEEE Softw.* 18.4 (July 2001), pp. 20–28. ISSN: 0740-7459. DOI: 10.1109/MS.2001.936213. URL: https://doi.org/10.1109/MS.2001.936213.

[182] Yasser Shoukry, Pierluigi Nuzzo, Ayca Balkan, Indranil Saha, Alberto L. Sangiovanni-Vincentelli, Sanjit A. Seshia, George J. Pappas, and Paulo Tabuada. "Linear temporal logic motion planning for teams of underactuated robots using satisfiability modulo convex programming." In: *56th IEEE Annual Conference on Decision and Control (CDC)*. 2017, pp. 1132–1137.

[183] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. "PENELOPE: Weaving Threads to Expose Atomicity Violations." In: *Proceedings of FSE*. 2010.

[184] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. "The Open Motion Planning Library." In: *IEEE Robotics & Automation Magazine* (2012).

[185] Alexander J. Summers and Peter Müller. "Actor Services." In: *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. 2016.

[186] Bharath Sundararaman, Ugo Buy, and Ajay D. Kshemkalyani. "Clock synchronization for wireless sensor networks: A Survey." In: *Ad Hoc Networks (Elsevier* (2005).

[187] Paul Thomson, Alastair F. Donaldson, and Adam Betts. "Concurrency Testing Using Schedule Bounding: An Empirical Study." In: ().

[188] Andrew Tinka, Thomas Watteyne, and Kris Pister. "A decentralized scheduling algorithm for time synchronized channel hopping." In: *Second International Conference on Ad-Hoc Networks (ADHOCNETS)*. Springer, 2010, pp. 201–216.

[189] Abhishek Udupa, Ankush Desai, and Sriram Rajamani. "Depth bounded explicit-state model checking." In: *Proceedings of SPIN*. 2011.

[190] Frits W Vaandrager and AL de Groot. "Analysis of a biphase mark protocol with Uppaal and PVS." In: *Formal Aspects of Computing* (2006).

[191] Viktor Vafeiadis and Matthew Parkinson. "A marriage of rely/guarantee and separation logic." In: *International Conference on Concurrency Theory*. Springer. 2007, pp. 256–271.

[192] Jur P Van Den Berg and Mark H Overmars. "Prioritized motion planning for multiple robots." In: *Intelligent Robots and Systems (IROS)*. IEEE. 2005, pp. 430–435.

[193] Robbert Van Renesse and Deniz Altinbuken. "Paxos Made Moderately Complex." In: *ACM Comput. Surv.* 47.3 (Feb. 2015).

[194] Prasanna Velagapudi, Katia Sycara, and Paul Scerri. "Decentralized prioritized planning in large multirobot teams." In: *International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2010, pp. 4603–4609.

[195] Willem Visser and Peter C. Mehlitz. "Model Checking Programs with Java PathFinder." In: *Proceedings of SPIN*. 2005.

[196] Glenn Wagner and Howie Choset. "M*: A complete multirobot path planning algorithm with performance bounds." In: *International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2011, pp. 3260–3267.

[197] Chao Wang, Mahmoud Said, and Aarti Gupta. "Coverage Guided Systematic Concurrency Testing." In: *Proceedings of ICSE 2011*.

[198] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. "Symbolic Predictive Analysis for Concurrent Programs." In: *Proceedings of FM 2009*.

[199] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. "Jitk: A Trustworthy In-kernel Interpreter Infrastructure." In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2014.

[200] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Tom Anderson. "Verdi: A Framework for Implementing and Formally Verifying Distributed Systems." In: *2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2015.

[201] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M Murray. "Receding horizon temporal logic planning." In: *IEEE Transactions on Automatic Control* 57.11 (2012), pp. 2817–2830.

[202] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard M Murray. "TuLiP: a software toolbox for receding horizon temporal logic planning." In: *International Conference on Hybrid Systems: Computation and Control (HSCC)*. 2011.

[203] Qiwen Xu, Willem-Paul de Roever, and Jifeng He. "The rely-guarantee method for verifying shared variable concurrent programs." In: *Formal Aspects of Computing* 9.2 (1997), pp. 149–174.

[204] Jianqiao Yang, Ankush Desai, and Koushik Sen. *Multi-Path Symbolic Execution for P Language*. https://github.com/thisiscam/MultiPathP. 2017.

[205] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. "MODIST: Transparent Model Checking of Unmodified Distributed Systems." In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2009.