SEPARATION OF DISTRIBUTED COORDINATION AND CONTROL FOR PROGRAMMING RELIABLE ROBOTICS

BY

RITWIKA GHOSH

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor Sayan Mitra, Chair
Professor Gul Agha
Professor Geir Dullerud
Assistant Professor Taylor Johnson, Vanderbilt University
Assistant Professor Sasa Misailovic

ABSTRACT

A robot's code needs to sense the environment, control the hardware, and communicate with other robots. Current programming languages do not provide the necessary hardware platform-independent abstractions, and therefore, developing robot applications require detailed knowledge of signal processing, control, path planning, network protocols, and various platform-specific details. Further, porting applications across hardware platforms becomes tedious.

With the aim of separating these hardware dependent and independent concerns, we have developed *Koord*: a domain specific language for distributed robotics. *Koord* abstracts platform-specific functions for sensing, communication, and low-level control. *Koord* makes the platform-independent control and coordination code portable and modularly verifiable. It raises the level of abstraction in programming by providing distributed shared memory for coordination and port interfaces for sensing and control. We have developed the formal executable semantics of *Koord* in the $\mathbb{K}$ framework. With this symbolic execution engine, we can identify proof obligations for gaining high assurance from *Koord* applications.

Koord is deployed on *CyPhyHouse*—a toolchain that aims to provide programming, debugging, and deployment benefits for distributed mobile robotic applications. The modular, platform-independent *middleware* of *CyPhyHouse* implements these functionalities using standard algorithms for path planning (RRT), control (MPC), mutual exclusion, etc. A high-fidelity, scalable, multi-threaded simulator for *Koord* applications is developed to simulate the same application code for dozens of heterogeneous agents. The same compiled code can also be deployed on heterogeneous mobile platforms.

This thesis outlines the design, implementation and formalization of the *Koord* language and the main components of *CyPhyHouse* that it is deployed on.

To my parents, for their endless patience, love and support.

ACKNOWLEDGMENTS

I knew I wanted to eventually pursue a Ph.D., even as a 13 year old starting to prepare for various college entrance exams in India. I didn't know then that it would be a such a long, arduous, challenging, but ultimately rewarding and enriching journey. First, I want to thank Professor Sayan Mitra for being my advisor. Sayan leads by example; his intellect, hard work, enthusiasm for research, and discipline will continue to inspire me to be better for the years to come. Aside from his professional guidance, he is truly considerate and kind, and a friend to every one of his students. I am not exaggerating when I say that had it not been for Sayan, I might not have persevered in this journey.

I am also grateful to my doctoral committee: Prof. Geir Dullerud, Prof. Sasa Misailovic, Prof. Gul Agha, and Prof. Taylor Johnson. The work presented in this benefited immensely from their wealth of knowledge, ideas, and suggestions. I am truly fortunate to have had the opportunity to interact and learn from these outstanding researchers.

I would like to express my gratitude to the National Science Foundation for funding my research. I am thankful to Viveka Kudaligama, Maggie Metzger Chappell, and Computer Science Graduate Advising for their diligence and continuous support during the entire Ph.D. process.

A big thank you to everyone who worked on the CyPhyHouse project, because their hard work and creativity made this thesis possible. Thank you Joao Porto, Peter Du, Amelia Gosse, and Chiao Hsieh and Minghao Jiang for suffering through the development pains, and working long hours in the lab with me. I am also very grateful to Chuchu Fan and Hussein Sibai, for being wonderful labmates and friends, and being there when I needed them.

Ph.D. is an isolating process, but I was lucky to have many friends who were there every step of the way. #Hashtag, my trivia team, thank you for always being a source of fun and happiness; thank you Carol Wisniewski, for being a mom away from home; and thank you David, for being a wonderful friend and support system, specially in the last few months of this process. I would be remiss if I didn't acknowledge Prof. Rajmanujam, whose lecture on the Incompleteness theorem made me venture into formal methods in the first place. He and Prof. Kamal Lodaya have my gratitude for their teaching and mentorship, that led me into this journey. Finally, I would like to thank my parents for everything. Their unconditional love is my fundamental axiom.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND SYMBOLS

BMC	Bounded Model Checker
DSM	Distributed Shared Memory
PO	Proof Obligation
V&V	Verification and Validation
DRA	Distributed Robotics Application
SMT	Satisfiablity Modulo Theory
DSL	Domain Specific Language
CPS	Cyber Physical Systems
RRT	Rapidly-exploring Random Tree
MPC	Model Predictive Control
PID	Proportional-Integral-Derivative (controller)
BNF	Backus-Naur Form
SATS	Small Aircraft Transport System
SLAM	Simultaneous Localization and Mapping
HVAC	Heating, Ventilation and Air-Conditioning
AST	Abstract Syntax Tree
I/O	Input/Output
UDP	User Datagram Protocol
RT factor	Real Time factor
GPS	Global Positioning System
$\mathbb{R}_{\geq 0}$	Set of non-negative real numbers
$[\![\varphi]\!]_c$	Evaluation of φ on a configuration c
$M[x \mapsto v]$	Replacing the value of x by v in the map M
$\mathbb{C}$	Set of all possible system configurations
$\mathbb{S}$	Set of all possible global memory mappings

$\mathbb{L}$	Set of all possible robot configurations
$\wp(X)$	Power set of X

CHAPTER 1: INTRODUCTION

Distributed robotic applications manufacturing [66, 31], agriculture [12, 69], transportation [32, 38], delivery [57], and mapping [75]. Following the trends in cloud, mobile, and machine learning applications, programmability is critical to unlocking this potential as robotics platforms become increasingly accessible, and hardware developers shift to the applications marketplace. Available domain-specific languages (DSL) for robotics are tightly coupled with platforms, and they combine low-level sensing, communication, and control tasks with the application-level logic. Programming languages like C#, Swift, Python, and development tools like LLVM [49] have helped make millions of people with diverse backgrounds, into mobile application developers. Open-source software libraries like Caffe [39], PyTorch [63], and Tensorflow [1] have propelled the surge in machine learning research and development. To a lesser degree, similar efforts are afoot in democratizing robotics. Most prominently, ROS [68] provides hardware abstractions, device drivers, messaging protocols, many common library functions and has become prevalent. Libraries such as PyRobot [59] and PythonRobotics provide hardware-independent implementations of common functions for physical manipulation and navigation of individual robots.

Nevertheless, it requires significant effort and time (of the order of weeks) to develop, simulate, and debug a new application for a single mobile robot—not including the effort to build the robot hardware. The required effort multiplies for distributed and heterogeneous systems, as none of the existing robotics libraries provide either

(i) support for distributed coordination or

(ii) easy portability of code across different platforms.

This tight-coupling and the attendant lack of abstraction hinders application development on all fronts—portability, code reuse, and verification and validation (V&V). In particular, formal reasoning about a collection of robots communicating, coordinating, and interacting with a physical environment is complexified by cyber-physical interactions. Correctness under concurrency and asynchrony are prominent research problems in distributed computing. Correctness under noise, disturbances, and imprecise platform (plant) models is studied intensively by roboticists and control theorists. These areas involve related but entirely different hardware level concepts of control and sensing and software level concepts of distributed protocols and program interactions. The analysis techniques from these communities are based on very different formal models and mathematics, and both would be necessary to provide satisfactory safety guarantees for distributed robotic applications. Our vision is, *not* to combine all of the above in an *all-encompassing formalism*, but to create a language that separates the concerns to divide and conquer using existing analyses from both communities.

1.1 MOTIVATION AND APPROACH

Our aim is to design abstractions and language constructs that separate the *platform-independent* decision and coordination from *platform-dependent* actions. For example, in an application for distributed package delivery with mobile robots, the code for waypoint assignment, load-balancing, and handling failures should abstract away implementations of waypoint tracking controllers and functions for navigation and communication. Such application code will be portable, with appropriate platform-specific implementations of these abstractions, e.g., steering controller for car, thrust controller for a quadcopter, and GPS or indoor localization subroutines.

Our goal towards building reliable distributed robotics applications (DRAs) requires us to be able to model them rigorously. The correctness or reliability of such applications is typically described in terms of specific properties such as collision avoidance, mutual exclusion, among others. These properties should be *invariants*, i.e., they should always hold true during any execution of the DRA.

This is the main motivation behind our desired separation of platform-independent decision and coordination from platform-dependent control. Formal models of distributed systems usually treat them as a collection of interacting individual agents or processes. Interactions between the agents or processes in a distributed system model can be easily captured using shared variables or resources. In Chapter 3, we present our model for distributed robotics systems, in which the execution of DRAs is captured as alternating discrete and continuous *transitions*. This model is similar to Hybrid I/O automata [54], which also decomposes hybrid system descriptions. Given such a formal model, we explore the following analysis techniques for verifying invariants.

Explicit State Bounded Model Checking The reachable states of the system are explicitly computed to check whether every state satisfies a given invariant. The number of reachable states of the system can grow exponentially in the number of variables. This state-space explosion is compounded even further in a multi-robot system, with continuous behaviors. We explore the explicit state bounded model checking technique as a proof of concept in Chapter 5, to provide a path towards symbolic bounded model checking for DRAs in the future.

Inductive Invariants Verification of invariants becomes more tractable when they are inductive; i.e., if a property is true in a state of the system, then it is true for any state reachable in a single transition from the system. In Chapter 5, we discuss how inductive invariants can be verified for our model. One of the issues that arise in verifying inductive invariants for distributed robotics systems is from continuous behavior during its interactions with the environment. In our model, a single transition is either discrete or continuous. Defining the notion of a state reachable in a single transition in this context becomes difficult as technically, there are infinitely many such states for the continuous component of the DRA. However, our separation of distributed coordination from control enables separate verification of said inductive invariants for its discrete and continuous components. We use data-driven verification techniques to compute an overapproximation of reachable states during a continuous transition from a state which satisfies the invariant.

Data-Driven Analysis Data-driven analysis [28] uses uses sampled execution traces of the continuous behaviors of the system from a given state or set of states to verify an invariant for states reachable during an execution with the same length (or duration). Such analysis can be performed to generalize these sampled traces to an overapproximation of reachable states, and the invariant can then be verified on this generalized set of reachable states. This can sometimes return inconclusive results if the overapproximation is too coarse. In Chapter 6 and 7, we show how we used data-driven analysis for verifying properties of DRAs through case studies.

Formal Semantics of Languages for DRAs Typical formal analyses of distributed systems and hybrid systems are performed on theoretical mathematical models. In our view, implementing DRAs with guarantees based on such models do not completely capture possible issues arising from their implementation in actual programming languages. Thus, guaranteeing correctness of DRAs implemented in a programming language requires verification of the formal model combined with verification of the language features themselves. This becomes difficult if the programming language in question lacks a formal semantics. Given a formal semantics for a language specifically designed with abstractions for programming DRAs, it becomes possible to perform a formal analysis of the distributed robotics system model directly using said semantics. We demonstrate such a semantics-based model in Chapter 3.

This thesis discusses the design, development, and implementation of *Koord*: a language and supporting verification and testing tools for programming distributed robotic systems.

Koord has been designed with the following three stakeholders in mind:

- *The application developer*, for whom our *Koord* language provides key abstractions (*sensor and actuator ports, distributed shared memory, and synchronous execution*) to develop robot applications that interact with the physical environment and other participating robot programs.

- *The V&V engineer*, for whom the $\mathbb{K}$-based formal executable semantics we have developed for *Koord* and our Z3-based prover, can help discharge key invariants of the *Koord* applications. This verification process also helps identify the *platform-dependent* proof obligations that have to be discharged or validated through simulation and testing.

- *The platform engineers* deploying the robot applications, for whom our abstraction makes the *Koord* programs portable across platforms. Our high-fidelity *Koord* simulator can be used in conjunction with other reachability analysis tools to test and validate the platform-dependent proof obligations.

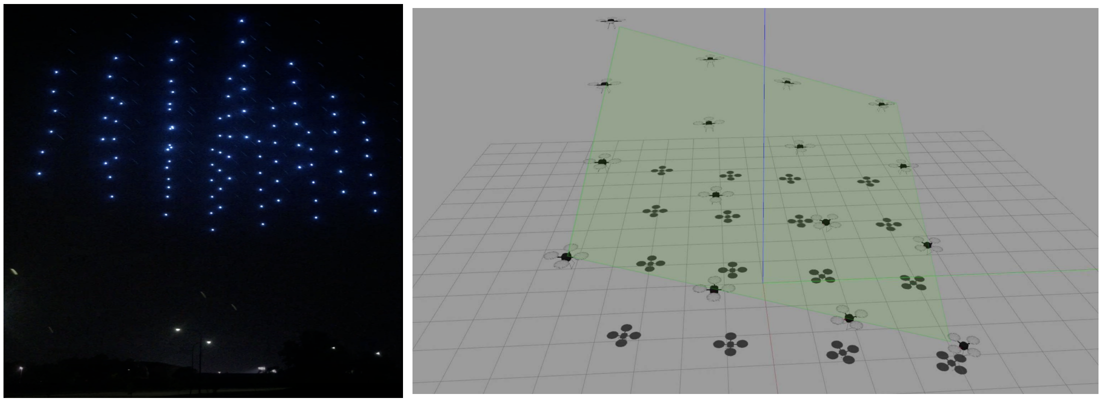

Figure 1.1: Swarm formation show by FireFly Inc. (*Left*). Simulation of shape formation (*Right*)
.

Additionally, intending to simplify application development for distributed and heterogeneous systems, we implemented *Koord* as a part of *CyPhyHouse*—an open-source software toolchain for programming, simulating, and deploying mobile robotic applications. Our *Koord* compiler, which is a part of *CyPhyHouse*, generates code that can be and has been directly deployed on aerial and ground vehicle platforms and simulated with the CyPhyHouse *simulator*. We have built the *CyPhyHouse middleware* with a modular structure to make it easy for roboticists to add support for new hardware. Figure 1.2 shows an overview of the *Koord* and *CyPhyHouse* tools and infrastructure that we have built.

In this work, we target distributed coordination applications such as collaborative mapping [20], surveillance, delivery, formation-flight, etc. with aerial drones and ground vehicles. Figure 1.1 shows a formation flight application in an aerial drone show, and a similar

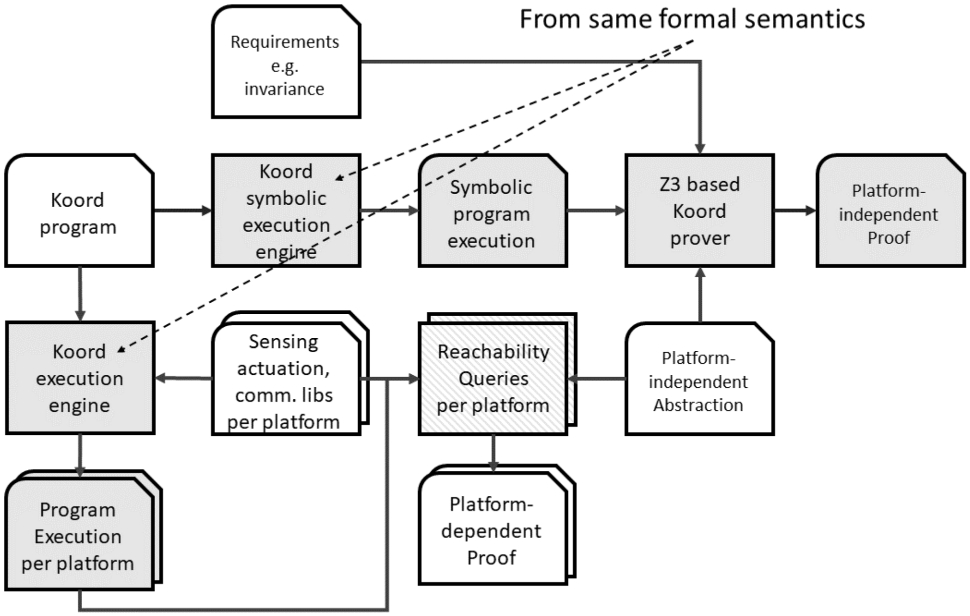
From same formal semantics
Requirements e.g. invariance
Koord program
Koord symbolic execution engine
Symbolic program execution
Z3 based Koord prover
Platform-independent Proof
Koord execution engine
Sensing actuation, comm. libs per platform
Reachability Queries per platform
Platform-independent Abstraction
Program Execution per platform
Platform-dependent Proof

Figure 1.2: Programmers develop distributed robotics applications in a high-level language and prove properties using symbolic executable semantics in $\mathbb{K}$. Platform-specific assumptions are abstracted and can be checked using simulations and hardware deployments.

formation application in the *CyPhyHouse* simulator for *Koord*. We believe that for such applications, low-level motion control for the individual robots is standard but tedious, and coordination across distributed (and possibly heterogeneous) robots is particularly difficult and error-prone.

1.2 RELATED WORK

Early domain specific languages for robotics were proprietary and tied to specific platforms. See [61] for a detailed survey. With the lowering hardware costs and increasing popularity, there is a growing interest in open and portable frameworks and languages [65, 15, 82, 79]. Our point of view on automating robot programming is different in that we expect that the programmer's creativity and efforts will be necessary well beyond writing high-level specs in solving distributed robotics problems; consequently only the tedious and standard steps in coordination and control are automated using the *Koord* compiler.

Languages for Distributed Shared Memory Systems Programming systems using the shared memory paradigm have been developed for several distributed computing systems [60, 2, 17, 48, 21]. Specifically, P [22] and PSync [24] are DSLs for asynchronous partially distributed systems, but cyber-physical interactions are not supported.

DSM has also been proposed as a programming model in the context of wireless networks [9, 34]. These programming models are defined mathematically in terms of state machines or in terms of APIs, and are typically not embodied in a programming language with carefully designed syntax and semantics to enforce the models.

Uncertainty and Robotics Abstractions λ_O is a probabilistic programming language in which sampling methods are used to specify probability distributions, while expressing and reasoning about these methods formally. It finds application in robot localization and mapping. In the same vein, *Uncertain*$\langle T \rangle$ provides a programming language abstraction for uncertain data. It is a departure from previous probabilistic programming languages in the wide range of developers it serves, as opposed to being accessible only by experts. The language provides abstractions and semantics for uncertain data, like sensed information about location, temperature, etc. *Koord* does not currently perform reasoning involving uncertainty in sensor readings or robot localization currently, and these are concerns that cannot currently be explored by exploiting the extensibility of the *Koord* semantics implemented in $\mathbb{K}$, as $\mathbb{K}$ doesn't support probabilistic reasoning yet. While these languages provide semantics for uncertainty in robot abstractions and sensing issues, they do not provide distributed application design capabilities.

Robotics Application Development Frameworks Robot Operating System (ROS) [68] is the predominant member in this category. At its core, ROS supports a publish-subscribe-based communication, and the ROS community has built drivers for numerous hardware components.

Framework /system	**Dist. Sys.**	**Heterogeneous**	**Sim**	**Prog. Lang.**	**Compiler**	**V&V**
ROSBuzz [73]	✓	✓	✓	Buzz	✓	
PythonRobotics		✓	✓	Python		
PyRobot [59]		✓	✓	Python		
MRPT [11]		✓		C++		
Robotarium [64]		✓	✓	Matlab		
DRONA [23]	✓		✓	P [22]	✓	✓
Live [18]		✓		LPR	✓	
Koord	✓	✓	✓	*Koord*	✓	✓

Table 1.1: Comparison of CyPhyHouse and existing robotics frameworks.

Our implementation of the *Koord* abstractions for the quadcopter and vehicle platforms use ROS just like thousands of other robotics products and projects.

The Voltron programming system has been to explore the concept of team-level programming in active sensing applications. Voltron offers programming constructs to create the illusion of a simple sequential execution model while still maximizing opportunities to dynamically re-task the drones as needed. However, the framework itself relies on testing and system heuristics to provide approximate guarantees, and doesn't provide any soundness or completeness assurances.

The Live language [18] allows the program to be changed while running, hence reducing the feedback loop across writing, compiling, and testing of robot programs. It does not support for distributed applications.

The table above gives a summary of robotics languages that have been deployed on hardware. ROSBuzz [73] supports the Buzz language, which doesn't provide abstractions like *Koord* for path planning and shared variables. The Live Robot Programming language provides abstractions in terms of nested state machines and allows the program to be changed while running. It does not support robot ensembles. Programming systems using the shared memory paradigm have been developed for several distributed computing systems [60, 2, 17, 48, 21].

Programming Robotics with Support for Formal Analysis P [22] and PSync [24] are DSLs for asynchronous partially distributed systems, but cyber-physical interactions are not supported. P has been integrated into the DRONA framework [23] and the latter has very similar objectives to our work, but the approaches and solutions are different. DRONA is a framework for multi-robot motion planning and so far deployed only on drones. Koord and the underlying middleware aims to be more general, and multiple applications have been deployed on cars and drones in both simulations and hardware. The explicit model checker (using Zing) of DRONA relies on manual proofs of their safe-plan-generator and path-executor, which are analogous to Koord function summaries and controller assumptions. DRONA's model checker explores reachable states upto a given depth (number of transitions from an initial state). Koord proves inductive invariants using our own symbolic executable semantics. Therefore, when all proof obligations are discharged for a candidate invariant, the Koord system proves the invariant holds for all reachable states. Further, while our Task application implements something similar to the distributed plan generator which is a built-in feature for DRONA, *Koord*'s port interfaces allow portability across arbitrary planners.

VeriPhy [15] also has some commonality with *CyPhyHouse*; however, instead of a programming language, the starting point is differential dynamic logic [14].

"Correct-by-construction" synthesis from high-level temporal logic specifications has been applied to mobile robotic systems (see, for example [45, 43, 80, 81, 77]).

1.3 THESIS OUTLINE

During the course of this Ph.D. research, we identified the following three major contributions.

- Identifying programming abstractions for DRAs: We have several baked-in abstractions in the design of *Koord* to simplify the distributed robotics programming. Our design of *Koord* creates a separation of platform-specific from platform-independent concerns. In Chapter 2, we present an overview of the *Koord* design, development, semantics, formal analyses, and its implementation in *CyPhyHouse*. We dive into the details of the *Koord* semantics in Chapter 3, and discuss how we implemented these semantics in the $\mathbb{K}$ semantic framework in Chapter 4.

- Developing verification approaches for distributed robotics applications: The formal semantics of *Koord* enables our formal analysis to benefit from the separation of distributed coordination and control. Chapter 5 includes our approach to formal verification of several benchmarks using explicit state bounded model checking and our decomposed verification of inductive invariants. Chapters 6, 7 and 8 present case studies on three benchmark applications using our decomposed verification approach.

- Implementing a simulation and deployment toolchain for *Koord*: There are several engineering challenges that arise while implementing abstractions for a distributed robotics system. We present our implementation of *Koord* in the *CyPhyHouse* toolchain in Chapter 9.

Reading this thesis Chapter 2 essentially presents a high level overview of the thesis. Chapter 3 defines the *Koord* language semantics, and is-self contained. Chapter 4 and 5 are parallel, and we recommended that they be read after Chapter 3. Chapter 6, 7 and 8 are independent of each other. These chapters present case studies based on the theory discussed in Chapter 5, so we would suggest reading Chapter 5 before them. While Chapter 9 can be read by itself, it heavily references the application discussed in Chapter 7. Finally, Chapter 10 includes our concluding remarks and presents directions for future work.

CHAPTER 2: OVERVIEW OF THE KOORD LANGUAGE FRAMEWORK

In this thesis, through our design and implementation of the programming language *Koord*, we explore a distributed, decentralized approach for coordination and control in multi-robot systems through an executable semantics and a simulation and deployment framework. We will also show how our design of the executable semantics of *Koord* aids in the formal verification of invariants for *Koord* application programs.

In this chapter, we will first discuss the key features of the *Koord* programming system. One of the most studied forms of coordination among multiple robots in a system is movement done in a formation-preserving manner. We first provide an overview of *Koord* with an example application for formation control in Section 2.1. This application causes a collection of quadrotors to form a pattern of the kind seen in aerial drone shows (Figure 1.1). In Sections 2.3 and 2.4, we highlight how *Koord*'s design enables verification techniques which decouple distributed coordination from low-level control. Later in Section 2.6, we introduce our implementation of *Koord* in the *CyPhyHouse* toolchain [7].

2.1 THE KOORD LANGUAGE

We first give a quick overview of the *Koord* with an example and introduce the semantic notions that we present in more detail in Chapter 3.

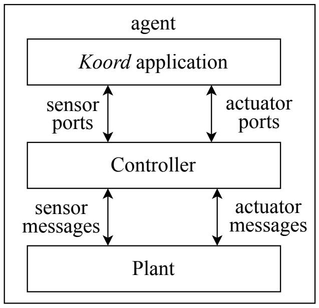

Figure 2.1: Koord system architecture.

A *Koord* program, LineForm, for a set of robots to form an equispaced line is shown in Figure 2.2. Each robot program has access to three constants (a) a unique integer identifier `pid` for itself, (b) a list `ID` of identifiers of all participating robots and (c) the number `N_sys` of participating robots.

Distributed Shared Variables for Platform-Independent Coordination *Koord* provides *shared* variables that participating robots can use to communicate and coordinate. At

```
1   using Motion:
2     sensors: Point psn
3     actuators: Point target
4
5   allread: Point x
6
7   TargetUpdate:
8     pre True
9     eff : if not(pid == N_sys − 1 or pid == 0):
10      Motion.target = mid([x[pid+1],x[pid−1]])
11      x[pid] = Motion.psn
```

$x_{t+1} = Ax_t$, where

x_0: initial position vector,
x_t: position at time t
A: transition matrix, e.g.,

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.2: LineForm program in *Koord* (*Left*) and its mathematical counterpart as it would appear in a typical robotics and control textbook (*Right*).

Line 5 in LineForm, the variable `x`, declared with the **allread** keyword, is a shared array from which all robots can read, but each robot `pid` can only write to `x[pid]`. The shared array `x` makes it possible for a robot to read the current position of other robots in a single line of code. LineForm uses:

(i) the unique integer identifier `pid` for itself and

(ii) the number `N_sys` of all participating robots.

For multi-robot programs that write to shared variables, *Koord* provides concurrency control with mutual exclusion and **atomic** blocks. The semantics of *Koord distributed shared memory (DSM)* is discussed in Chapter 3.

Port Abstractions for Platform-Dependent Control For abstract functions such as reading the current position, sensing of obstacles, and movement between points in space, different robot platforms need different implementations. *Koord* abstracts the details of those implementations and allows the robot program to interact with its environment through *sensor* and *actuator* ports. For example, LineForm uses a *module* (library) called `Motion`, which provides a sensor port called `psn` that publishes the robot's position (with some periodicity and accuracy), and an actuator port called `target` for specifying a target position that the controller should try to drive to. From the programmer's perspective, ports work like variables: the program can read from the sensor ports and write data to actuator ports to specify the control objectives. Implementations of Motion would use different strategies for different platforms. In our experiments, the Motion module for a quadcopter uses an indoor-camera-based positioning system to update the psn port and uses an RRT-based [50] path planner and PID controller. On the other hand, for a small racing vehicle platform, the implementation uses a model-predictive controller [47, 35].

Events *Koord* uses an event-driven style of programming, in which events are written in a usual precondition-effect style to define how program variables are updated. LineForm uses a single TargetUpdate event, which sets the target of each robot (except the extremal robots) to be the center of the position of its neighbors. This event has a precondition which always evaluates to *True*. As we shall see in Chapter 3, *Koord* semantics ensures a synchronous round-by-round execution of events for all robots. That is, for a given execution parameter $\delta > 0$, one event per robot can occur every δ time. Notice that the mathematical representation of the linear system shown in Figure 2.2 (*Right*) is similar to the synchronous update encoded on Line 10 of the *Koord* implementation of LineForm on Figure 2.2 (*Left*). This type of a synchronous update rule is a typical example of a large family of textbook algorithms for distributed consensus, rendezvous, optimization, flocking, and pattern formation [76, 13, 55].

2.2 KOORD SEMANTICS

In a *distributed robotics application* (DRA), multiple instances of the same program are executed by all participants to solve a problem. The execution semantics of such a DRA is complicated by issues of asynchrony and concurrency, as well as by the interactions between software and the physical environment. In our design of the *Koord* semantics, we made a few simplifying assumptions:

- The execution of a *Koord* program advances in a synchronous, *round-by-round* fashion. Each round lasts for some $\delta > 0$ time, where δ is a parameter of the semantics.
- During a δ-duration round, the robots compute, move, and communicate with each other through distributed shared memory.

We have developed the full executable semantics of *Koord* in the $\mathbb{K}$ semantics framework [70]. $\mathbb{K}$ is a rewriting-based executable framework for defining language semantics. A syntax and an executable semantics of a language implemented in $\mathbb{K}$ behave as a parser and an interpreter for the language. $\mathbb{K}$ also supports exhaustive nondeterministic exploration of program behaviors, at no additional development cost. A consequence of our use of $\mathbb{K}$ to implement the executable language semantics is that the *Koord* language is consistently extensible, to support a broader set of applications.

In Chapter 3, we discuss the formal semantics of the *Koord* language and our implementation of this formal semantics in $\mathbb{K}$ in Chapter 4.

2.3 SEMANTICS-DRIVEN DECOMPOSED VERIFICATION: KOORD PROVER

We have implemented the *Koord* Prover tool on top of the *Koord* semantics in order to perform symbolic checking of inductive invariants for *Koord* programs. Through case studies in Chapters 5, 6, and 7, we show how we used the $\mathbb{K}$ executable semantics of *Koord* with DryVR [28, 27] to check inductive invariants for *Koord* application.

Consider a *geofencing* requirement [74], a natural requirement for LineForm: given a (hyper)rectangle $\mathit{rect}(\mathsf{a},\mathsf{b})$ defined by two corners a and b, if all robots are initialized within $\mathit{rect}(\mathsf{a},\mathsf{b})$, then we would like them to stay in $\mathit{rect}(\mathsf{a},\mathsf{b})$ at all times. This requirement can be stated as an invariant of the system:

Invariant 2.1.

$$\bigwedge_{i\in ID}\Big(\ \mathtt{Motion.psn}_i \in \mathit{rect}(\mathsf{a},\mathsf{b}) \land x_i \in \mathit{rect}(\mathsf{a},\mathsf{b})\ \Big) \tag{2.1}$$

To check Invariant 2.1, it is necessary to reason about both platform-dependent and independent parts of the application. Using the *Koord* Prover, one can reason about the application in a decomposed fashion as follows.

(1) Assuming that all shared positions x[i] are in $\mathit{rect}(\mathsf{a},\mathsf{b})$, we have to show that the targets computed by LineForm are in $\mathit{rect}(\mathsf{a},\mathsf{b})$. This platform-independent proof obligation is about the correctness of the program logic of LineForm (particularly line 10, which updates Motion.target). To validate this proof obligation, one has to compute the states of the system that are *reachable* after the TargetUpdate event executes, and check that Invariant 2.1 holds on those states.

    The *Koord* Prover uses symbolic execution to compute these states and encodes the check as an SMT problem. For Invariant 2.1, and many other applications and invariants, this proof obligation is discharged fully automatically as it involves only linear arithmetic.

(2) Assuming that the sensed current position $\mathtt{Motion.psn}_i$ and the computed target are in $\mathit{rect}(\mathsf{a},\mathsf{b})$, we have to show that a given robot's controller indeed keeps it in $\mathit{rect}(\mathsf{a},\mathsf{b})$. This platform-dependent proof obligation is about the correctness of the controller implemented

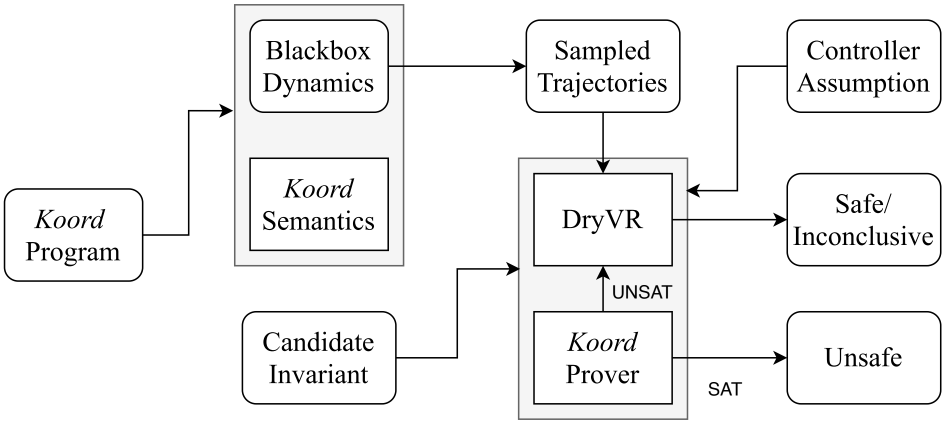

Figure 2.3: Workflow for invariant verification using *Koord* Prover.

in the `Motion` controller. *Koord* helps identify such obligations or assumptions about controller implementations. If we have a model *traj* for computing the trajectory of a vehicle's motion, we can state and prove this proof obligation as follows:

Assumption 2.1.

$$\begin{aligned}\forall t \in [0,\delta], &\, \mathit{traj}(\texttt{Motion.psn}, \texttt{Motion.target}, t)\\ &\subseteq \mathit{rect}(\texttt{Motion.psn}, \texttt{Motion.target}),\end{aligned} \tag{2.2}$$

where *traj* gives the position of the robot at time t as a function of the target and its initial position at the beginning of the round. To check such assumptions, we can use a reachability analysis tool for dynamical and hybrid systems. Many such tools exist [30, 19, 26, 8, 28]. In our experiments, we used the simulation-driven reachability tool DryVR [28], to verify these assumptions. DryVR is scalable and does not require complete dynamical models of the controller. It computes overapproximations of reachable states from sampled system trajectories. When such overapproximations are completely contained within the set of states allowed by the assumption (or the invariant), the output remains *safe*. If not, then the output becomes *inconclusive* because of two possibilities: either the assumption may be invalid, or the computed overapproximation is too coarse.

Figure 2.3 shows the overview of the workflow using the *Koord* Prover to verify inductive invariants. Given a *Koord* program and a candidate invariant, the *Koord* semantics generates a symbolic execution. The *Koord* Prover then uses this symbolic execution and controller assumptions to check the validity of the candidate invariant. To verify the controller assumptions, sampled trajectories obtained from the blackbox dynamics of the robots are analyzed using DryVR. For valid invariants with verified controller assumptions, the *Koord* program is pronounced ‘safe’ (with respect to the candidate invariant). When DryVR is unable to verify the controller assumption, the result is ‘inconclusive’.

2.4 ENGINEERING RELIABLE DRAS WITH THE KOORD SYSTEM

The *Koord* Prover supports the engineering of reliable systems by helping discover and validate platform-dependent proof obligations.

In general, if the assumptions needed for proving the correctness of an application are too strong, a DRA engineer could either revise the assumptions or modify the invariant requirement so that weaker assumptions may be sufficient. Using the high-fidelity *Koord* simulator, which is part of the *CyPhyHouse* toolchain, we can gain insights about when such assumptions are violated.

For instance, we see in Chapter 6 that sensitivity analysis using DryVR is able to detect violations of Assumption 2.1. A quadcopter model with poor PID control could temporarily go out of bounds because of inertia while moving towards the target. After we configured the same quadcopter model with a different PID controller, DryVR was able to verify Assumption 2.1. Similarly, we see in Chapter 7 that DryVR is able to detect a ground vehicle’s possible inability to follow a path computed by a *path planner* as closely as required to maintain safe distances between vehicles. As we shall see in those case studies, the aforementioned assumptions require us to reason only about the platform-dependent control ports, allowing us to decouple the verification of these assumptions from the distributed program logic.

2.5 SEMANTICS-DRIVEN BOUNDED MODEL CHECKING

Koord’s executable semantics enables explicit and exhaustive exploration of the nondeterministic behaviors of *Koord* applications. Using that capability, we implemented a tool, the *Koord* Bounded Model Checker, or *Koord* BMC, for checking bounded invariants for Koord applications that use explicit state reachability analysis. The inputs to the *Koord* BMC are

(i) P, the *Koord* application program;

(ii) *inv*, a candidate invariant predicate;

(iii) *ID*, the set of robots executing P;

(iv) δ, the duration of a round; and

(v) n, the number of rounds of execution.

The tool uses the *Koord* executable semantics to compute all reachable configurations that can be reached during program transitions. As mentioned earlier, the *Koord* semantics allows the program to use a blackbox model of the environment. The blackbox controllers generate traces of the sensor and actuator ports of the system during the environment transitions of duration δ; those traces can be analyzed using a reachability analysis or model checking tool for dynamical and hybrid systems. We used DryVR for analysis of the sensor and actuator traces of *Koord* programs. The tool outputs *safe* if the invariant holds for n rounds of execution and *unsafe*, if it finds a counterexample execution of length n. Figure 2.4 shows an overview of the architecture of the *Koord* BMC.

To complete the verification, we then perform DryVR analysis of the blackbox traces of *safe* outputs to verify safety during environment transitions.

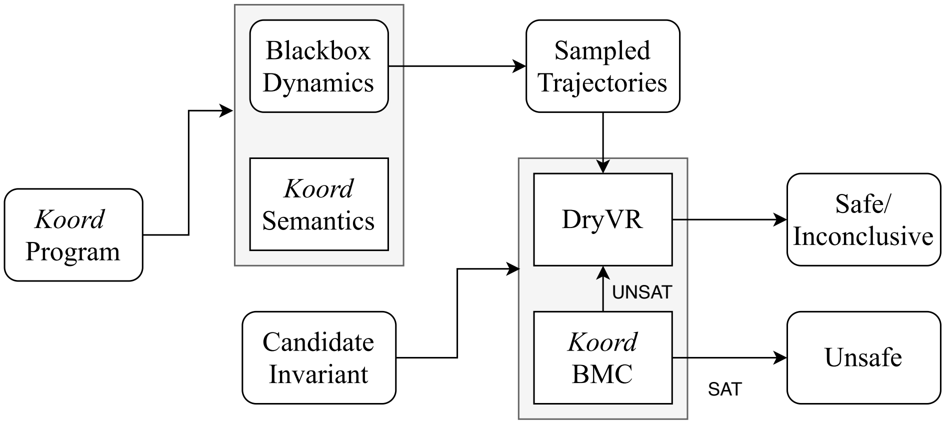

Figure 2.4: Workflow for bounded model checking using the *Koord* BMC.

While we require the programmer to provide a candidate inductive invariant for the *Koord* Prover, *Koord* BMC makes the verification procedure accessible to programmers who have no expertise in formal methods. Bounded model checking is hampered by the state-space explosion problem, as expected. However, we have several heuristics for pruning the search tree when the program has certain properties. For instance, if the events do not write to any shared variables, it is enough for the *Koord* BMC to explore only one order of execution as opposed to the worst case of `N_sys` orders, where `N_sys` is the number of robots in the system. Domain knowledge and observation of symmetries in the program can also lead to improvements in performance.

The importance of using programming language semantics in the context of distributed robotics applications is that it allows us is to verify the implementation of the applications in a programming language, as opposed to verifying them based on an abstract description of an automaton. Traditional theorem provers and model checking tools require the programmer to encode every automaton through its states and transitions. In contrast, our tool makes such formal analysis tools available to the programmer without requiring expertise in formal methods. We show in Chapter 5 how we performed verification through bounded model checking of applications through our $\mathbb{K}$ executable semantics. To our knowledge, our framework of *Koord* executable semantics and our associated formal analysis tools *Koord* BMC and *Koord* Prover is the only distributed multi-robot system analysis framework with a language semantics component to it, thus reducing the gap between theory and practice.

2.6 IMPLEMENTING KOORD

Recall that we made some simplifying assumptions in Section 2.2 about the *Koord* system and that they sidestepped issues of asynchrony and failures. While those assumptions make our executable semantics tractable, we also implemented an actual runtime system for *Koord* to confirm that the assumptions can be met by the platforms on which *Koord* is deployed. The *CyPhyHouse* toolchain [7] is an open-source implementation of *Koord*, presented in [7], which includes programming tools for simulation and hardware deployment.

As part of *CyPhyHouse*, we developed a software-hardware interface (*middleware*) in Python 3.5 to support the three-plane architecture comprising the *Koord* runtime system as shown in Figure 2.1.

The *Koord* compiler included with *CyPhyHouse* generates Python code for the application using all the supported libraries, such as the implementation of distributed shared variables using message passing over WiFi, motion libraries of the robots, high-level collision and obstacle avoidance strategies, etc. The application then runs with the Python *middleware* for *CyPhyHouse*.

At a high level, updates to a shared variable by one robot are propagated by the *CyPhyHouse* middleware and become visible to other robots in the next round. *CyPhyHouse* implements the shared memory between robots through UDP messaging over WiFi. Any shared memory update translates to an update message which the robot broadcasts over WiFi.

CyPhyHouse uses ROS to handle the low-level interfaces with hardware. To communicate between the high-level programs and low-level controllers, we use rospy, a Python client library for ROS, which enables the (Python) middleware to interface with ROS topics and services used for deployment or simulation. Each hardware platform and its controller requires its own interface in the *CyPhyHouse* middleware. We present the details of the *CyPhyHouse* middleware in Chapter 9. Figure 2.5 shows a high-level overview of the *CyPhyHouse* toolchain. The *CyPhyHouse* simulator for *Koord* is *high-fidelity, scalable, and flexible.* It executes

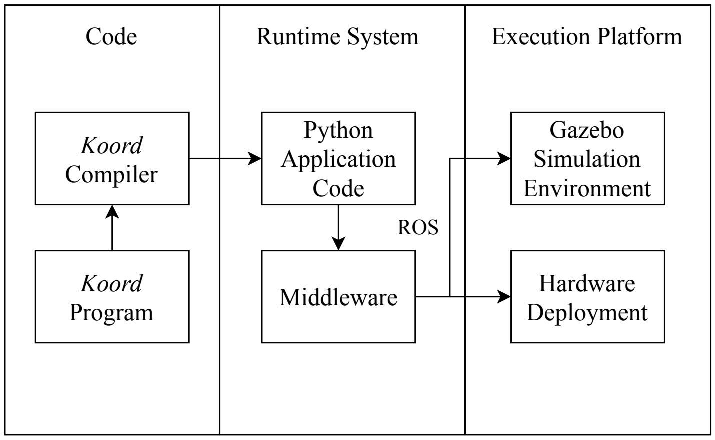

Figure 2.5: Overview of the CyPhyHouse Toolchain.

multiple instances of the application code, one for each robot in the scenario. Within the simulator, individual robots communicate with each other over a wired or a wireless network and with their own simulated sensors and actuators through ROS topics. For example, a simulation with 16 drones can spawn over 1.4K ROS topics and 1.6K threads, yet our simulator is engineered to execute and visualize such scenarios in Gazebo, running on standard workstations and laptops. Figure 2.6 shows a screenshot of such a simulation in Gazebo.

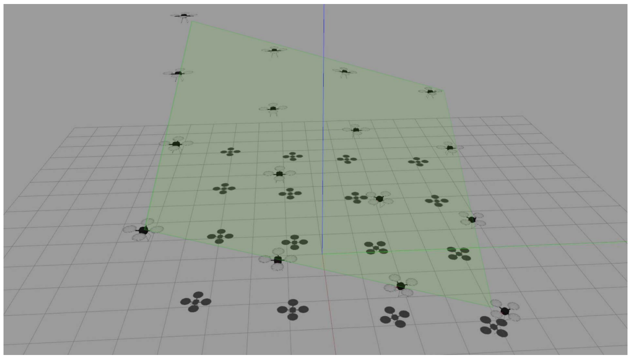

Figure 2.6: *CyPhyHouse* simulator running different scenarios with the same *Koord* application.

Koord applications have also been deployed on heterogeneous multi-robot systems of quadcopters and small racing vehicles using the *CyPhyHouse* toolchain.

2.7 SUMMARY

In this chapter, we presented an overview of our design of the *Koord* language with an illustrative example. We highlighted our semantics-driven approach to decomposed verification of inductive invariants, and explicit state bounded model checking and outlined our implementation of *Koord* in the *CyPhyHouse* toolchain. In the next chapter, we elaborate upon the semantics of *Koord*, which we only touched upon so far.

CHAPTER 3: FORMAL SEMANTICS OF KOORD

Programming language semantics are used to precisely define the behavior of programs one can write in the language. Language semantics gives every syntactic construct of the language a mathematical meaning. While individual robots run instances of a *Koord* program, the language design also determines the multi-robot system's behavior. *Koord* language semantics, therefore, should also precisely define the behavior of the system.

In designing such a distributed robotics system, one feature that requires particular attention is how and when each robot executes its events. Additionally, the shared memory design should capture the impact of the order in which robots execute their events. A *Koord* program execution consists of program transitions where the robots perform the computational steps required for distributed coordination; and environment transitions where the robots behave according to the control parameters set during the aforementioned computational steps. Since another facet of the semantics design is the interaction of the robots with the environment through port abstractions, we need to specify how these interactions interleave with computational steps performed by the robot while executing the program. Specifying such interactions becomes even more challenging when viewing the system of robots as a whole.

In this chapter, we present the syntax and discuss how our design of *Koord* semantics addresses the above challenges. We present some notable features of the *Koord* semantics as rewrite rules, and in Chapter 4, we discuss how we implemented these semantics in $\mathbb{K}$.

3.1 KOORD LANGUAGE SYNTAX

When a *Koord* application runs in a distributed multi-robot system, each robot executes an instance of the *Koord* application program. Figure 3.2 shows a *Koord* program Follow, where the participating agents move through a shared set of waypoints, where each point is visited exactly once by a participating robot.

Figure 3.1 shows the core grammar of *Koord* syntax in BNF. In the syntax presented in Figure 3.1, given an nonterminal NT, $NT^?$ means that it is optional in the syntax at that position, NT^* refers to zero or more occurrences, and NT^+ refers to one or more occurrences. The expression ($E1$ | $E2$) denotes that one can use either $E1$ or $E2$. Each syntactic production is presented as a rule in the format $NT ::= A$. This rule indicates that the nonterminal NT expands to A, where A can be nonterminal or terminal. We indicate *Koord* keywords and data types in bold.

$$
\begin{array}{lll}
\textit{Program} & ::= & \textit{Defs Module}^{?}\ \textit{DeclBlock Init}^{?}\ \textit{Event}^{+} \\
 & & \\
\textit{Defs} & ::= & [\textit{FuncDef}^{+}]^{?}\ [\textit{AdtDef}^{+}]^{?} \\
\textit{FuncDef} & ::= & \textbf{def fun}\ \textit{identifier}(\textit{Param}^{+}):\ \textit{Stmt}^{+} \\
\textit{AdtDef} & ::= & \textbf{def adt}\ \textit{identifier}:\ \textit{Decl}^{+} \\
\textit{Param} & ::= & \textit{Type identifier} \\
 & & \\
\textit{Module} & ::= & \textbf{using module}\ \textit{identifier}:\textit{SPorts APorts} \\
\textit{SPorts} & ::= & \textbf{sensors}:\textit{Decl}^{+} \\
\textit{APorts} & ::= & \textbf{actuators}:\textit{Decl}^{+} \\
 & & \\
\textit{Decl} & ::= & \textit{Type identifier} \mid \textit{Type identifier} = \textit{Val} \\
\textit{Type} & ::= & \textbf{int} \mid \textbf{float} \mid \textbf{bool} \mid \textbf{Point} \mid \textit{adt} \\
 & & \mid \textit{Type}[\textit{Int}] \mid \textbf{List}\,\langle\textit{Type}\rangle \mid \textbf{Queue}\,\langle\textit{Type}\rangle \\
 & & \\
\textit{DeclBlock} & ::= & \textit{AWDecls ARDecls LocalDecls} \\
\textit{AWDecls} & ::= & \textbf{allwrite}:\ \textit{Decl}^{+} \\
\textit{ARDecls} & ::= & \textbf{allread}:\ \textit{Decl}^{+} \\
\textit{LocalDecls} & ::= & \textbf{local}:\ \textit{Decl}^{+} \\
 & & \\
\textit{Init} & ::= & \textbf{init}:\ \textit{Stmt}^{+} \\
\textit{Event} & ::= & \textit{identifier}:\textbf{pre}\ (\textit{Cond})\ \textbf{eff}:\ \textit{Stmt}^{+}
\end{array}
$$

$$
\begin{array}{lll}
\textit{Expr} & ::= & \textit{AExpr} \mid \textit{BExpr} \\
\textit{AExpr} & ::= & \textit{AExpr AOp AExpr} \\
 & & \mid \textit{Expr}\texttt{++} \mid \texttt{-}\textit{AExpr} \mid \textit{Var} \mid \textit{AVal} \\
\textit{AOp} & ::= & + \mid - \mid * \mid / \\
\textit{BExpr} & ::= & \textit{Expr RelOp Expr} \mid \textit{Expr COp Expr} \\
 & & \mid \textbf{not}\textit{Expr} \mid \textit{Var} \mid \textit{BVal} \\
\textit{RelOp} & ::= & \geq \mid \leq \mid \geq \mid == \mid > \mid < \mid \neq \\
\textit{COp} & ::= & \textbf{and} \mid \textbf{or} \\
 & & \\
\textit{Stmt} & ::= & \textit{Assign} \mid \textit{FnCall} \mid \textit{Atomic} \\
 & & \mid \textit{Ite} \mid \textit{Loop} \mid \textit{Return} \\
\textit{Assign} & ::= & \textit{Var} = \textit{Expr} \\
\textit{Ite} & ::= & \textbf{if}\ \textit{BExpr Stmt}^{+}\ [\textbf{else}\ \textit{Stmt}^{+}]? \\
\textit{FnCall} & ::= & \textit{identifier}(\textit{Expr}^{+}) \\
\textit{Atomic} & ::= & \textbf{atomic}:\textit{Stmt}^{+} \\
\textit{Loop} & ::= & \textbf{for}\ \textit{identifier}\ \textbf{in}\ \textit{AExpr}:\textit{Stmt}^{+} \\
\textit{Return} & ::= & \textbf{return}\ \textit{Expr} \mid \textbf{return} \\
 & & \\
\textit{Var} & ::= & \textit{identifier} \mid \textit{identifier}[\textit{Expr}] \\
 & & \mid \textit{identifier}\textbf{.}\textit{identifier} \\
\textit{Val} & ::= & \textit{AVal} \mid \textit{BVal} \\
\textit{AVal} & ::= & \textit{Int} \mid \textit{Float} \\
\textit{BVal} & ::= & \textit{Bool}
\end{array}
$$

Figure 3.1: Core *Koord* program syntax.

A *Koord* program essentially consists of the following:

(i) Declarations of *ports*, which act as interfaces between the program and the sensor/actuator modules.

(ii) Declarations of shared and local program variables.

(iii) Events, which define how variables and sensor and actuator ports are written to and read from during the execution of this program.

Robot programs (rule *Program*) first can import sensor/actuator modules. In a *Koord* program, the controller (rule *Module*) specifies the ports: it contains all input and output ports for actuators (*APorts*) and sensors (*SPorts*) that the program uses. In Follow, the *Motion* module includes the declaration of the `target` actuator, which can be used to set a target waypoint for the robot. It also includes the sensors `psn` and `done`, which indicate the robot's current position, and whether it has reached its set target, respectively. One can also write *Koord* programs that have no interaction with the environment, in which case no such module needs to be specified.

Koord has three types of variables for reading/writing values.

(i) *Sensor and actuator ports* are used to read from sensor ports and write to actuator ports of controllers.

(ii) *Local program variables* record the state of the program.

(iii) *Distributed shared variables* are used for coordination across robots. All participating robots can read all shared variables. Any participating robot can write to an **allwrite** variable; while an **allread** variable, which is an array of size the same as the number of robots, can only have a single writer. Given an allread variable x, each robot with `pid` i can read any index in x, but can only write to $x[i]$.

```
1  using Motion
2     actuators:
3        Point target
4     sensors:
5        Point psn
6        bool done
7
8  allwrite:
9     List⟨Point⟩ dests
10
11 local:
12    bool pick = True
13    Point currentDest
14
15 init:
16   dests = [(200,10,0);(100,100,0)]
17
18 PickDest:
19   pre pick
20   eff: atomic:
21          if !isEmpty(dests):
22            currentDest = head(dests)
23            remove(dests,currentDest)
24            Motion.target = currentDest
25            pick = False
26
27 Remove:
28   pre !pick
29   eff: if Motion.status == done:
30        pick =  True
```

Figure 3.2: *Koord* code for Follow.

For instance, in the Follow application shown in Figure 3.2, `dests` on Line 9 is a shared variable including the destinations to be visited by the robots; on Line 13, `pick` is a local variable which determines whether the robot is picking a new destination waypoint or currently moving towards a destination.

After declaring these variables, programmers can optionally specify the initial values of program variables (rule *Init*). In Follow, we use the `init` block to initialize the set of shared destinations to be visited by the robots on Line 16.

The main body of the program comprises of a set of events (rule *Event*) which has a Boolean precondition (**pre**) and an effect (**eff**). The effect of an event is also a statement (rule *Effect*). Each event has a name, a precondition for its execution, and an effect consisting of statements. For instance, in Follow, the *PickDest* event on Line 18 can be executed only when `pick` is True, and its effect determines how the robot picks its destination and how it sets its target actuator.

A statement (rule *Stmt*) in *Koord*, like those in most imperative languages, can be a conditional statement, a function call, a variable assignment, or a block of statements. Mutual exclusion is always an important feature when shared variables are involved. *Koord* provides a locking mechanism using the keyword **atomic** to update the shared variable safely. In Follow, the keyword **atomic** is used to update the shared set of destinations by the robots during the `PickDest` event. This application does not allow the robots to pick a destination that another robot has already picked. The `atomic` keyword indicates that only one robot is allowed to execute the statements on Lines 21 to 25 during a round of execution.

To discuss the behavior of such *Koord* application programs and provide a formal semantics for *Koord*, we need to establish the notion of system and robot state.

3.2 CONFIGURATIONS

We now describe the system state, or *system configurations* used in defining *Koord* semantics. Figure 3.3 shows a general system configuration. We denote the set of all possible system configurations by $\mathbb{C}$. A *system configuration* is a tuple $\boldsymbol{c} = (\{L_i\}_{i \in \mathtt{ID}}, S, \tau, \mathit{turn})$, where

(i) $\{L_i\}_{i \in \mathtt{ID}}$ or $\{L_i\}$ in short, is an indexed set of *robot configurations*–one for each participating robot. L_i refers to the configuration of the i-th element, i.e., the i-th robot in the system.

(ii) $S : \mathit{Var} \mapsto \mathit{Val}$ is the *global memory*, mapping all shared variable names to their values.

(iii) $\tau \in \mathbb{R}_{\geq 0}$ is the *global time*.

(iv) $\mathit{turn} \in \{\mathtt{prog}, \mathtt{env}\}$ is a binary *bookkeeping* variable indicating whether program or environment transitions are being processed.

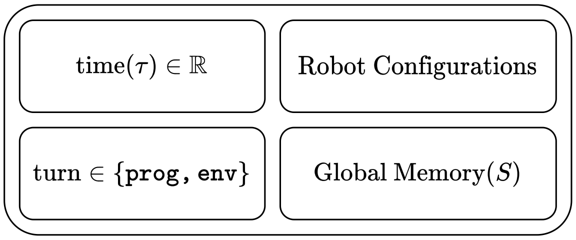
time(τ) $\in \mathbb{R}$
Robot Configurations
turn $\in \{\mathtt{prog}, \mathtt{env}\}$
Global Memory(S)

Figure 3.3: A general system configuration.

Bookkeeping variables are invisible in the language syntax and only used in the semantics. The variable *turn* for the system configuration is a bookkeeping variable. We will see later

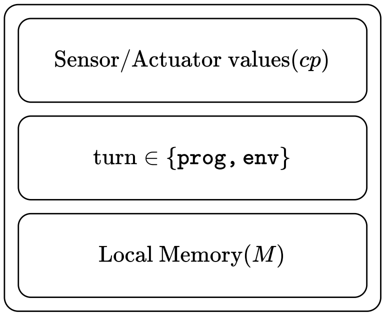

Figure 3.4: A general robot configuration.

in Section 3.4 how the semantics uses this variable to achieve the separation of platform-dependent and platform-independent concerns in the semantics. We now define the robot configurations which define the state of individual robots in the system.

A *robot configuration* is used to specify the semantics of each robot. Figure 3.4 shows a general robot configuration. Given a *Koord* program P, we define *Var* to be the set of shared and local variables, *Val* to be the set of values that an expression in *Koord* can evaluate to, *CPorts* to be the set of sensor and actuator ports of the controller, and *Events* to be the set of events in P. The configuration for robot i is a tuple $L_i = (M, cp, turn)$, where

(i) $M : Var \mapsto Val$ is its *local memory* mapping both local and shared variables to values. Note that this implies that M includes a copy of shared variable values.

(ii) $cp : CPorts \mapsto Val$ is the mapping of sensor and actuator ports to values.

(iii) $turn \in \{\texttt{prog}, \texttt{env}\}$ is a bookkeeping variable indicating whether this robot is executing a program or environment transition.

For readability, we use the dot ("$.$") notation to access components of a robot configuration L. For example, $L.M$ means accessing the local memory M in the tuple L.

We mentioned in Section 2.1 that a *Koord* application is deployed on a fleet of `N_sys` robots. Each robot runs an instance of the same program. Each robot is assigned a unique index `pid` from a known set of identifiers $\texttt{ID} = \{0, 1, \ldots, \texttt{N\_sys} - 1\}$. For a system of `N_sys` robots running the Follow application, an example configuration for robot i has the form:

$$\begin{aligned} L_i = &([\texttt{dests} \mapsto v_1, \texttt{pick} \mapsto v_2, \texttt{currentDest} \mapsto v_3], \\ &[\texttt{target} \mapsto t_1, \texttt{psn} \mapsto t_2, \texttt{done} \mapsto t_3], \texttt{prog}), \end{aligned} \tag{3.1}$$

where $v_1, v_2, v_3, t_1, t_2, t_3 \in Val$.

An example system configuration looks like

$$c = (\{L_i\}, [\texttt{dests} \mapsto s_1], 100.0, \texttt{env}), \tag{3.2}$$

. Since `dests` is a shared variable, it appears once in the local memory of each robot and once again in the shared memory.

3.3 KOORD EXECUTION ROUNDS

As mentioned earlier in Chapter 2, the semantics of a *Koord* program execution is based on synchronous *rounds*. Each round is divided into *program transitions* and *environment transitions* that update the *system configuration*. Program transitions are comprised of event executions by robots. In each round, each robot performs at most one event. We model the update performed by a single robot executing an event as an instantaneous transition that updates the program variables and potentially actuator ports; however, different events executed by different robots may interleave in an arbitrary order.

In between the events of successive rounds, $\delta > 0$ duration of time elapses, where the program variables remain constant while the values held by the sensor and actuator ports may change.

We model these changes as environment transitions that advance time and the sensor and actuator ports. Thus, each round consists of a burst of (at most `N_sys`) program transitions followed by an environment transition. This alternating program and environment transition model is a standard one for distributed systems where computation speed is much faster than the speed of communication [52, 6]. Such models are also standard for hybrid automaton models where computation is faster than physical movements [51, 42].

The execution semantics for a *Koord* program captures the separation of the platform-independent distributed program behaviors and the platform-specific controller behaviors (the program and environment transitions) of the robots through *rewrite rules*. Rewrite rules at various levels: *Robot*, *System*, and *Expression* are used to specify the semantics of a *Koord* program, allowing us to create a framework for formal analysis. We first describe some of the semantic rules at the robot level.

3.4 ROBOT-LEVEL SEMANTICS OF EVENT EXECUTION AND DYNAMICS

During a program transition, each robot executes an event's effect, which is a sequence of statements. We express the semantics of these statements as rewrite rules for statement

$$\frac{\begin{array}{l} L.turn = \texttt{prog} \\ \wedge\text{``}Name\textbf{: pre: } Cond \textbf{ eff: } Body\text{''} \in Events \wedge [\![Cond]\!]_{S,L} \end{array}}{\langle S, L, \oplus\rangle \rightarrow_{stmt} \langle S, L, Body\rangle} \quad \textsc{SelectEvent}$$

$$\langle S, L, \oplus\rangle \rightarrow_{stmt} \langle S, L, \cdot\rangle \quad \textsc{SkipEvent}$$

$$\langle S, (M, cp, \texttt{prog}), \cdot\rangle \rightarrow_{stmt} \langle S, (M, cp, \texttt{env}), \cdot\rangle \quad \textsc{EndEvent}$$

$$\frac{turn = \texttt{env} \wedge \forall x \in Keys(S), M' = M[x \mapsto S[x]] \wedge cp' = f(cp, \delta)}{\langle S, (M, cp, \texttt{env})\rangle \rightarrow_{env} \langle S, (M', cp', \texttt{prog})\rangle} \quad \textsc{RobotEnv}$$

Figure 3.5: Partial per robot semantic rules for *Koord*.

semantics of the following type:

$$\rightarrow_{stmt} \subseteq (\mathbb{S} \times \mathbb{L} \times (Stmt \cup \{\oplus, \cdot\})) \mapsto \wp(\mathbb{S} \times \mathbb{L} \times Stmt \cup \{\cdot\}), \tag{3.3}$$

where $\mathbb{S}$ refers to the set of all possible valuations of the global memory, $\mathbb{L}$ refers to all possible robot configurations, and *Stmt* refers to the set of all possible statements allowed by *Koord* syntax. We use internal syntactic symbols '$\oplus$' and '$\cdot$', (which are not part of the *Koord* syntax themselves) to represent control flow in *Koord* programs, as we will see in the discussion on per-robot semantics. '$\oplus$' denotes nondeterministic selection of events, and '$\cdot$' indicates an "empty" statement.

The $\rightarrow_{stmt}$ relation takes as input a tuple of ((i)) a global memory, ((ii)) a robot configuration, and ((iii)) a statement,and maps it to a set of such tuples.

We first go into some detail of the $\rightarrow_{stmt}$ rewrites, that specify each robot's behavior during a program transition. The semantics uses such rules to modify individual *robot* configurations.

3.4.1 Event Execution Semantic Rules

Events are the main computational blocks in a *Koord* program. We present the core semantic rules for a robot executing an event in a *Koord* program. Rule SelectEvent in Figure 3.5 shows that any event may be executed when the precondition *Cond* is evaluated to true. After replacing $\oplus$ with the event effect *Body*, the $\oplus$ rule ensures that only one event is selected and executed. For Follow, if the program counter indicates $\oplus$, then the only rule applicable by the semantics is the SelectEvent rule.

Consider a robot with a configuration as follows:

$$\begin{aligned} L_i =& ([\texttt{dests} \mapsto [(100,100,0)], \texttt{pick} \mapsto \mathit{True}, \texttt{currentDest} \mapsto (200,10,0)],\\ & [\texttt{target} \mapsto (200,10,0), \texttt{psn} \mapsto (100,10,0), \texttt{done} \mapsto \mathit{False}], \texttt{prog}). \end{aligned} \tag{3.4}$$

The tuple $\langle[\mathit{dests} \mapsto [(100,100,0)]], L_i, \oplus\rangle$ is mapped to $\langle[\mathit{dests} \mapsto [(100,100,0)]], L_i, B\rangle$ by the SelectEvent rule, where B refers to Lines 20 to 25 of the program in Follow.

The robot then executes the event effect following the semantics of each statement in *Body*. Rule SkipEvent allows the robot to skip the event altogether. At the end of the event, the sequence of statements becomes empty '$\cdot$'. Rule EndEvent then makes sure the robot sets its *turn* to `env`, indicating that an environment transition will occur afterward.

While $\rightarrow_{stmt}$ rewrites define each robot's behavior during a program transition, we separate the platform-dependent semantics of how each robot interacts with the environment (including other robots) using environment transition rules of the following type:

$$\rightarrow_{env} \subseteq (\mathbb{S} \times \mathbb{L}) \mapsto \wp(\mathbb{S} \times \mathbb{L}), \tag{3.5}$$

which takes the global memory and a robot configuration as input.

Rule RobotEnv simply states that the new local memory M' is the old local memory M updated with the global memory S; thus ensuring that all robots have consistent shared variable values before the next program transition. It is only applied when the *turn* of the system is `env`. For Follow, when a robot i updates the shared list `dests` by removing a destination from it, it writes to the shared memory. Before the next round, every other robot copies the shared memory into its local memory, so each robot has an updated list of destinations.

To define the executable $\mathbb{K}$ semantics of *Koord* applications, we must provide executable descriptions for the environment transitions. The type of this executable object (f) is defined by *CPorts*, namely, $f : [\mathit{CPorts} \mapsto \mathit{Val}] \times \mathbb{R}_{\geq 0} \mapsto [\mathit{CPorts} \mapsto \mathit{Val}]$. Given old sensor and actuator values and a time point, f should return the new values for all sensor and actuator ports, making it a blackbox for simulating the dynamics of each robot in the system.

New sensor readings cp' are then obtained by evaluating the blackbox dynamics f with time δ. For Follow, Figure 3.6 shows that given a robot currently at position (100,0,0) with its target set to (200,10,0), f returns the position of the robot at δ time from now.

In actual execution, the controller would run the program on hardware, whose sensor ports evolve for δ time between program transitions. This formalization allows the ports to behave arbitrarily over δ-transitions. Hence in verification, additional assumptions over the behavior

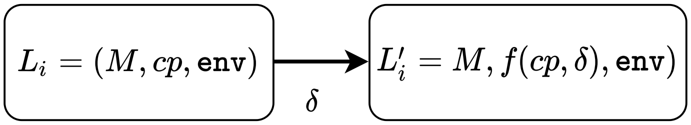
$L_i = (M, cp, \texttt{env})$
δ
$L'_i = M, f(cp, \delta), \texttt{env})$

Figure 3.6: Blackbox dynamics simulated by f updates the sensor and actuator values of the robot at δ time.

of the sensor and actuator ports are needed. Finally, the *turn* of the robot reverts to `prog`.

3.4.2 Assignment and Control Flow Semantic Rules

Aside from the broader semantics discussed above, during program transitions, *Koord* semantics include rewrite rules that dictate the impact of the shared memory abstractions on each of the robot configurations, control flow, among others. We illustrate a few of these rules, in Figure 3.7. The rules StmtSeq1 and StmtSeq2 show how a statement representing a sequence of statements executes. Rule LvarAssign and Rule SvarAssign show the semantic rules for local and shared variable assignments, respectively, are also examples of statement-level rules.

$$\frac{\langle S, L, St\rangle \rightarrow_{\mathit{stmt}} \langle S', L', St'\rangle}{\langle S, L, St\ StList\rangle \rightarrow_{\mathit{stmt}} \langle S', L', St'\ StList\rangle} \qquad \textsc{StmtSeq1}$$

$$\langle S, L, \cdot\ StList\rangle \rightarrow_{\mathit{stmt}} \langle S, L, StList\rangle \qquad \textsc{StmtSeq2}$$

$$\frac{x \in \mathit{Keys}(S) \wedge x \in \mathit{Keys}(L.M) \wedge L'.M = L.M[x \mapsto v]}{\langle S, L, x = v\rangle \rightarrow_{\mathit{stmt}} \langle S[x \mapsto v], L', \cdot\rangle} \qquad \textsc{SvarAssign}$$

$$\frac{x \notin \mathit{Keys}(S) \wedge x \in \mathit{Keys}(L.M) \wedge L'.M = L.M[x \mapsto v]}{\langle S, L, x = v\rangle \rightarrow_{\mathit{stmt}} \langle S, L', \cdot\rangle} \qquad \textsc{LvarAssign}$$

Figure 3.7: Example statement level semantic rules for *Koord*.

For Follow, the `pick` variable is updated on Line 30. Given a robot with configuration

$$\begin{aligned} L_i =&([\texttt{dests} \mapsto [(100,100,0)], \texttt{pick} \mapsto \mathit{True}, \texttt{currentDest} \mapsto (200,10,0)],\\ &[\texttt{target} \mapsto (200,10,0), \texttt{psn} \mapsto (200,10,0), \texttt{done} \mapsto \mathit{True}], \texttt{prog}). \end{aligned} \qquad (3.6)$$

$\langle[\mathit{dests} \mapsto [(100,100,0)]], L_i, \mathit{pick} = \mathit{False}\rangle$ maps to $\langle[\mathit{dests} \mapsto [(100,100,0)]], L'_i, \cdot\rangle$, where $L'_i = L_i.M[\mathit{pick} \mapsto \mathit{False}]$ by the LvarAssign rule.

3.5 EXPRESSION-LEVEL SEMANTICS

Evaluating these robot-level rules requires expression-level rules, which include variable lookup, arithmetic, boolean, and relational operations. We demonstrate a few illustrative examples below. The expression level semantics are given by rewrite rules of the type

$$\rightarrow_E \subseteq (\mathbb{S} \times \mathbb{L} \times \mathbb{E}) \times (\mathbb{S} \times \mathbb{L} \times \mathbb{E}), \tag{3.7}$$

where $\mathbb{S}$ is the set of all possible global memory mappings S, $\mathbb{L}$ is the set of all possible values for configurations of a robot, and $\mathbb{E}$ is the set of all possible expressions allowed by the language syntax.

$$\frac{L.M[x] = v}{\langle S, L, x \rangle \rightarrow_E \langle S, L, v \rangle} \qquad \textsc{Var-Lookup-rule}$$

$$\frac{E_1 \rightarrow_E E_1'}{\langle S, L, E_1 + E_2 \rangle \rightarrow_E \langle S, L, E_1' + E_2 \rangle} \qquad \textsc{Aexpr-Add-rule-1}$$

$$\frac{E_1 \in \mathit{Val} \wedge E_2 \rightarrow_E E_2'}{\langle S, L, E_1 + E_2 \rangle \rightarrow_E \langle S, L, E_1 + E_2' \rangle} \qquad \textsc{Aexpr-Add-rule-2}$$

$$\frac{v_1 + v_2 \rightarrow_E v_3}{\langle S, L, v_1 + v_2 \rangle \rightarrow_E \langle S, L, v_3 \rangle} \qquad \textsc{Aexpr-Add-rule-3}$$

Figure 3.8: Partial expression semantic rules for *Koord*.

The variable lookup rule Var-Lookup-Rule states that every robot has a local copy of every variable in the program. If a robot is evaluating an expression involving variable x, it will replace x with the current value v from the local memory M. $M[x]$ here obtains the value corresponding to the key x.

Figure 3.8 also includes the rules for addition. They are fairly standard: the execution first evaluates the left subexpression (Aexpr-Add-rule-1); given that left is already evaluated fully (i.e., it is in *val*), it evaluates the right subexpression (Aexpr-Add-rule-2); finally, it adds the two values of fully evaluated subexpressions (Aexpr-Add-rule-3).

3.6 SYSTEM-LEVEL SEMANTICS

Now we turn to another set of important design choices, which capture the system's behavior as a whole with interleaving program and environment transitions. For system-level semantics, the rewrite rule is a mapping from an initial system configuration to a set of configurations. It has the following type:

$$\rightarrow_G \subseteq \mathbb{C} \mapsto \wp(\mathbb{C}). \tag{3.8}$$

where that $\mathbb{C}$ is the set of all possible system configurations.

The bookkeeping variable *turn* is used by the system to determine whether the system (all robots in the system) is performing a program transition or an environment transition.

An event is *enabled* when its precondition evaluates to *True* in the current configuration. Rule EventTrans expresses that given a system configuration $\boldsymbol{c} = (\{L_i\}, S, \tau, \texttt{prog})$, a robot i with the configuration L_i first selects an enabled event, executes the event via a sequence of $\rightarrow_{stmt}$ rewrites, and sets its own *turn* to `env` at the end of the event execution. The system itself goes from a configuration $\boldsymbol{c}$ to $\boldsymbol{c}' = (\{L_i'\}, S', \tau, \texttt{prog})$, with possibly different robot configurations and shared memory depending on whether any statement executed resulted in writes to shared variables. Additionally, the system can display nondeterministic behaviors arising from different robots executing their events in different orders.

The system executes an environment transition only when the local *turn* of each robot is `env`. After all robots enter the `env` turn, rule EndProgTrans sets the global *turn* from `prog` to `env` indicating the end of program transition, and an environment transition will occur afterward.

$$\frac{\forall i \in \texttt{ID}, L_i.turn = \texttt{env}}{(\{L_i\}, S, \tau, \texttt{prog}) \rightarrow_G (\{L_i\}, S, \tau, \texttt{env})} \qquad \textsc{EndProgTrans}$$

$$\frac{\begin{array}{c}\forall i \in \texttt{ID}, \langle S, L_i \rangle \rightarrow_{env} \langle S, L_i' \rangle \\ \wedge\ L_i.turn = \texttt{env} \wedge L_i'.turn = \texttt{prog}\end{array}}{(\{L_i\}, S, \tau, \texttt{env}) \rightarrow_G (\{L_i'\}, S, \tau + \delta, \texttt{prog})} \qquad \textsc{EnvTrans}$$

$$\frac{\begin{array}{c}\exists i \in \texttt{ID}, \langle S, L_i, \oplus \rangle \rightarrow_{stmt} \langle S', L_i', \cdot \rangle \\ \wedge\ L_i.turn = \texttt{prog} \wedge L_i'.turn = \texttt{env}\end{array}}{(\{L_i\}, S, \tau, \texttt{prog}) \rightarrow_G (\{L_i'\}, S', \tau, \texttt{prog})} \qquad \textsc{EventTrans}$$

Figure 3.9: System semantic rules for *Koord*.

Rule EnvTrans shows the evolution of the system configuration after the rule EndProgTrans is applied. This rule synchronizes the *environment* transitions of the robots and advances the global time from τ to $\tau + \delta$. Note that the separation of the distributed coordination and control happens explicitly in these rules: the rule EndProgTrans determines when the program transitions (distributed coordination) end and rule EnvTrans determines when the environment transitions(control) occur.

3.7 SYNCHRONIZATION AND CONSISTENCY ASSUMPTIONS

Our semantic rules in Section 3.4 assume that all program transitions of *Koord* program take *zero* time. However, the environment transitions take δ time for the evolution of the sensor and actuator ports together with the update of the local memory from the global memory.

To reiterate, the following are the timing requirements from rule EventTrans and EnvTrans:

(i) a program transition takes *zero* time,

(ii) new values of sensor and actuator ports are sampled at the end of each round

(iii) shared variables should reach consistent values within δ time, and

(iv) we can use a global clock to synchronize each δ-time round.

The first two requirements are approximately satisfied if the time taken to complete a program transition is negligible compared to δ. Furthermore, δ can be a common multiple of the sampling intervals of all controller ports in use. These constraints are reasonable when computation and communication are comparatively much faster. Using the `Motion` module as an example, our position sensor on each device publishes every 0.01 sec (100Hz) while the CPU on each drone is 1.4 GHz. If we set δ to be 0.01 sec, a program transition taking 10K CPU cycles is still less than 0.1% of δ. We discuss how the choice of δ can impact program behavior in Chapter 5, Section 5.3.

How to satisfy requirements (iii) and (iv), is a common research question in distributed computing with extensive literature. We can achieve a global clock with existing techniques that synchronize all local clocks on robots. The toolchain in [7] uses message passing to implement distributed shared memory for shared variables. It ensures that the time taken to propagate values through messages and reach consistency is smaller than δ, and the update is visible in the next round of program transitions for all robots. We, therefore, conclude our round-based semantics with shared memory is a reasonable abstraction.

The semantic rules we discussed realize the (distributed) and computational components of the *Koord* system. We designed the memory-consistency model, and the synchronization model of *Koord* to complement the separation and analysis of the platform-independent program transitions and platform-dependent environment transitions.

3.8 SUMMARY

In this chapter, we presented the formal syntax of the *Koord* language and illustrated it using a waypoint following example. We then introduced the notion of *configurations*, which we use to define the formal semantics of the *Koord* language. We presented the core semantics rules for the expression evaluation, individual robot behavior, and the behavior of the system executing a *Koord* application program. In the next chapter, we show how we implemented these executable semantics in $\mathbb{K}$. In Chapter 5, we will demonstrate how we use executable semantics to perform reachability analysis and verification of *Koord* programs.

CHAPTER 4: IMPLEMENTING AN EXECUTABLE SEMANTICS OF KOORD

We have built the semantics of *Koord* in the $\mathbb{K}$ framework[1]. $\mathbb{K}$ is a rewriting-based executable framework for defining language semantics. Given a syntax and a semantics of a language, $\mathbb{K}$ generates a parser and an interpreter for application programs written in the language at no additional development cost. In this chapter, we discuss how we implemented our *Koord* semantics in $\mathbb{K}$.

4.1 EXECUTION OF A KOORD PROGRAM IN THE $\mathbb{K}$ FRAMEWORK

We first parse each *Koord* program written in syntax presented in Figure 3.1 using standard indentation parser that we implemented in Python. We add parameters including the required rounds of execution, the duration of a round δ and the number of robots `N_sys` along with the parsed program as input for the $\mathbb{K}$ executable semantics.

We can view the execution of a *Koord* program in $\mathbb{K}$ proceeds in *stages*. We specify the code stages in $\mathbb{K}$ as shown in Figure 4.1.

```
syntax CodeStage ::= "Preproc"
                   |  "ActPreproc"
                   |  "SensorPreproc"
                   |  "AWPreproc"
                   |  "ARPreproc"
                   |  "EventPreproc"
                   |  "ParamPreproc"
                   |  "LocPreproc"
                   |  "Init"
                   |  "Prog"
                   |  "Update"
                   |  "Env"
                   |  "End"
```

Figure 4.1: Declaring possible execution stages of a *Koord* program in $\mathbb{K}$.

Figure 4.2 shows the stages of execution of the *Koord* code in our implementation of of the executable semantics. First, the code is preprocessed through several intermediate preprocessing stages for (i) sensor declarations (`SensorPreproc`), (ii) actuator declarations (`ActPreproc`), (iii) shared variable declarations (`AWPreproc`, `ARPreproc`), (iv) event

[1] The semantics of *Koord* is available at https://github.com/ritwika314/koord

code (`EventPreproc`), (v) execution parameters (`ParamPreproc`), and (vi) local declarations (`LocPreproc`) .

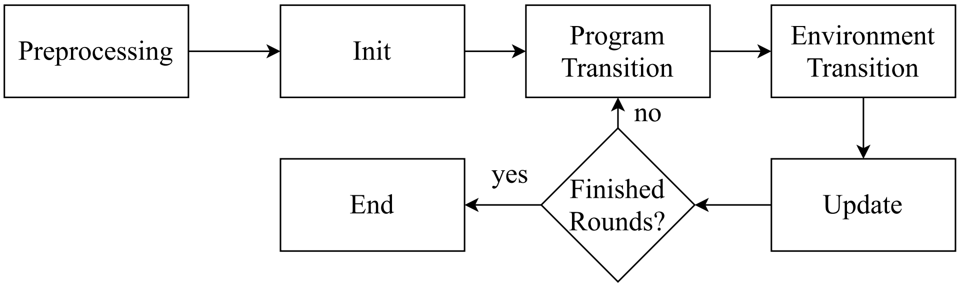

Figure 4.2: Control flow in stages of execution of a *Koord* program in $\mathbb{K}$.

Once the code has been preprocessed, the *Init* block of the program, if it exists, is executed. After that, program (`Prog`) and environment (`Env`) transitions followed by a shared variable update (`Update`) stage are executed repeatedly until the specified number of rounds is finished. After that, the code reaches its end stage (`End`).

4.2 CONFIGURATIONS IN $\mathbb{K}$

In Chapter 3, we expressed the *Koord* semantics using configurations. Semantics in $\mathbb{K}$ is also expressed using configurations, which organize the components in elements called *cells*. Cells are labelled, have types indicating what kind of elements can be contained in them, and help specify rewrite rules. A cell is specified using the notation `<cellname> cell-contents </cellname>`. We show how the components of robot and system configurations correspond to configuration cells in our $\mathbb{K}$ implementation.

4.2.1 System Configurations in $\mathbb{K}$

Figure 4.3 shows a (partial) system configuration. The '. . .' notation indicates that we are omitting the details of some intermediate cells in the configuration. The top level cell is `System`, which indicates that this is the configuration of the system in $\mathbb{K}$. We have omitted some of the details of the cells in the configuration in this figure, and we will address them separately in later sections in this chapter. Recall that a system configuration in our formal semantics is a tuple $(\{L_i\}_{i \in \mathtt{ID}}, S, \tau, \mathit{turn})$, where $\{L_i\}$ is the set of robot configurations, S is the global memory, τ is the global time, and *turn* indicates whether program or environment transitions are being processed. The `System` configuration specified in $\mathbb{K}$ captures our notion of system configurations as follows:

```
configuration
  <System>
    <robot multiplicity = "*" type="Set">
    ...
    </robot>
    ...
    <codeStage> Preproc </codeStage>
    ...
    <AWEnv> .Map </AWEnv>
    <AWStore> .Map </AWStore>
    <AREnv> .Map </AREnv>
    <ARStore> .Map </ARStore>
    ...
    <tau> 0.0 </tau>
    ...
  </System>
```

Figure 4.3: Partial $\mathbb{K}$ System configuration.

1. The cell `robot` corresponds to robot configurations L_i. For the `robot` cell, the statement '`multiplicity = *`' indicates that there can be multiple robot cells contained in the `System` configuration, and '`type = Set`' indicates that each robot cell is unique.

2. The cells `AWEnv`, `AWStore`, `AREnv`, and `ARStore` collectively makeup the global memory S in the system configuration. `AWEnv` contains the mapping of *allwrite* variables to *memory locations*, and `AWStore` contains a mapping of memory locations to the values of the *allwrite* variables. The notation `.Map` indicates that the the type of elements that can be stored in these cells are maps, and that there is currently an empty map in the cells. Similarly, `AREnv` contains the mapping of *allread* variables to *memory locations*, and `ARStore` contains a mapping of memory locations to the values of the *allread* variables.

3. The `tau` cell contains the global time or τ. It is initialized as 0.0, which also indicates that it has type float.

4. The `codeStage` cell actually corresponds to *turn*. The code stages aside from `Prog` and `Env` are conveniences for specifying the executable semantics of *Koord* in $\mathbb{K}$.

Having defined system configurations, we now turn to robot configurations in $\mathbb{K}$. We use the `robot` cell which we omitted expanding in the system configuration to define robot configurations.

4.2.2 Robot Configurations in $\mathbb{K}$

Recall that we defined robot configurations in Chapter 3 as a tuple $L_i = (M, cp, turn)$ where M is the local memory of each robot, *cp* a mapping of sensor and actuator ports to their values, and *turn* indicates whether the robot is performing a program or an environment transition. Figure 4.4 shows the partial robot configuration in $\mathbb{K}$.

```
<robot multiplicity = "*" type="Set">
  <k> $PGM:Pgm </k>
  <env> .Map </env>
  <store> .Map </store>
  ...
  <senseEnv> .Map </senseEnv>
  <senseStore> .Map </senseStore>
  <actEnv> .Map </actEnv>
  <actStore> .Map </actStore>
  ...
  <LocAWEnv> .Map </LocAWEnv>
  <LocAWStore> .Map </LocAWStore>
  <LocAREnv> .Map </LocAREnv>
  <LocARStore> .Map </LocARStore>
  ...
  <pid> 0 </pid>
  <turn> Prog </turn>
</robot>
```

Figure 4.4: Partial Robot Configuration in $\mathbb{K}$.

The `robot` configuration specified in $\mathbb{K}$ captures our notion of robot configurations as follows:

1. The cells `env` contains the mapping of local variables to memory locations, and `store` contains a mapping of memory locations to the values local variables. The cells `LocAWEnv`, `LocAWStore`, `LocAREnv` and `LocARStore` constitute the robot's local copies of the shared variables. Together, these cells make up the local memory M.

2. The `senseEnv` cell contains the mapping of the robot's sensor ports to memory locations, and `senseStore` contains a mapping of memory locations to the values of the robot's sensor ports. Similarly, `actEnv` contains the mapping of actuator ports to memory locations, and `actStore` contains a mapping of memory locations to the values of the actuator ports. These cells comprise the *cp* component of the robot configuration.

3. The `turn` cell corresponds to *turn*.

Note that the robot configuration in $\mathbb{K}$ also consists of a `pid` cell, which is the same as the unique integer identifier `pid` of the robot.

Before we look at how we implement executable *Koord* semantics in $\mathbb{K}$, we first discuss some features of $\mathbb{K}$ that are essential to be able to express the semantics.

4.3 DEFINING EXECUTABLE SEMANTICS IN $\mathbb{K}$

One can view a language semantics naturally as a set of rewrite (reduction) rules over configurations. $\mathbb{K}$ allows underspecification of rewrite rules, meaning, only the rewrite rules affecting part of the configurations need to be specified if the rule doesn't affect the other parts of the rule. The rewrite rules in Sections 4.4, 4.5 and 4.6 heavily utilize this functionality.

$\mathbb{K}$ provides computational structures, or computations to express language semantics. Computations are are sequences of computational tasks, where each computational task is a term over an extended core syntax. Computations are used to define evaluation strategies of the various language constructs. We provide the rewrite rules that convert the core language syntax into computations.

$\mathbb{K}$ uses a special cell called the `k` cell to store the current computation in the program. Each robot has a `k` cell to store its computation, as shown in Figure 4.4.

In our implementation, we heavily rely on a feature of $\mathbb{K}$ that allows us specify how computations can be sequentialized: "$\sim>$", which is read as "followed by. If $\mathtt{t_1}, \mathtt{t_2}, \ldots, \mathtt{t_n}$ are computations, then $\mathtt{t_1} \sim> \mathtt{t_2} \sim> \ldots \sim> \mathtt{t_n}$ represents that the computation consisting of $\mathtt{t_1}$ is followed by $\mathtt{t_1}$ is followed by $\ldots$, is followed by $\mathtt{t_1}$.

4.3.1 Bookkeeping Cells

We use several bookkeeping cells to help simplify the specification of rewrite rules. Figure 4.5 shows the some of the bookkeeping cells used at the system level: we only present the ones that we will require to demonstrate the rewrite rules in this chapter.

(i) `numBots` stores the number of robots in the system `N_sys`.

(ii) `numProcessed` stores the number of robots that have been preprocessed and are ready to move to the initialization stage of execution. Since the rewrite rules are applied sequentially, only one robot applies a rewrite rule at a time. This is required in order to ensure that no robot moves on to initialization before all robots have been preprocessed.

```
<System>
  ....
  <numBots> 1 </numBots>
  <numProcessed> 0 </numProcessed>
  <updateReady> 0 </updateReady>
  <inProg> 0 </inProg>
  <inEnv> 0 </inEnv>
  <active> -1 </active>
  <delta> 0 </delta>
  <rounds> 0 </rounds>
  <module> .K </module>
  <numEvents> 1 </numEvents>
  <effmap> .Map </effmap>
</System>
```

Figure 4.5: System level bookeeping cells in $\mathbb{K}$.

(iii) `updateReady` stores the number of robots that are ready to move to the update stage of execution after a round of program and environment transitions, during which each robot updates its local copies of the shared variables from the shared memory.

(iv) `inProg` stores the number of robots that are currently executing their program transitions. This is used to conveniently update the `turn` of the system when the `turn` of each robot becomes `env`.

(v) `inEnv` stores the number of robots that are currently executing their environment transitions. This is used to conveniently update the `turn` of the system when the `turn` of each robot becomes `prog`.

(vi) `active` stores the `pid` of the robot that is currently executing its event. This is helpful in documenting the order of execution of events.

(vii) `delta` stores the duration of each round.

(viii) `rounds` stores the number of rounds that the *Koord* application is required to execute.

(ix) `module` stores the module (if any) of the *Koord* program. The type `.K` indicates that it is a syntactic object in *Koord*.

(x) `numEvents` stores the number of events. This is used to evaluate the preconditions of all events before a program transition.

(xi) `effmap` is a map of each event to its effect.

```
     <robot multiplicity = "*" type="Set">
       ...
       <evPreMap> .Map </evPreMap>
     </robot>
```

Figure 4.6: Robot level bookkeeping cells in $\mathbb{K}$.

Figure 4.6 shows a bookkeeping cell used by each robot, `evPreMap`, which is used to store the evaluation of each event precondition. This is used to pick which event to execute.

4.4 ROBOT-LEVEL SEMANTICS IN $\mathbb{K}$

We extend the core syntax by the following computational terms :

(i) `startEvent`, which we use to indicate that the effect of an event will start execution.

(ii) `endEvent`, which we use to indicate ending an event execution.

(iii) `callBB`, to indicate that the blackbox dynamics corresponding to the declared module should be used to update the sensor/actuator ports. `endCallBB` to indicate that the robot's sensor and actuator ports have been updated.

4.4.1 Event Execution by Robots

After each robot finishes their preprocessing stage of execution, `startEvent` is the computation which remains at the in the `k` cell of each robot. Figure 4.7 shows how the each robot executes an event starting from that point.

The `effmap` cell stores a mapping of events to their effects denoted by a variable `EFFMAP`. If there is an event `L` whose precondition evaluates to true as indicated by `L |-> true` in the `evPreMap` cell of the robot, then the `startEvent` computational task rewrites to the effect of the event stored in `EFFMAP[L]`. Then, this computation should be followed by `endEvent`, which is followed by `callBB`, and then `endcallBB`. This rule can only be applied when the `turn` of the robot is `Prog`. The `active` cell is set to the robot's `pid` to indicate that robot `I` will be executing an event. We use the `active` cell to that there no interleaving rewrites during an event execution by another robot.

Figure 4.8 shows our implementation of EndEvent in $\mathbb{K}$.

```
rule <robot>
        <k> startEvent => EFFMAP[L] ~> endEvent
              ~> callBB ~> endcallBB ... </k>
        <id> I </id>
        <evPreMap> ... L |-> true ... </evPreMap>
        <turn> Prog </turn>
        ...
     </robot>
     <codeStage> Prog </codeStage>
     <effmap> EFFMAP </effmap>
     <active> -1 => I </active>
```

Figure 4.7: SELECTEVENT rule in 𝕂

```
rule <robot> <k> endEvent => . ... </k>
      <turn> Prog => Env </turn>
      ...
     </robot>
      <active> _ => -1 </active>
      <inEnv> N => N +Int 1 </inEnv>
```

Figure 4.8: ENDEVENT rule in 𝕂.

When a robot finishes executing its event, the `endEvent` computation rewrites to empty (`.`), and the remainder of the computation in the `k` cell can execute next. The `turn` of the robot is set to `Env`, and the bookkeeping cell `inEnv` is incremented by one to show that one more robot is ready to start its environment transition. The `active` cell is rewritten to -1, to allow another robot to execute its event if required.

4.4.2 Variable Updates

Figure 4.9 shows how local variables are updated in our implementation of the LVARASSIGN rule in 𝕂. Variables in our implementation are assigned a builting 𝕂 type `Id`, which stands for "string identifier". `V:Id` indicates that `V` is a term with type `Id`. When a variable `X` is assigned to `V` of type Val, which we defined to be the set of values that a variable in *Koord* can take, its value is updated as follows. Given memory location storing the value of `X` is `L` in the `env` cell of the robot, whatever value (_) is currently stored at the location `L` is rewritten to `V` in the `store` cell. This rule is only applicable to a currently `active` robot during the `Prog` stage. Note that the ordering of cells in the rewrite rule is immaterial.

```
  rule <robot> <k> X:Id = V:Val ; => . ... </k>
        <pid> I </pid>
        <env> ... X |-> L ... </env>
        <store> ... L |-> (_ => V) ... </store>  ... </robot>
        <active> I </active>
        <codeStage> Prog </codeStage>
```

Figure 4.9: LvarAssign rule in 𝕂.

We now turn to shared variable updates. Figure 4.10 shows how shared variables are updated in our implementation of the SvarAssign rule in 𝕂.

```
  rule <robot> <k> X:Id = V:Val ; => . ... </k>
        <pid> I <pid>
        <LocAWEnv> ... X |-> L ... </LocAWEnv>
        <LocAWStore> ... L |-> (_ => V) ... </LocAWStore>
        <turn> Prog </turn> ... </robot>
        <AWEnv> ... X |-> L1 ... </AWEnv>
        <AWStore> ... L1 |-> (_ => V) ... </AWStore>
        <codeStage> Prog </codeStage>
        <active> I </active>

   rule <robot> <k> X:Id [I] = V:Val ; => . ... </k>
        <LocAREnv> ... X |-> L ... </LocAREnv>
        <LocARStore> ... (L +Int I) |-> (_ => V) ... </LocARStore>
        <turn> Prog </turn>
        <pid> I </pid> ... </robot>
        <AREnv> ... X |-> L1 ... </AREnv>
        <ARStore> ... (L1 +Int I)  |-> (_ => V) ... </ARStore>
        <active> I </active>
        <codeStage> Prog </codeStage>
```

Figure 4.10: Rules for shared variable assignment (SvarAssign) in 𝕂.

The first rule in Figure 4.10 shows how allwrite variables are updated. Both the robot's local copy of the variable `X` and the shared memory are updated at the same time. In the second rule in Figure 4.10, the update of an allread variable is shown. Given an allread variable `X`, the robot can only update it at the index corresponding to its own `pid`. The memory location of this allread variable in the shared memory starts at `L1` as indicated by `X |-> L1`, and it is stored at an offset equal to the `pid` of the robot in the `ARStore`

```
    rule <robot> <k> callBB => . ... </k>
        <senseEnv> SE </senseEnv>
        <senseStore> SS => BBSense(M, AE, AS, SE, SS, D) </senseStore>
        <actEnv> AE </actEnv>
        <actStore> AS => BBAct(M, AE, AS, SE, SS, D) </actStore>
        ... </robot>
        <delta> D </delta>
        <module> M </delta>

    rule <robot> <k> endcallBB => . ~> updateMem ... </k> ... </robot>
       <updateReady> N => N +Int 1 </updateReady>
       <codeStage> Env </codeStage>
```

Figure 4.11: Updating sensor/actuator ports through a blackbox function in $\mathbb{K}$.

cell. The local copy of the robot is also analogously updated. Both the rules implementing SvarAssign are applicable only to an `active` robot.

4.4.3 Updating Sensor/Actuator Ports Through a Blackbox Function

Figure 4.11 shows how the blackbox function to update the sensor and actuator ports is called. We extended the core syntax by another computational term `updateMem` to indicate that at the end of a round, each robot updates its copies of the shared variables.

We implemented blackbox functions `BBSense` and `BBAct` for each module, at the backend of K to update the sensor and actuator ports. These functions take as input:

(i) `M` : the module,

(ii) `AE` : the map of actuator ports to memory locations,

(iii) `AS` : the map of memory locations to actuator values,

(iv) `SE` : the map of sensor ports to memory locations,

(v) `SS` : the map of memory locations to sensor values, and

(vi) `D` : the duration of the round δ.

4.4.4 Shared Memory Updates

At the end of a round, we need to update the shared memory.

```
rule <updateReady> N => 0  </updateReady>
    <numBots> N  </numBots>
    <codeStage> Env => Update </codeStage>
    <rounds> I => I -Int 1 </rounds>

rule <robot>
        <k> updateMem =>  endUpdate ... </k>
        <senseStore> SS </senseStore>
        <actStore> AC </actStore>
        <LocAWStore> AW => AWS </LocAWStore>
        <LocARStore> AR => ARS </LocARStore>
          ...
    </robot>
    <numEvents> N </numEvents>
    <AWStore> AWS </AWStore>
    <ARStore> ARS </ARStore>
    <codeStage> Update </codeStage>
```

Figure 4.12: Updating shared memory after a round in $\mathbb{K}$.

Figure 4.12 shows how shared variables are updated after a round. We extended the core syntax by the computational terms `endUpdate` to indicate that a robot has finished updating its memory after a round. The first rule in Figure 4.12 indicates When the number of robots ready to update the shared memory in the `updateReady` cell is the same as the number of robots (`numbots`) the system goes from the `Env` execution stage to the `Update` execution stage. The `updateReady` cell is reset for the next round. The number of remaining rounds is reduced by 1. The second rule shows that the `updateMem` computation rewrites to `endUpdate`, while copying the shared memory maps to the robot's local maps.

The rules in Figure 4.12 and Figure 4.11 implement the RobotEnv in Chapter 3.

4.5 EXPRESSION-LEVEL SEMANTICS

The rule that each subexpression of an arithmetic expression requires to be evaluated to a numerical value can be added in $\mathbb{K}$ by simply adding an attribute "strict" to the description of its syntax in $\mathbb{K}$. The *left* attribute indicates that this operation is left associative. We also specify that numerical values qualify as *result* values, so that the rewriting system doesn't attempt to rewrite them further. The syntactic production in $\mathbb{K}$ subsumes the rules aexpr-add-rule-1 and aexpr-add-rule-2.

```
Expr ::= Expr "+" Expr            [left,strict]
```

Figure 4.13: $\mathbb{K}$ implementation of aexpr-add-rule-1 and aexpr-add-rule-2.

We add rewrite rules for every type of expression that evaluates to a numerical value. Figure 4.14 shows a direct translation of rule aexpr-add-rule-3. For instance, given I_1 and I_2 which are integer values, indicated by `I1:Int` and `I2:Int`, the following rules indicate that their sum should be simply the integer addition of their values. Similarly for floats, the sum should be the float addition of the float values. These rules are specified without any cells as only applied when such an addition is at the top of the computation cell `k`.

```
I1:Int + I2:Int => I1 +Int I2
F1:Float + F2:Float => F1 +Float F2
```

Figure 4.14: $\mathbb{K}$ implementation of aexpr-add-rule-3.

```
  rule <robot> <k> X:Id => V ...</k>
       <env>... X |-> L ...</env>
       <store>... L |-> V:Val ...</store> ... </robot>

  rule <robot> <k> X:Id => V ...</k>
       <LocAWEnv>... X |-> L ...</LocAWEnv>
       <LocAWStore>... L |-> V:Val ...</LocAWStore> ... </robot>

    rule <robot> <k> X:Id [I:Val] => V ...</k>
       <LocAREnv>... X |-> L ...</LocAWEnv>
       <LocARStore>... L +Int I |-> V:Val ...</LocARStore> ... </robot>
```

Figure 4.15: Var-Lookup-Rule in $\mathbb{K}$

Figure 4.15 shows the variable lookup rules. Each robot looks up a local variable through its `env` and `store` cells. For allwrite variables, it uses the `LocAWEnv` and `LocAWStore` cells; for allread variables it uses the `LocAREnv` `LocARStore` cells.

4.6 SYSTEM-LEVEL SEMANTICS

We now discuss some system level rewrite rules which implement the EnvTrans rule in $\mathbb{K}$, as shown in Figure 4.16.

```
rule  <robot>
    <k> endUpdate => startEvent ... </k>
    <turn> Env => Prog </turn> ... <robot>
  <inProg> N => N +Int 1 </inProg>
  <codeStage> Update </codeStage>

rule <inProg> N => 0 </inProg>
  <numBots> N </numBots>
  <codeStage> Update => Prog </codeStage>
  <delta> D </delta>
  <tau> T => T + D </tau>
  <rounds> I </rounds>           requires I >Int 0

rule <inProg> N => 0 </inProg>
  <numBots> N </numBots>
  <codeStage> Update => Final </codeStage>
  <delta> D </delta>
  <tau> T => T + D </tau>
  <rounds> 0 </rounds>
```

Figure 4.16: Rule EnvTrans in $\mathbb{K}$.

The first rule in Figure 4.16 shows that each robot rewrites the `endUpdate` computation to `startEvent`, and changes its `turn` to `Prog`. The bookkeeping cell `inProg` increments by 1 to show that one more robot has finished updating its shared memory.

The second rule shows that when all the robots that have finished updating their shared memory, the `inProg` cell is reset for the next round. The global time in the `tau` cell is incremented by the duration of each round ($\delta = \texttt{D}$) in the `Delta` cell. If there are rounds remaining, then the `codeStage` goes from `Update` to `Prog`. The third rule captures the case when there are no more rounds remaining, and the execution has reached its `Final` stage.

4.7 SUMMARY

We presented the $\mathbb{K}$ implementation of our semantics in this chapter. In Section 4.1, we outlined the various stages of execution of *Koord* code. We then showed how our definition of configurations in Chapter 3 and configurations in $\mathbb{K}$ are consistent with each other. Finally, in Sections 4.4, 4.5 and 4.6, we discussed robot level rewrite rules, expression level rewrite rules and system level rewrite rules corresponding to semantic rules discussed earlier in Chapter 3.

In the next chapter, we discuss how we use *Koord* semantics to verify invariants for *Koord* programs.

CHAPTER 5: SEMANTICS-DRIVEN VERIFICATION OF KOORD PROGRAMS

Verification and debugging of distributed systems is generally a difficult problem: partially due to multiple possible points of failure and nondeterminism in the presence of any amount of asynchrony. While our design requires that *Koord* applications execute in a partially synchronous fashion, various sources of nondeterminism exist, such as the order of writes to shared variables. Further, the robots' dynamic behavior adds to the complexity of the verification problem.

Our verification approach uses the *Koord* executable semantics to explore possible behaviors or *reachable* states (configurations) of a *Koord* program. Traditional verification techniques for verifying the required properties of distributed algorithms typically require embedding a *model* of the system within a theorem proving or verification environment. However, such methods cannot provide the same guarantees for a programming language implementation of the system model. Our executable semantics allows us to compute the system's reachable configurations through program execution and check whether the required properties hold in each of those configurations.

This chapter discusses formal verification methods for *Koord* programs enabled by *Koord* design and semantics and presents some bounded verification results. We also set up preliminaries for verification of inductive invariants for *Koord* applications.

5.1 DEFINING REACHABLE CONFIGURATIONS

Given a set of system configurations $\mathcal{C}$, we define the following sets using the semantic rules of Chapter 3:

(i) $\mathit{Post}(\boldsymbol{c}, i, e)$ returns the set of configurations obtained by robot i executing event $e \in \mathit{Events}$ from a configuration $\boldsymbol{c}$.

(ii) $\mathit{Post}(\mathcal{C}, i)$ returns the set of configurations obtained by robot i executing any event from a configuration in $\mathcal{C}$.

(iii) $\mathit{Post}(\mathcal{C}, \vec{p})$ returns all configurations visited, when robots execute their events in the order $\vec{p}$, where $\vec{p}$ is a sequence of $p_i \in \texttt{ID}$.

(iv) $\mathit{Post}(\mathcal{C})$ is the union of $\mathit{Post}(\mathcal{C}, \vec{p})$ over all orders $\vec{p}$.

(v) $\mathit{End}(\mathcal{C})$ is the set of configurations reached from $\mathcal{C}$ *after* a program transition.

$$
\begin{aligned}
\mathit{Post}(\boldsymbol{c}, i, e) &:= \{\boldsymbol{c}' \mid \exists \boldsymbol{c} \in \mathcal{C}, [\![\mathit{Cond}]\!]_{\boldsymbol{c}.S, \boldsymbol{c}.L_i} \\
&\qquad \wedge \langle \boldsymbol{c}.S, \boldsymbol{c}.L_i, \mathit{Body}\rangle \rightarrow_{\mathit{stmt}} \langle \boldsymbol{c}'.S, \boldsymbol{c}'.L_i, \cdot\rangle\}, \\
\mathit{Post}(\mathcal{C}, i) &:= \bigcup_{e \in \mathit{Events}} \mathit{Post}(\boldsymbol{c}, i, e), \\
\mathit{Post}(\mathcal{C}, \vec{p}) &:= \begin{cases} \emptyset, & \text{if } \vec{p} = () \\ \mathit{Post}(\mathit{Post}(\mathcal{C}, p_0), \vec{p}'), & \text{if } \vec{p} = (p_0, \vec{p}') \end{cases} \\
\mathit{Post}(\mathcal{C}) &:= \bigcup_{\vec{p} \in \mathit{perms}(\mathtt{ID})} \mathit{Post}(\mathcal{C}, \vec{p}), \\
\mathit{End}(\mathcal{C}) &:= \{\boldsymbol{c} \mid \boldsymbol{c} \in \mathit{Post}(\mathcal{C}) \wedge \forall i \in \mathtt{ID}, \boldsymbol{c}.L_i.\mathit{turn} \neq \mathtt{prog}\}.
\end{aligned}
$$

Figure 5.1: Intermediate definitions for defining reachable configurations.

All these definitions can be restricted naturally to individual configurations. Figure 5.1 shows the exact definitions of these sets. In the above, a sequence $\vec{p} = (p_0, \vec{p}')$, is written as a concatenation of the first element p_0 and the suffix $\vec{p}'$. Also, $\mathit{perms}(\mathtt{ID})$ refers to the set of permutations of ID.

Next, we identify configurations that the system reaches during and after an environment transition. Recall that environment transitions capture the evolution of the sensor and actuator ports over a time interval $[0, \delta]$; all other parts of the configuration remain unchanged. Recall from Chapter 3 that our *Koord* semantics defines the environment transitions with an executable oobject which is possibly a blackbox function that captures the dynamics of individual robots.[1] Given such a function f_i for each robot i, we define the function $\mathit{traj} : \mathbb{C} \times [0, \delta] \mapsto \mathbb{C}$ to represent the evolution of the system over a $[0, \delta]$ time interval. The function *traj* is constructed by updating all controller ports *cp* of each robot i using the function f_i that captures their respective dynamics. That is,

$$
\boldsymbol{c}' = \mathit{traj}(\boldsymbol{c}, t) \Leftrightarrow \begin{pmatrix} \forall i \in \mathtt{ID}, \boldsymbol{c}'.L_i.\mathit{cp} = f_i(\boldsymbol{c}.L_i.\mathit{cp}, t) \\ \wedge\ \boldsymbol{c}'.L_i.M = \boldsymbol{c}.L_i.M \\ \wedge\ \boldsymbol{c}'.L_i.\mathit{turn} = \boldsymbol{c}.L_i.\mathit{turn} \\ \wedge\ \boldsymbol{c}'.S = \boldsymbol{c}.S \wedge \boldsymbol{c}'.\tau = \boldsymbol{c}.\tau \\ \wedge\ \boldsymbol{c}'.\mathit{turn} = \boldsymbol{c}.\mathit{turn} \end{pmatrix} \tag{5.1}
$$

Notice that there are additional constraints denoting that all other fields of $\boldsymbol{c}$ and $\boldsymbol{c}'$ stay the same.

The set of all transient system configurations $\mathcal{C}_{[0,t]}$ reached in an interval $[0, t]$ from $\mathcal{C}$ is

[1] For different platforms, this function could be defined in closed form, as solutions of differential equations, or in terms of a numerical simulator.

defined as follows:

$$\mathcal{C}_{[0,t]} := \{\boldsymbol{c}' \mid \exists \tau \in [0,t], \exists \boldsymbol{c} \in \mathcal{C}, \boldsymbol{c}' = \mathit{traj}(\boldsymbol{c}, \tau)\}\,. \tag{5.2}$$

We denote the set of points reached precisely at the end of an environment transition from $\mathcal{C}$ as $\mathcal{C}_{\texttt{env}}$.

$$\mathcal{C}_{\texttt{env}} := \{\boldsymbol{c}' \mid \exists \boldsymbol{c} \in \mathcal{C}, \boldsymbol{c}' = \mathit{traj}(\boldsymbol{c}, \delta)\} \text{ where } \delta \text{ is the time for a round.} \tag{5.3}$$

Now, to conform to our semantics, we carefully define the exact set of configurations reached right at the end of each round without transient configurations. A *frontier* set of configurations $\mathcal{C}^n$ represents those configurations that are reached from $\mathcal{C}$ *exactly when* n rounds are completed. Formally,

$$\mathcal{C}^n := \begin{cases} \mathcal{C}, & \text{if } n = 0 \\ (\mathit{End}(\mathcal{C}^{n-1}))_{\texttt{env}} & \text{otherwise} \end{cases} \tag{5.4}$$

Finally, given a set of configurations $\mathcal{C} \subseteq \mathbb{C}$, we can inductively define the set of all reachable configurations in n rounds:

$$\mathit{Reach}(\mathcal{C}, n) := \begin{cases} \mathcal{C}, & \mathit{if} n = 0 \\ \mathit{Reach}(\mathcal{C}, n-1) \cup\ \mathit{Post}(\mathcal{C}^{n-1}) \cup (\mathit{End}(\mathcal{C}^{n-1}))_{[0,\delta]}, & \text{otherwise} \end{cases} \tag{5.5}$$

Notice that all *Reach* includes the transient configurations reached during both program and environment transitions.

An *invariant* of a *Koord* program is a predicate that holds in all reachable configurations. Invariants can express safety requirements for an application, for instance, that no two robots are ever too close (Collision avoidance), or that robots always stay within a designated area (Geofencing). Formally,

Definition 5.1. *An invariant inv is a predicate (Boolean valued function) over a configuration or set of configurations such that, given a set of initial configurations of the system* $\mathcal{C}_0$*,*

$$\forall n \in \mathbb{N}, [\![\mathit{inv}]\!]_{\mathit{Reach}(\mathcal{C}_0, n)} \tag{5.6}$$

where $[\![\mathit{inv}]\!]_{\mathcal{C}}$ *represents the evaluation of inv over each configuration in* $\mathcal{C}$*. We use the notation* $[\![\mathit{inv}]\!]_{\boldsymbol{c}}$ *for evaluating inv over a single configuration* $\boldsymbol{c}$ *as well.*

We can also define a bounded invariant, or an n-invariant as follows:

Definition 5.2. *An n-invariant inv is a predicate over a configuration or a set of configuration such that given an $n \in \mathbb{N}$, $[\![inv]\!]_{Reach(\mathcal{C}_0,n)}$.*

Figure 5.2 shows the invariant specification syntax for *Koord* programs.

$$
\begin{array}{lcl}
\mathit{Term} & ::= & \mathit{Var} \mid \mathit{Val} \mid \mathit{CPorts} \\
 & \mid & \mathit{Term} + \mathit{Term} \;\mid\; \mathit{Term} \times \mathit{Term} \\
 & \mid & \mathit{Term} - \mathit{Term} \;\mid\; \mathit{Term}\ /\mathit{Term} \\
\mathit{BExpr} & ::= & \mathit{Term} \geq \mathit{Term} \;\mid\; \mathit{Term} \leq \mathit{Term} \\
 & \mid & \mathit{Term} = \mathit{Term} \;\mid\; \mathit{Term} > \mathit{Term} \;\mid\; \mathit{Term} < \mathit{Term} \\
 & \mid & \mathit{BExpr} \wedge \mathit{BExpr} \;\mid\; \mathit{BExpr} \vee \mathit{BExpr} \\
 & \mid & \neg\ \mathit{BExpr} \;\mid\; \mathit{BExpr} \Rightarrow \mathit{BExpr} \\
\mathit{inv} & ::= & \mathit{BExpr}
\end{array}
$$

Figure 5.2: Invariant specification syntax.

We define a function $eval : \mathbb{C} \times (\mathit{Var} \cup \mathit{CPorts}) \times (\mathtt{ID} \cup \{shared\}) \mapsto \mathit{Val}$, which takes as input a configuration, a variable from the set of all possible variables and sensor ports, and an indicator determining whether the variable to be evaluated is *shared* or local to a particular robot:

$$eval(c, x, shared) = c.S[x] \tag{5.7}$$

$$eval(c, x, i) = \begin{cases} c.L_i.M[x] & \text{if } x \in Keys(c.L_i.M) \\ c.L_i.cp[x], & \text{otherwise} \end{cases} \tag{5.8}$$

We formally define the property semantics of *Koord* using a function $\rightarrow_{sat}: \mathbb{C} \times \mathbb{P}rop \rightarrow Bool$ as shown in Figure 5.3. For properties involving only shared variables, the property semantics ensures that only the evaluation of the property on the global memory needs to be considered. Essentially, for properties with local variables or controller ports, the conjunction of the evaluation of the property on all the individual robot configurations needs to be considered.

5.2 VERIFICATION THROUGH BOUNDED MODEL CHECKING

Koord BMC is a tool for checking n-invariants for *Koord* applications using explicit state reachability analysis[2]. The input to *Koord* BMC is

(i) P: the program,

[2] The tool is available at https://github.com/ritwika314/koord.

$$(\boldsymbol{c}, \mathit{true}) \rightarrow_{sat} \mathit{true}$$

$$\bowtie \in \{==, \geq, \leq\} \qquad \frac{v \in \mathit{Val}, x \in \mathit{Keys}(\boldsymbol{c}.S)}{(\boldsymbol{c}, x \bowtie v) \rightarrow_{sat} (\boldsymbol{c}, \mathit{eval}(\boldsymbol{c}, x, \mathit{shared}) \bowtie v)}$$

$$\frac{v \in \mathit{Val}, x \notin \mathit{Keys}(\boldsymbol{c}.S)}{(\boldsymbol{c}, x \bowtie v) \rightarrow_{sat} (\boldsymbol{c}, \wedge_{i \in \mathtt{ID}}(\mathit{eval}(\boldsymbol{c}, x, i) \bowtie v))}$$

$$\frac{x_1 \in \mathit{Keys}(\boldsymbol{c}.S) \wedge x_2 \in \mathit{Keys}(\boldsymbol{c}.S)}{(\boldsymbol{c}, x_1 \bowtie x_2) \rightarrow_{sat} (\boldsymbol{c}, \mathit{eval}(\boldsymbol{c}, x_1, \mathit{shared}) \bowtie \mathit{eval}(\boldsymbol{c}, x_2, \mathit{shared}))}$$

$$\frac{x_1 \notin \mathit{Keys}(\boldsymbol{c}.S) \wedge x_2 \in \mathit{Keys}(\boldsymbol{c}.S)}{(\boldsymbol{c}, x_1 \bowtie x_2) \rightarrow_{sat} (\boldsymbol{c}, \wedge_{i \in \mathtt{ID}}(\mathit{eval}(\boldsymbol{c}, x_1, i) \bowtie \mathit{eval}(\boldsymbol{c}, x_2, \mathit{shared})))}$$

$$\frac{x_1 \in \mathit{Keys}(\boldsymbol{c}.S) \wedge x_2 \notin \mathit{Keys}(\boldsymbol{c}.S)}{(\boldsymbol{c}, x_1 \bowtie x_2) \rightarrow_{sat} (\boldsymbol{c}, \wedge_{i \in \mathtt{ID}}(\mathit{eval}(\boldsymbol{c}, x_1, \mathit{shared}) \bowtie \mathit{eval}(\boldsymbol{c}, x_2, i)))}$$

$$\frac{x_1 \notin \mathit{Keys}(\boldsymbol{c}.S) \wedge x_2 \notin \mathit{Keys}(\boldsymbol{c}.S)}{(\boldsymbol{c}, x_1 \bowtie x_2) \rightarrow_{sat} (\boldsymbol{c}, \wedge_{i \in \mathtt{ID}}(\mathit{eval}(\boldsymbol{c}, x_1, i) \bowtie \mathit{eval}(\boldsymbol{c}, x_2, i)))}$$

Figure 5.3: Property semantics for *Koord*.

(ii) *inv*: a candidate invariant function,

(iii) `ID`: set of robots,

(iv) δ: duration of the round and

(v) n: number of rounds to perform reachability analysis on.

The tool outputs ‘safe’ if the *inv* is indeed an n-invariant, ‘unsafe’ if it finds a counterexample execution of length n. In some cases, it can return ‘inconclusive’ as we will discuss presently. The *Koord* BMC tool uses the *Koord* executable semantics to produce all configurations that the system reaches during program transitions. It also uses the semantics along with implementations of low-level controllers to generate traces of the sensor ports of the system during the environment transitions of duration δ. These traces only include a sampling of values at the sensor ports and not *all* the intermediate values that the system reaches.

Algorithm 5.1 shows how *Koord* BMC computes reachable configurations and checks n-invariance. The set of permutations of the robots is denoted by *perms*. The subset of *perms*(ID) that may result in unique behaviors of a program P is denoted by $perms_P$(ID). We can compute this set by manually analyzing the shared variables written to during enabled events of each robot in a round of program transitions. The algorithm first checks that the invariant is valid on the set of initial states .

Algorithm 5.1: Bounded invariant checking algorithm.

```
Input: $P$, $inv$, ID, $\delta$, $n$
$C_0 \leftarrow Init(P, N)$   $p \leftarrow perms_P$(ID)
for $c$ in $C_0$ do
    if $Sat(c, \neg inv)$ then return 'unsafe';
end
$C \leftarrow C_0$
for $i = 0$ to $n$ do
    for $j = 0$ to $len(p)$ do
        $C' \leftarrow Post(C, p[j])$
        for $c$ in $C'$ do
            if $Sat(c, \neg inv)$ then return ('unsafe',c);
        end
        $C' \leftarrow End(C')$
        for $c$ in $C'$ do
            tr ← tr + BBTraces($c_{[0,\delta]}$)
        end
        $C \leftarrow C'_{env}$
    end
end
if TraceVerify(tr,inv) == 'unsafe' then return ('unsafe', tr);
else if TraceVerify(tr,inv) == 'inconclusive' then return 'inconclusive';
return 'safe'
```

Then the algorithm uses the *Koord* semantics to compute the set of all configurations reached from a set of configurations C (initially set to C_0) by the system during that round of program transitions. It does so by iterating over every order of program transitions in $perms_P$(ID). The algorithm checks *inv* is valid, or that the negation of *inv* is unsatisfiable, using the procedure *Sat*. If it finds an unsafe configuration c, then it returns 'unsafe' along with c. Otherwise, the samples of the system's trajectory from every configuration at the end of a round of program transitions are collected using the method *BBTraces*. These are then analyzed by an external verification tool (such as DryVR) in *TraceVerify*.

In the next iteration, the algorithm sets C to be the frontier set of configurations.

For *TraceVerify*, we used DryVR to compute over-approximations of the system's continuous trajectories. If it returns 'unsafe' along with a trace of the corresponding blackbox trace, *inv* is not an invariant of the system. DryVR may also return 'inconclusive', which means that the over-approximation computed is too coarse.

Lemma 5.1 summarizes the soundness of *Koord* BMC.

Lemma 5.1. *If Algorithm 5.1 returns 'unsafe' then there exists a counterexample to* inv *of length at most n. Assuming that the correctness of TraceVerify, if Algorithm 5.1 returns 'safe' then* inv *is an n-invariant.*

Proof. (*Sketch.*) If algorithm 5.1 returns *safe*, then for the n loop iterations of the outer loop, given a set of configurations C and the property `inv`, according to the algorithm $\forall c \in Post(C), c \models$ `inv`. Therefore, if the algorithm returns safe, then the set of configurations reached from C during a program transition is also safe. Finally, the algorithm invokes *TraceVerify* on all traces of the system collected using *BBTraces*. Since *TraceVerify* is assumed to be correct if it returns safe, then the reachable configurations are safe. QED.

5.3 BOUNDED MODEL CHECKING OF BENCHMARK APPLICATIONS

We briefly describe some benchmark applications implemented using *Koord*, their associated requirements, and present results on the verification of these applications using *Koord* BMC.

SATS This is a simplified version of distributed landing protocol [40, 58] originally designed for a small aircraft, but we apply it to drones. The protocol uses a shared list to sequence the drones approaching the strip, and a separation check to ensure that drones do not collide. The requirement is to ensure that a minimum separation between any two consecutive drones is maintained.

Fischer's Mutual Exclusion Protocol This is a well-known timing-based distributed mutual exclusion protocol, and it uses a shared variable [42] to decide which process executes its critical section. The desired invariant is that no two processes are in the critical section simultaneously.

HVAC This application models a room with a heater, which is on or off, based on readings from a thermostat. It is a popular benchmark in hybrid systems literature [29]. The thermal dynamics with the effect of the heater are modeled as the environment of the *Koord* program. The design should ensure that the temperature stays within a specified range.

Waypoints This application drives vehicles around obstacles to visit a sequence of waypoints in order. The property of interest is that no vehicles collide with any obstacles, or with each other.

Traffic In this program, the system handles robots navigating a traffic intersection without traffic lights, by communicating through shared variables. The 4-way intersection is divided into four areas, and a robot looking to make a turn at the intersection needs at most three of the said critical sections, and one going straight through needs at most two of them. We verify that no two robots visit the four areas of the intersections simultaneously and that the robots are contained in the areas during their navigation.

These applications cover a variety of real-world applications. For instance, Fischer's protocol is a standard timing-based mutual exclusion algorithm; Traffic uses shared variables for physical coordination. The SATS protocol we implemented is a simplified version of a real protocol developed by NASA.

All our results were obtained by executing *Koord* BMC on a machine with Intel Core i7-4960X CPU 3.60GHz and DDR3 RAM 64GB. We summarize our experiments with bounded model checking in Table 5.1.

Benchmark	**Robots**	δ (s)	**Rounds**	**Time to Verify** (s)	**Safe**
waypoint following	2	50	4	38.451	safe
waypoint following	3	50	4	59.934	safe
waypoint following	4	50	4	81.241	safe
HVAC	1	40	10	26.239	safe
lineform	3	10	4	20.183	safe
lineform	4	10	4	24.886	safe
lineform	5	10	4	28.113	safe
fischer's protocol	2	1	2	479.931	safe
SATS landing protocol	2	50	4	32.496	safe
SATS landing protocol	3	50	4	38.382	safe
traffic	2	3	5	40.493	safe
traffic	3	3	5	131.423	safe

Table 5.1: Benchmark Summary

The *Robots* column in the table refers to the number of robots we performed the experiments with, δ is the time increment during controller transitions, the *rounds* column is the number of rounds. The time taken to verify these applications increases exponentially with the number of robots, as well as the number of rounds.

5.3.1 Choosing δ

If δ is too large, then there are less updates to actuator ports which may result in unsafe behaviors due the controller running unchecked for too long. For instance, if the in the *HVAC* example, we do not check the temperature values with sufficient frequency, the heaters may remain on even when temperature has been more than the desired maximum for a long time. If δ is too small, the executable semantics potentially produces a huge number of behaviors, and it is difficult to analyze the program for a non-trivial program execution time. In essence, the choice of δ also depends on the environment physics, and it is usually chosen experimentally. Table 5.2 shows the variation of verification results for different values of δ for the *HVAC* benchmark. We notice that for large values of δ, the system doesn't satisfy the property that the temperature stays within an allowed range. This is expected as if the switch is on for too long then the temperature may exceed the allowed range, and if it is off for too long then the temperature may become too low. This set of experiments shows how the choice of δ can be made for the HVAC benchmark.

δ (s)	**Rounds**	**Time to Verify** (s)	**Safe**
40	10	26.2	safe
50	10	21.9	safe
75	8	26.2	safe
100	6	23.2	unsafe
125	6	26.1	unsafe

Table 5.2: δ variation in HVAC

While our Lemma 5.1 shows that our bounded verification algorithm is sound, it is not complete. Additionally, even though it can be used to verify applications with a larger number of robots, the time taken is prohibitive for experiments due to the state-space explosion caused by the distributed, nondeterministic nature of *Koord* applications. In the next section, we show how we explore scalable verification of *inductive* invariants.

5.4 DECOMPOSING INDUCTIVE INVARIANCE VERIFICATION

We first define the notion of inductive invariants formally.

Definition 5.3. *A predicate inv is an* inductive invariant *of a system if given a set of initial configurations of the system $\mathcal{C}_0$, the following proof obligations (POs) hold:*

$$[\![inv]\!]_{\mathcal{C}_0} \tag{5.9}$$

$$[\![inv]\!]_{\mathcal{C}} \Rightarrow [\![inv]\!]_{Reach(\mathcal{C},1)} \tag{5.10}$$

That is, *inv* holds in the initial configuration(s) (PO (5.9)), and *inv* is preserved by both platform-independent program transitions (distributed program logic) and the platform-dependent environment transitions (controllers). according to PO (5.10). It is straightforward to prove that an inductive invariant is an invariant of the system.

Our verification strategy for programmer-specified (inductive) invariants is to discharge the proof obligations. PO (5.9) is usually trivial. Therefore, we focus on PO (5.10). The *Koord* semantics enables us to *decouple* the environment and program transitions in *Reach*, and analyze each separately. PO (5.10) can be restated as

$$[\![inv]\!]_{\mathcal{C}} \Rightarrow [\![inv]\!]_{Post(\mathcal{C})} \tag{5.11}$$

$$[\![inv]\!]_{\mathcal{C}} \Rightarrow [\![inv]\!]_{End(\mathcal{C})_{[0,\delta]}} \tag{5.12}$$

As in other concurrent systems, a major bottleneck in computing $Post(\mathcal{C})$ for PO (5.11) is the required enumeration of all $\vec{p} \in perms(\texttt{ID})$ permutations for all robots with reads/writes to the global memory. We, therefore, seek a stronger and easier to prove proof obligation.

Lemma 5.2. *Given a predicate φ and a configuration c, if $[\![\varphi]\!]_c \Rightarrow \bigwedge_{i \in ID} \bigwedge_{e \in Events} [\![\varphi]\!]_{Post(c,i,e)}$, the following always holds:*

$$[\![\varphi]\!]_c \Rightarrow [\![\varphi]\!]_{Post(c)} \tag{5.13}$$

Proof. Consider an execution in which the robots execute their events in the order $\vec{p} = p_1, p_2, \ldots p_{\texttt{N\_sys}}$. We know that $Post(c, \vec{p}) = Post((Post(c, p_1), (p_2, \ldots, p_{\texttt{N\_sys}}))$, since $\vec{p}$ is not an empty sequence. From our assumption, we get that

$$\bigwedge_{e \in Events} [\![\varphi]\!]_{Post(c,p_1,e)} \tag{5.14}$$

Using 5.14 and the definition of $Post(c, p_1)$, we get that $[\![\varphi]\!]_{Post(c,p_1)}$. A similar argument can be used to derive that $[\![\varphi]\!]_{Post(c,p_i)}$ for any $p_i \in \vec{p}$. Since $[\![\varphi]\!]_{Post(c,p_1)}$, $[\![\varphi]\!]_{Post(c',p_2)}$, where $c' \in Post(c, p_1)$. In fact, for robots with pids p_i, p_{i+1} in $\vec{p}$ executing their events consecutively from a configuration c,

$$[\![\varphi]\!]_{Post(c,p_i)} \Rightarrow [\![\varphi]\!]_{Post(Post(c,p_i),p_{i+1})} \tag{5.15}$$

Given 5.15 and the definition of $Post(c, \vec{p})$, we can conclude that:

$$[\![\varphi]\!]_c \Rightarrow [\![\varphi]\!]_{Post(c,\vec{p})} \tag{5.16}$$

Further, since we proved the 5.16 for an arbitrary permutation $\vec{p}$, we can conclude that it is true for every permutation, i.e , $\bigwedge_{\vec{p} \in perms(\texttt{ID})} [\![\varphi]\!]_{Post(c,\vec{p})}$. Hence, $[\![\varphi]\!]_c \Rightarrow [\![\varphi]\!]_{Post(c}$ QED.

Essentially this lemma states that as φ is preserved before and after every event transition of every robot, the order of robot event execution does not impact the validity of φ.

With Lemma 5.2, we strengthen and rewrite PO (5.11) as

$$[\![inv]\!]_{\mathcal{C}} \Rightarrow \bigwedge_{i \in \mathtt{ID}} \bigwedge_{e \in Events} [\![inv]\!]_{Post(\mathcal{C},i,e)} \tag{5.17}$$

which no longer requires enumeration of all permutations. We use this lemma for scalable verification of *Koord* applications in our synchronous round-based model of execution.

We now discuss our approach to discharge PO (5.12). To further decouple program and environment transitions, we rewrite PO (5.12) by expanding $[\![inv]\!]_{End(\mathcal{C})_{[0,\delta]}}$ and derive:

$$[\![inv]\!]_{\mathcal{C}} \Rightarrow (\forall \boldsymbol{c}', \boldsymbol{c}'', \forall t \in [0,\delta], \boldsymbol{c}' \in End(\mathcal{C})) \wedge \boldsymbol{c}'' = traj(\boldsymbol{c}', t) \Rightarrow [\![inv]\!]_{\boldsymbol{c}''}). \tag{5.18}$$

PO (5.18) requires reasoning about the dynamic behavior of *traj* during environment transitions, and it is a challenging research problem by itself.

5.5 CAPTURING CONSTRAINTS ON CONTROLLER BEHAVIOR

We introduce *controller assumption* to abstract away the continuous dynamic behavior by controllers (including sensor and actuator ports) during environment transitions.

Definition 5.4. *A* controller assumption *is a pair of predicates $\langle P, Q\rangle$, where P is defined over $CPorts \times Val \times CPorts \times Val$ and Q is over $CPorts \times Val$. Given a* controller assumption *$\langle P, Q\rangle$, the traj function satisfies the assumption if starting from any $\boldsymbol{c}'$ with port values satisfying P then any reachable configuration $\boldsymbol{c}''$ within $[0,\delta]$ must satisfies Q. Formally,*

$$\forall \boldsymbol{c}', \boldsymbol{c}'', \forall t \in [0,\delta], P(\boldsymbol{c}'.Acts, \boldsymbol{c}'.Sens) \wedge \boldsymbol{c}'' = traj(\boldsymbol{c}', t) \Rightarrow Q(\boldsymbol{c}''.Sens) \tag{5.19}$$

where $\boldsymbol{c}'.Acts$ refers to its actuator port values, $\boldsymbol{c}'.Sens$ refers to the sensor port values, A controller assumption $\langle P, Q\rangle$ is similar to preconditions and postconditions for the *traj* function with an additional guarantee that Q must hold at all time during the time horizon $[0,\delta]$. It allows programmers to over-approximate the set of all transient configurations reached by *traj* and prove the invariant. We demonstrate in Chapter 6 and Chapter 7 how controller assumption can be validated with specialized tools for continuous dynamics.

We know by definition $End(\mathcal{C}) \subseteq Post(\mathcal{C})$. With Lemma 5.2, we can merge PO (5.17) and PO (5.18), add program and controller assumptions, and simplified our proof obligation as:

$$\bigwedge_{i \in \mathtt{ID}} \bigwedge_{e \in Events} [\![inv]\!]_{\mathcal{C}} \wedge \boldsymbol{c}' \in \mathcal{C}_i^e \wedge (P(\boldsymbol{c}'.Acts, \boldsymbol{c}'.Sens) \Rightarrow Q(\boldsymbol{c}''.Sens)) \Rightarrow [\![inv]\!]_{\boldsymbol{c}''} \tag{5.20}$$

Notice the continuous dynamics no longer appear in PO (5.20), and it allows us to reason in per event fashion as well as per robot fashion. We then can use our $\mathbb{K}$ symbolic execution semantics to construct the symbolic post configurations $\mathcal{C}_i^e$ for each event e, and prove the validity with SMT solvers.

5.6 DEALING WITH FUNCTIONS AND LOOPS

Many programs can include both module related functions and loops. Proving invariants over such loops is by itself a well studied and difficult research problem.

To mitigate this problem, we instruct our symbolic execution to treat module related functions such as computing distance between two points, and paths generated by path planners as *uninterpreted functions*, and we introduce *function summary* for such uninterpreted functions similar to controller assumptions.

Definition 5.5. *A* function summary $F(x, y)$ *for an uninterpreted function* $f(x)$ *is a predicate which we can prove the following proof obligation:*

$$\forall x, F(x, f(x)) \tag{5.21}$$

where x can be extended according to the arity of f. Verification and generation of good function summaries is extensively discussed and widely used in software verification. We believe writing a good function summary requires strong domain knowledge in particular robot devices and the problem to be solved.

Both controller assumptions and function summaries require proofs for end-to-end verification of inductive invariants for *Koord* programs. The *Koord* Prover we have built only deals with PO (5.20)

5.7 SYMBOLIC EXECUTION FOR PROVING INDUCTIVE INVARIANTS

We reason about the system in terms of symbolic *configurations*, which is the symbolic representation of a system during a given program or environment transition. *Koord* Prover first computes the initial symbolic configuration and checks that the invariant holds in that configuration. Then our symbolic execution constructs the program transition constraints and the environment transition constraints. It then generates such constraints for every robot *pid* and checks their validity using the Z3Py file generated by the $\mathbb{K}$-Z3 interface. It returns 'safe' if the *negation* of the conjunction of all the generated constraints is unsatisfiable. It returns 'unsafe' along with a counterexample model otherwise.

We implemented the symbolic expression semantics in the $\mathbb{K}$ rewriting system (*Symbolic Post Generation*) and used it to compute the (symbolic) reachable sets of configurations for every event e in a given program in *Koord* Prover[3].. We used bookkeeping to collect the constraints generated by the candidate invariants as well as conditional statements along with their relevant memory mappings, as part of the configuration in $\mathbb{K}$. For instance, suppose there is a statement : `if (Cond) : Ss else : Ss2` where `Ss` and `Ss2` are statement blocks. Suppose further that when *Cond* is processed by robot i, the configuration of the system is C_1, the configuration after Ss is executed is C_2, and the configuration after Ss_2 is executed is C_3. The constraints generated for the candidate invariant I are $(\llbracket Cond \rrbracket_{C_1} \wedge \llbracket I \rrbracket_{C_2}) \vee not(\llbracket Cond \rrbracket_{C_1}) \wedge \llbracket I \rrbracket_{C_3}$. C_1, C_2 and C_3 possibly differ only in terms of the local memory of robot i. $\mathbb{K}$ is specially useful for generating such *context-sensitive* constraints. These constraints were then parsed using the $\mathbb{K}$-Z3 interface: a parser we implemented using Python lex-yacc (ply). We used it to generate a python file that contained a Z3 solver and added the system constraints to it.

5.8 SUMMARY

In this chapter, we defined the notion of reachable configurations for *Koord* programs. We then discussed our explicit bounded model checking approach by computing the set of reachable configurations using the executable *Koord* semantics and presented a verification summary for benchmark *Koord* applications.

We then introduced a decoupled analysis technique for inductive invariants, which separates the discrete distributed coordination and the continuous control components of a *Koord* program. We presented a lemma (Lemma 5.2), which enables this decoupled analysis to scale. We will show in Chapters 6, 7 and 8 how this decoupled analysis is applied to three benchmark applications.

[3] Koord Prover is available at https://github.com/ritwika314/koordprover

CHAPTER 6: CASE STUDY: DISTRIBUTED FORMATION CONTROL

In this chapter, we revisit the LineForm program of Chapter 2 and demonstrate the decomposed verification approach for inductive invariants we motivated in Chapter 5. Recall that the application (Figure 2.2) forms an equispaced line of robots between two fixed endpoints. The robots executing the application are required to satisfy a geofencing requirement, which we will state as an invariant. We demonstrate how our decomposed verification enables plugging in separate analyses of the distributed coordination logic that determines the formation algorithm and the low-level control that needs to obey the geofencing requirement of the LineForm application.

6.1 FORMAL MODELING AND ANALYSIS

As mentioned in Chapter 5, the symbolic post configuration generated by $\mathbb{K}$ is used to represent a set of *system configurations*. For a variable x in $\boldsymbol{c}$, x' represents its corresponding value in $\boldsymbol{c}' \in \mathit{Post}(c)$; x'' represents its corresponding value in $\boldsymbol{c}'' \in (\mathit{End}\mathcal{C})_{[0,\delta]}$.

Consider a candidate invariant:

Invariant 6.1. *:*

$$[\![\mathit{Geofencing}_i]\!]_{\boldsymbol{c}} := \texttt{Motion.psn}_i \in \mathit{rect}(\mathsf{p}_{min}, \mathsf{p}_{max}) \wedge x[i] \in \mathit{rect}(\mathsf{p}_{min}, \mathsf{p}_{max}) \tag{6.1}$$

This invariant asserts that the position of each robot i is always within $\mathit{rect}(\mathsf{p}_{min}, \mathsf{p}_{max})$, and that each agent updates its shared variable value to one within $\mathit{rect}(\mathsf{p}_{min}, \mathsf{p}_{max})$ as well.

We first try to prove Invariant 6.1 without any assumptions, only from the constraints generated through the symbolic execution of LineForm. *Koord* Prover symbolically executes the event TargetUpdate (for robot i) and automatically generates the post event configuration constraint $\mathit{TargetUpdate}_i$:

$$\mathit{TargetUpdate}_i := \left(\begin{array}{l} \neg(i = \texttt{N\_sys} - 1 \vee i = 0) \\ \wedge\ \texttt{Motion.target}'_i = (\texttt{x[i} - \texttt{1]} + \texttt{x[i} + \texttt{1]})/2 \\ \wedge \texttt{x}'\texttt{[i]} = \texttt{Motion.psn}_i \wedge\ \ u\_vars \\ \wedge \texttt{Motion.psn}''_i := \mathit{traj}(\texttt{Motion.psn}'_i, \texttt{Motion.target}_i, t) \\ \wedge t \in [0, \delta] \end{array} \right) \tag{6.2}$$

where *traj* is treated as an uninterpreted function over $\mathbb{R} \times \mathbb{R}$. The function *rect* can both be precisely defined as well as left uninterpreted. Recall that the primed copies of the variables in $\boldsymbol{c}$ are their values in $\boldsymbol{c}'$, and the double primed copies are their values in $\boldsymbol{c}''$. The rest of the formula includes a subformula u_vars that ensures the values of unmodified variables are unchanged; for instance, $\texttt{Motion.psn}'_i = \texttt{Motion.psn}_i$ and $x'[j] = x[j]$ for $j \neq i$.

Since there is only one event, the induction proof obligation, the *Koord* Prover generates the following proof obligation PO (6.1) for LineForm:

Proof Obligation 6.1.

$$\bigwedge_{i \in ID} [\![\mathit{Geofencing}_i]\!]_{\boldsymbol{c}} \wedge \mathit{TargetUpdate}_i \Rightarrow [\![\mathit{Geofencing}_i]\!]_{\boldsymbol{c''}} \tag{6.3}$$

The *Koord* Prover returns that the negation of PO (6.1) is satisfiable, meaning that either our proposed invariant cannot be proved without any additional constraints, or it doesn't hold. The satisfying assignment serves as a counterexample. Since automatically generated proof obligations do not include any sensor or actuator assumptions, it is not surprising that the *Koord* Prover could not prove the invariant. Specifically, PO (6.1) does not include any restrictions on $\texttt{Motion.psn}''_i$, $\texttt{Motion.target}''_i$ w.r.t any of the variables in the symbolic post configuration after the `TargetUpdate` event.

Next, we introduce a controller assumption $\langle P_i, Q_i \rangle$:

$$\begin{aligned} P_i &:= \texttt{Motion.psn}'_i \in \mathit{rect}(\mathsf{a}, \mathsf{b}) \wedge \texttt{Motion.target}'_i \in \mathit{rect}(\mathsf{a}, \mathsf{b}) \\ Q_i &:= \texttt{Motion.psn}''_i \in \mathit{rect}(\mathsf{a}, \mathsf{b}), \end{aligned} \tag{6.4}$$

where a and b are constants in $\mathbb{R}^3$, c' is the configuration P_i is evaluated on, and c'' is the configuration Q_i is evaluated on. PO (6.1) is then refined to PO (6.2):

Proof Obligation 6.2.

$$\bigwedge_{i \in ID} [\![\mathit{Geofencing}_i]\!]_{\boldsymbol{c}} \wedge \mathit{TargetUpdate}_i \wedge (P_i \Rightarrow Q_i) \Rightarrow [\![\mathit{Geofencing}_i]\!]_{\boldsymbol{c''}} \tag{6.5}$$

Having added the controller assumption (6.4), the *Koord* Prover returns that the negation of PO (6.2) is unsatisfiable, i.e., (6.4) is sufficient to prove Invariant 6.1 . Table 6.1 summarizes the verification of PO (6.1) on instances of LineForm with different `N_sys`.

6.2 VALIDATING DYNAMIC BEHAVIOR

We now turn towards validating the controller assumption (Controller Assumption 6.4). Recall, from PO (5.19) defined in Chapter 5 and $\langle P_i, Q_i \rangle$ defined above, we can derive the following:

N_sys	**dim**	T_K (s)	T_V (s)	**Safe**
3	1	4.90	9.09	safe
3	2	4.19	10.13	safe
4	1	4.79	12.21	safe
4	2	5.28	12.49	safe
4	3	5.06	12.77	safe
5	1	4.91	18.46	safe
5	2	5.60	18.91	safe
5	3	4.33	20.30	safe
10	1	4.92	32.34	safe
10	2	5.16	32.42	safe
10	3	4.34	33.61	safe
15	1	5.23	53.89	safe

Table 6.1: Summary of semantics based verification for LineForm. T_K is the symbolic post computation time in $\mathbb{K}$, T_V is the time taken for construction of constraints and verification in Z3. Robots moving along a line are represented by **dim** $= 1$, along a plane by **dim** $= 2$, and in a 3D space by **dim** $= 3$.

Controller Proof Obligation 6.1.

$$\begin{aligned} \forall t \in [0, \delta], \texttt{Motion.psn}_i \in \mathit{rect}(\mathsf{a}, \mathsf{b}) \wedge \texttt{Motion.target}_i \in \mathit{rect}(\mathsf{a}, \mathsf{b}) \\ \wedge c'' = \mathit{traj}(c', t) \Rightarrow \texttt{Motion.psn}''_i \in \mathit{rect}(\mathsf{a}, \mathsf{b}) \end{aligned} \tag{6.6}$$

It states that if the current position and the target of the robot are within the rectangle $\mathit{rect}(\mathsf{a}, \mathsf{b})$, then it remains within the rectangle for the next δ interval.

To prove (6.4), one has to use the dynamics of the specific robot and the specifics of the waypoint-tracking controller driving the vehicles running the *Koord* application. Several approaches are available for checking this type of properties for control systems, such as, barrier certificates [67], reachability analysis [5, 46], and Lyapunov analysis [16, 25]. For each of these approaches, there are many available algorithms and tools. For our verification of LineForm, we use the reachability analysis approach, and in the remainder of this section, we give an overview of this analysis.

Reachability analysis computes the set of states of a control system that is reachable from a set of initial states. Proving Controller Proof Obligation 6.1 boils down to computing the set of reachable states from a set of initial positions bounded by $\mathit{rect}(\mathsf{a}, \mathsf{b})$ and with the target also in the same rectagle, and checking that the result is contained in $\mathit{rect}(\mathsf{a}, \mathsf{b})$. Typically, computing the exact set of reachable states is undecidable for nonlinear control system models, and therefore, the available algorithms rely on over-approximations

In this case study, we use the DryVR [28] reachability analysis tool, which uses numerical simulations to learn the sensitivity of the robot's trajectories. It then uses this sensitivity and additional simulations to either prove the required property (with a probabilistic guarantee), or it finds a counterexample trace. DryVR has been used to analyze automotive and aerospace control systems [27]. Here we use the *Koord* simulator to generate traces of a quadcopter using the Hector quadrotor model [56], from which DryVR computes the reachsets.

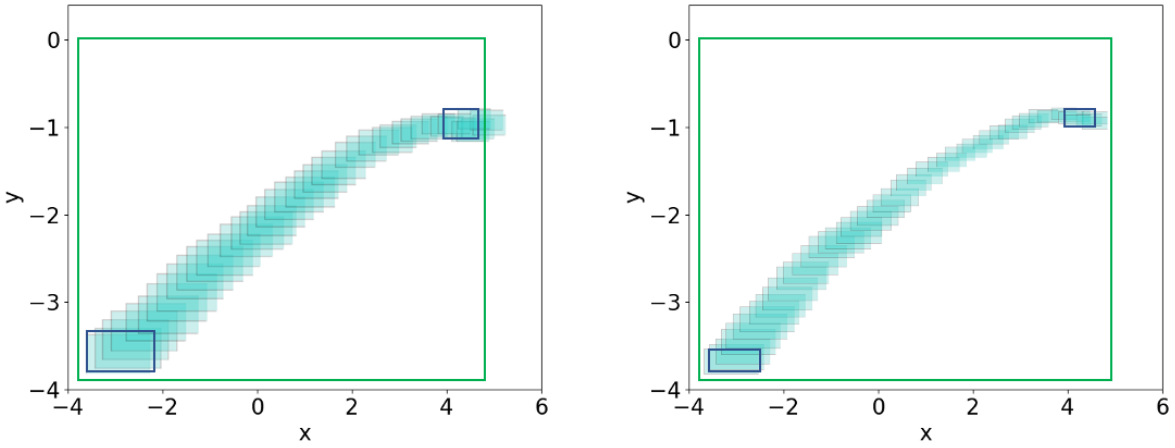

Figure 6.1: Reachtube computations for lineform, for the quadcopter.

Figure 6.1 shows the outputs of the reachability analysis performed on quadcopters models. Here we have computed reach sets from a smaller initial rectangle and with a target that is also in a smaller rectangle than $rect(\mathsf{a}, \mathsf{b})$. In each of the plots, the green rectangle represents $rect(\mathsf{a}, \mathsf{b})$. The blue rectangle at the bottom left corner of each plot represents starting points in the simulated trajectories used to generate these reachtubes. The blue rectangle on the top right corner is the bound on the targets reached in the trajectories.Figure 6.1 (*Left*) shows that the reachtube of the quadcopter using a simple PID controller overshoots its target, and violates the Controller Proof Obligation 6.1. Figure 6.1 (*Right*) shows that, for a quadcopter with the same controller with different control gains with a lower settling time, the controller assumption is satisfied.

However, we note that the model of the quadcopter is symmetric under translations, planar reflections, and rotations. Therefore, using Theorem 10 from [71] and as shown in [72], the computed reachsets can be translated and rotated to cover all initial and target choices in $rect(\mathsf{a}, \mathsf{b})$.

6.3 SUMMARY

We demonstrated the decomposed verification approach for inductive invariants for *Koord* programs on a formation control application, LineForm. We showed how the *Koord* Prover automatically generates a basic proof obligation from the proposed inductive invariant, and symbolic execution of the *Koord* program. We then showed how we use controller assumptions on the robots executing the program to aid in verifying the inductive invariant. Finally, we showed how the controller assumptions themselves are proof obligations that can be validated through DryVR. While we didn't have to use the notion of function summaries in this chapter, we will show how they can become useful in the next chapter for the verification of a distributed delivery application.

CHAPTER 7: CASE STUDY: DISTRIBUTED DELIVERY

Many distributed multi-robot applications can be seen as distributed task allocation problems, with different points in a shared environment that robots collaboratively visit. We view visiting points as an abstraction for location-based objectives like package delivery, mapping, surveillance, or fire-fighting. In this chapter, we discuss a *Koord* application that performs distributed delivery. We then show how our decomposed verification approach can verify the safety requirements for this application.

7.1 PROBLEM SETUP

We first define the *distributed delivery* problem as follows: Given a set of (possibly heterogeneous) robots, a safety distance $\epsilon > 0$, and a fixed sequence of delivery points (or tasks) $\mathit{list} = x_1, x_2, \ldots \in \mathbb{R}^3$, there are following two requirements: (a) every unvisited x_i in the sequence is *visited* precisely by one robot and (b) no two robots ever get closer than ϵ.

The flowchart in Figure 7.1 shows a simple idea for solving this problem from the perspective of a single robot: Robot A looks for an unassigned task τ from *list*; if there is a clear path to τ, then A assigns itself the task τ. Then A visits τ following the path; once done, it repeats.

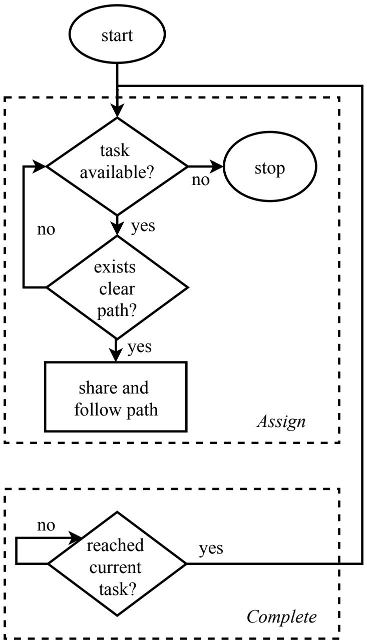

Figure 7.1: The flowchart for a simple solution to Delivery application.

Converting this to a working solution for a distributed system is challenging as it involves combining distributed mutual exclusion ([53, 33]) to assign a task τ exclusively to a robot

```
using Motion:
  actuators:
    List⟨Point⟩ path
  sensors:
    Point position
    bool reached
    PathPlanner planner
local:
  bool on_task = s
  List⟨Point⟩ curr_path
  Task cur_task

allread:
  List⟨Point⟩ shared_paths[N_sys]
allwrite:
  List⟨Task⟩ all_tasks

Complete:
pre: on_task and Motion.reached
eff: on_task=False
shared_paths[pid]=[Motion.position]

Assign:
  pre: !on_task
  eff:
    if len(all_tasks) == 0:
      stop
    else: atomic:
      for t in all_tasks:
        curr_path=Motion.planner(t.target)
        if pathIsClear(shared_paths, \
                    curr_path, pid):
          on_task=True
          cur_task=t
          break
      if on_task:
        all_tasks.remove(cur_task)
        shared_paths[pid]=curr_path
        Motion.path=curr_path
      else:
        shared_paths[pid]=[Motion.position]
```

Figure 7.2: *Koord* code for distributed Delivery application.

A from the *list* (step 1), dynamic conflict-free path planning (step 2), and low-level motion control (step 3).

Figure 7.2 shows our *Koord* language implementation of Delivery, which intuitively follows this flowchart.

7.2 FORMAL MODELING AND ANALYSIS

In this case study, we only prove requirement (b) for Delivery. Delivery consists of two events

(i) `Assign`, in which each robot looks for an unassigned task `t` from `all_tasks`; if there is a clear path to `t` then the robot assigns itself the task `t`, set the actuator port `Motion.path`, and shares its path with all other robots through `shared_path`. Otherwise, it shares its position as the path. The `atomic` keyword provides mutual exclusion to ensure that no two robots can pick the same task or pick conflicting paths to tasks.

(ii) `Complete`, which checks whether a robot has visited its assigned task.

A path here is a list of points that a robot visits in sequence. The `Motion` module drives the robot along a path, as directed by the position value set at its actuator port `Motion.path`. The sensor port `Motion.planner` returns a path to the target of an unassigned task, and a (user-defined) function called `pathIsClear` is used to determine whether the currently planned path is within ϵ distance of any path in `shared_path`.

7.3 PROOF SETUP

Suppose there is a function parameterized by ϵ, taking two paths as input $\mathit{clear}_\epsilon : \mathit{List}\langle \mathit{Point}\rangle \times \mathit{List}\langle \mathit{Point}\rangle \mapsto \mathit{bool}$, it returns true only if the minimum distance between the two paths is greater than ϵ. We restate requirement (b) as: no two robots ever get closer than ϵ. This gives us the following invariant for the current configuration c:

Invariant 7.1.

$$[\![\mathit{SafeDistance}]\!]_{\boldsymbol{c}} = \forall j \in [\texttt{N\_sys}], (i \neq j \wedge \mathit{clear}_\epsilon(\texttt{shared\_path}[i], \texttt{shared\_path}[j])) \vee (i = j) \tag{7.1}$$

Computing the *clear* function involves nested loops over the length of each path, then computing the minimum distance between each path segment `pathIsClear` further has to iterate over all shared paths and check whether they satisfy the requirements on *clear*. We use the notion of *function summary* to capture the notion of correctness for for `pathIsClear`. The function summary *PIC* is defined below as:

Function Summary 7.1.

$$\mathit{PIC}(sp, cp, i, y) := \forall j \in \boldsymbol{\mathit{ID}}, j \neq i \wedge \neg \mathit{clear}_\rho(sp[j], cp) \Rightarrow \neg y \tag{7.2}$$

where $0 \leq \rho < \epsilon$. The function summary simply says, if a robot's current path cp is not more than ϵ distance to any path $sp[j]$ shared by other robots, the output of $y =$ $\texttt{pathIsClear}(sp, cp, i)$ should be false,[1] and PO (5.21) becomes:

Proof Obligation 7.1.

$$\forall sp, cp, i, \mathit{PIC}(sp, cp, i, \texttt{pathIsClear}(sp, cp, i)) \tag{7.3}$$

[1] The index i in the `pathIsClear` function indicates that the previously shared path of the robot i shouldn't be considered in the computation.

Validating PO (7.1) requires us to reason about the implementation of the `pathIsClear` function, which is beyond the scope of this case study.

To construct the symbolic set of configurations, we use a list with four tasks signified by $\{t_1, t_2, t_3, t_4\}$ so that the symbolic execution terminates. The for-loop iterating through the task list is unrolled into a sequence of (nested) *if-else* statements. The *automatically generated* symbolic post event configuration of the *Assign* event for an execution when robot i picks t_1 is as follows:

$$\begin{aligned} E_i^{t_1} :=\ & \neg \texttt{on\_task}_i \wedge \texttt{on\_task}'_i \\ & \wedge \texttt{curr\_path}'_i = \texttt{Motion.planner}(\texttt{t}_1\texttt{.target}) \\ & \wedge \mathit{PIC}(\texttt{shared\_path}, \texttt{curr\_path}'_i, i, \texttt{True}) \\ & \wedge \texttt{shared\_path}'[i] = \texttt{curr\_path}_\texttt{i}{}' \\ & \wedge \texttt{Motion.path}'_i = \texttt{shared\_path}'[i] \wedge \mathit{u\_vars} \end{aligned} \tag{7.4}$$

where *u_vars* again, ensures the values of unmodified variables are unchanged. Notice how we can use *PIC* to summarize `pathIsClear`. Similarly, we get $E_i^{t_2}$, $E_i^{t_3}$ and $E_i^{t_4}$ for other execution paths choosing corresponding tasks. When none of the tasks is picked, the post event configuration generated is

$$E_i^{none} := \neg \texttt{on\_task}_i \wedge \texttt{shared\_path}'[i] = [\texttt{Motion.pos}_i] \wedge \mathit{u\_vars} \tag{7.5}$$

For the event *Assign*, the post event configuration is:

$$\mathit{Assign}_i := \left(\begin{array}{l} \forall j \in [\texttt{N\_sys}], E_i^{t_1} \wedge E_i^{t_2} \wedge E_i^{t_3} \wedge E_i^{t_4} \wedge E_i^{none} \\ \wedge(\texttt{Motion.pos}'', \texttt{Motion.reached}'') = \\ \mathit{traj}(\texttt{Motion.pos}', \texttt{Motion.reached}', \texttt{Motion.path}', t) \wedge t \in [0, \delta] \end{array} \right) \tag{7.6}$$

The *Koord* Prover then automatically generates the proof obligation :

Proof Obligation 7.2.

$$\bigwedge_{i \in ID} [\![\mathit{SafeDistance}_i]\!]_{\mathbf{c}} \wedge \mathit{Assign}_i \Rightarrow [\![\mathit{SafeDistance}_i]\!]_{\mathbf{c}''} \tag{7.7}$$

Without any assumptions on robot motion, our tool is unable to discharge this proof automatically. For abstracting the movement of robots, a robot i should move closely ($\neg \mathit{clear}_\beta$, where $2\beta + \rho \leq \epsilon$) along its `Motion.path` actuator whose value is denoted by $\texttt{Motion.path}''_i$

Benchmark	`N_sys`	T_K (s)	T_V (s)	**Safe**
Task	3	9.90	10.6	safe
Task	4	9.79	11.78	safe
Task	5	9.91	14.92	safe
Task	10	12.92	18.34	safe

Table 7.1: Summary of semantics based verification for Delivery. T_K is the symbolic post computation time in $\mathbb{K}$, T_V is the time taken for generation of constraints and verification in Z3 and `N_sys` is the number of robots in the systeMotion.

until it finishes traversing the path. We add $\langle P_i, Q_i \rangle$ with

$$\begin{aligned} P_i &:= \neg\texttt{Motion.reached}'_i \\ Q_i &:= \neg clear_\beta(\texttt{Motion.pos}''_i, \texttt{Motion.path}''_i) \end{aligned} \tag{7.8}$$

PO (7.2) is then refined to PO (7.3):

Proof Obligation 7.3.

$$\bigwedge_{i \in ID} [\![SafeDistance_i]\!]_{\boldsymbol{c}} \wedge Assign_i \wedge (P_i \Rightarrow Q_i) \Rightarrow [\![SafeDistance_i]\!]_{\boldsymbol{c}''} \tag{7.9}$$

The proof obligation for the induction hypothesis for the event `Complete` is generated similarly (omitted here), and the overall proof obligation is a conjunction of the two. Table 7.1 summarizes the verification of these constraints with different numbers of robots.

7.4 VALIDATING DYNAMIC BEHAVIOR

From PO (5.19) defined in Chapter 5 and $\langle P_i, Q_i \rangle$ defined above, we can derive the corresponding proof obligation for controller assumption (Controller Assumption 7.8):

Controller Proof Obligation 7.1.

$$\begin{aligned} \forall t \in [0, \delta], &\neg\texttt{Motion.reached}'_i \wedge c'' = traj(c', t) \\ &\Rightarrow \neg clear_\beta(\texttt{Motion.pos}''_i, \texttt{Motion.path}''_i) \end{aligned} \tag{7.10}$$

We now turn towards DryVR based validation for Controller Proof Obligation 7.1. We computed reachtubes for our vehicle models and checked whether they were contained within β distance of the desired path. In both the plots shown in Figure 7.3, the grey shaded area is *unsafe* and needs to be avoided. The blue path is the computed path, and the green

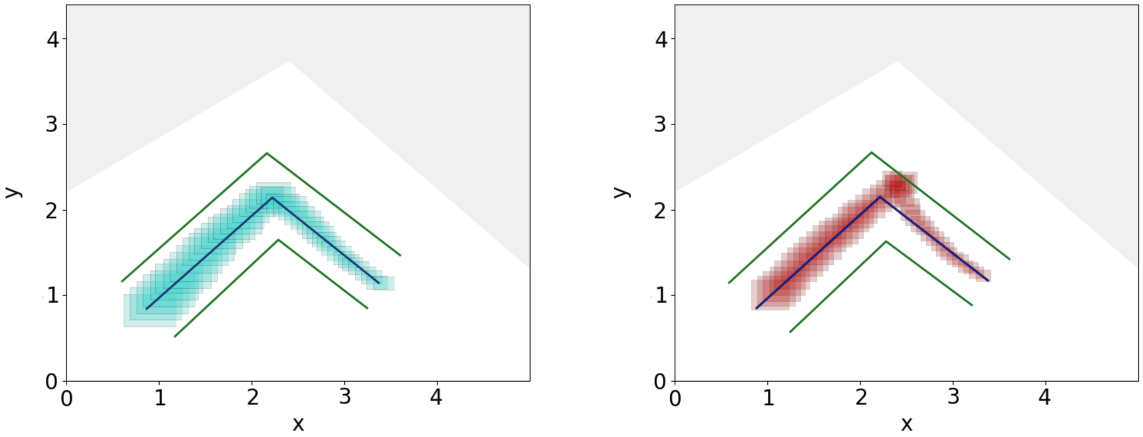

Figure 7.3: Reachtube computations for Delivery.

lines indicate the bounds at β distance from the path. Figure 7.3 (*Left*) shows the computed reachtube for the quadcopter lies within β of the actual path. Thus the vehicle will not violate Controller Proof Obligation 7.1. Figure 7.3 (*Right*) shows the computed reachtube for our car model is not contained within β distance of the computed path. Therefore, the vehicle may violate the assumption. The car-model [41] we used has non-holonomic constraints, and making the turn formed by the two components of the path shown in Figure 7.3 requires the car to perform a reverse maneuver that may violate the safety constraint.

7.5 SUMMARY

In this chapter, we demonstrated the decomposed verification approach for inductive invariants for *Koord* programs on a distributed delivery application. We showed how we use function summaries to express the correctness requirements on functions such as `pathIsClear` in the applications. Since an application such as Delivery serves as a basic template for several other multi-robotics applications, including warehouse management, emergency response, natural resource monitoring, and on-site industrial fault diagnosis and repair. Consequently, we believe this decomposed verification approach can be used to prove invariants for such applications. In the next chapter, we present a distributed, collaborative mapping application to demonstrate its efficacy further.

CHAPTER 8: CASE STUDY : COLLABORATIVE MAPPING

In this chapter, we present a case study of the distributed grid mapping problem. The distributed grid mapping problem is a simplified version of the distributed Simultaneous Localization and Mapping problem, a classical problem in robotics research. This problem requires a set of robots to collaboratively mark the position of static *obstacles* within a given area D quantized by a *grid*, which any robot should avoid while moving in D.

The difference between this problem and the classical distributed SLAM, comes with the assumption that the robots know their global coordinates within the area of deployment, and only attempt to map static obstacles within this area. Further, the only sensors available for sensing obstacles are LIDAR based, and the robots are constrained to move in a 2D space. We first formally state the problem, invariants that a *Koord* application program should satisfy, and the assumptions that we make to verify such an invariant for this problem.

8.1 PROBLEM SETUP

Given a set of `N_sys` participating robots, the mapping problem is defined in terms of the following parameters:

1. A domain D which is a bounded rectangle $[a_1, a_2] \times [b_1, b_2]$ in $\mathbb{R}^2$ corresponding to the physical arena.

2. A *ground truth* function $world : D \mapsto \{0, 1\}$ that gives the actual occupancy of obstacles in this arena. That is, $\forall \vec{x} \in D$,

$$world(\vec{x}) = \begin{cases} 1 & \textit{if } \vec{x} \textit{ is occupied} \\ 0 & \textit{otherwise}. \end{cases} \tag{8.1}$$

3. A set $Q \subseteq D \cap \mathbb{Q}^2$ which is a quantized representation of D.

4. A Koord program variable, let us call it map_i, for each robot $i \in [N]$, that stores the Q-quantized, shared map.

Specifically, we assume that Q is a $(n_x \times n_y)$-grid representation of D for some resolution constants $n_x, n_y \in \mathbb{N}$. That is, $Q = \{q_{ij} \in \mathbb{Q}^2\}_{i\in[1..n_x], j\in[1..n_y]}$ such that every q_{ij} uniquely represents a disjoint $[x_i, x_{i+1}] \times [y_j, y_{j+1}]$ in D. The *quantization function* $quant : D \mapsto Q$ maps points in D to their quantized versions. That is $quant(\vec{x}) = q_{ij}$ iff $\vec{x} \in [x_i, x_{i+1}] \times [y_j, y_{j+1}]$.

The inverse is defined accordingly: $quant^{-1}(q_{ij}) = [x_i, x_{i+1}] \times [y_j, y_{j+1}]$ for any $q_{ij} \in Q$. The quantization defines a quantized version of the world $world_Q : Q \mapsto \{0, 1\}$, where

$$world_Q(q) = \begin{cases} 1 \ \Leftrightarrow \exists \vec{x} \in quant^{-1}(q), world(\vec{x}) = 1, \\ 0 \text{ otherwise.} \end{cases} \tag{8.2}$$

Figure 8.1 shows an example of a quantized domain with two obstacles shown in black.

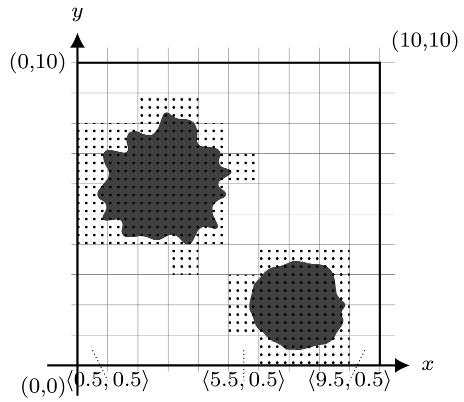

Figure 8.1: $D = [0.0, 10.0]^2 \subset \mathbb{R}^2$, $Q = \{0.5, \cdots, 9.5\}^2 \subset \mathbb{Q}^2$, for example, $world_Q(\langle 5.5, 0.5\rangle) = 0$ and $world_Q(\langle 6.5, 0.5\rangle) = 1$.

For each robot i, the program variable map_i will ideally store a quantized restriction of *world*. That is, the type of this variable will be $map_i : Q' \rightarrow \mathbb{B}$, for some $Q' \subseteq Q$. Finally, we will assume that each robot i occupies space within the arena D, and its position is available from a sensor port pos_i, which takes values in D.

Now we can formally state the desirable requirements of the mapping application.

1. *(Individually sound)* Always, each robot's map is a quantized restriction of the ground truth. That is, $map_i : Q_i \mapsto \mathbb{B}$, where $Q_i \subseteq Q$ and $map_i(q) = world_Q(q)$ for every $q \in Q_i$.

2. *(Consistent)* Always, robot maps are consistent. That is, for any two robots $i, j \in [N]$, $map_i(q) = map_j(q)$ for any $q \in domain(map_i) \cap domain(map_j)$.

3. *(Safe location)* Always, each robot is located in a part of the arena that is known to be free. That is, for any $i \in [\texttt{N\_sys}]$, $quant(pos_i) = map_i(q)$ for any $q \in domain(map_i) \cap domain(map_j)$.

4. *(Eventual completeness)* Eventually, the constructed maps cover *world*. That is, for each robot $i \in [\texttt{N\_sys}]$, $domain(map_i) = Q$.

```
using MotionScan
    sensors:
        Point position
        List⟨Point, Scan⟩ pscan
        bool reached
        PathPlanner planner
    actuators:
        List⟨Point⟩ path

allwrite:
    GridMap map

#omitting initialization
local:
  GridMap localMap
  Point target
  bool on_path = True
  List⟨Grid⟩ obstacles

GUpdate:
  pre MotionScan.reached
  eff: atomic:
    map = merge(map, localMap)
    on_path = False

NewTarget:
  pre !on_path
  eff:
    target = pickFrontierPos(map, \
                 MotionScan.position)
    obs = findObs(map)
    MotionScan.path = MotionScan.planner(target, \
                                 obs)
    if MotionScan.path != []:
      on_path = True
    else:
      on_path = False
    localMap = map

LUpdate:
  pre on_path and !MotionScan.reached
  eff:
    for p, s in MotionScan.pscan:
      localMap = merge(localMap, scanToMap(p, s))
```

Figure 8.2: *Koord* code for Distributed Mapping Application Mapping.

The Mapping algorithm shown in Figure 8.2 works in the following manner. Each robot constructs a *local grid map* over D using sensors, and updates it using information from other robots shared via a *global grid map*. In Mapping, the `MotionScan` module provides a `pscan` sensor used to read the LIDAR scan of the actual robot. The other sensors and actuators `position, reached, planner, path` have the same functionality as that in the `Motion` module. The shared allwrite variable `map` is used to construct a shared map of obstacles within the domain D and has type `GridMap`, which is a 2-D array representing a grid over D. The local variable `localMap` represents each robot's *local* knowledge of the domain D, and has the same type as D. There are three events: `NewPoint`, `LUpdate`, and `GUpdate`. A robot executing the `NewPoint` event, finds an unoccupied point to move to using a user-defined function `pickFrontierPos` and plans a path to it using `MotionScan.planner`. It then updates its `localMap` from the shared variable `map`. The `LUpdate` event updates the `localMap` with scanned sensor data while the robot is in motion, and the `GUpdate` event updates the shared `map` with the updated `localMap` information corresponding to the scanned data.

8.2 FORMAL MODELING AND ANALYSIS

We can define a function $chk : GridMap \mapsto \mathbb{B}$ such that $chk(g) := \forall q \in Q, (g(q) = 1) \Rightarrow (world_Q(q) = 1)$ to make sure the detected occupied grids are consistent. We then can formally define the invariant as:

Invariant 8.1.

$$[\![Consistent_i]\!]_c := chk(\texttt{localMap}_i) \wedge chk(\texttt{map}) \tag{8.3}$$

For the `NewPoint` event, the *Koord* Prover generates the post event configuration for robot i.

$$\begin{aligned} NewPoint_i := & \neg on\_path \wedge \texttt{localMap}'_i = \texttt{map} \wedge \texttt{map}' = \texttt{map} \\ & \wedge \texttt{map}'' = \texttt{map}' \wedge \texttt{localMap}''_i = \texttt{localMap} \wedge u_{vars} \end{aligned} \tag{8.4}$$

As in the previous examples, u_{vars} refers to unchanged variables after the event. The *Koord* Prover automatically generates proof obligation PO (8.1),

Proof Obligation 8.1.

$$\bigwedge_{i \in ID} [\![Consistent_i]\!]_{\boldsymbol{c}} \wedge NewPoint_i \Rightarrow [\![Consistent_i]\!]_{\boldsymbol{c''}} \tag{8.5}$$

By itself, this is verified automatically by the *Koord* Prover.

For the event *GUpdate*, a function summary for the *merge* function is required. We simply provide a summary *MERGE* stating that the *merge* function returns a map satisfying *chk* when given two maps satisfying *chk*.

Function Summary 8.1.

$$MERGE(m_1, m_2, m') := chk(m_1) \wedge chk(m_2) \Rightarrow chk(m') \tag{8.6}$$

The associated proof obligation with the above function summary is:

Proof Obligation 8.2.

$$\forall m1, m2, MERGE(m1, m2, merge(m1, m2)) \tag{8.7}$$

The *Koord* Prover generates the following post event configuration for robot i as

$$GUpdate_i := \begin{array}{l} Motion.reached \\ \wedge\ MERGE(\texttt{map}, \texttt{localMap}_i, \texttt{map}') \\ \wedge \neg on\_path' \wedge\ \texttt{localMap}'_i = \texttt{localMap}_i \wedge u_{vars} \end{array} \tag{8.8}$$

For proof obligation for this event is therefore:

Proof Obligation 8.3.

$$\bigwedge_{i \in ID} [\![Consistent]\!]_c \wedge GUpdate_i \Rightarrow [\![Consistent]\!]_{c''} \tag{8.9}$$

We omit the presentation of `LUpdate` as the analysis of `LUpdate` follows along the lines of the previous constraints.

Benchmark	`N_sys`	T_K (s)	T_V (s)	**Safe**
DMap	3	9.23	14.53	safe
DMap	4	9.33	19.25	safe
DMap	5	9.19	24.30	safe
DMap	10	9.31	59.81	safe

Table 8.1: Summary of semantics based verification for DMap. T_K is the symbolic post computation time in $\mathbb{K}$, T_V is the time taken for generation of constraints and verification in Z3 and `N_sys` is the number of robots in the system.

Table 8.1 summarizes the verification of soundness property of the DMap application on systems of different `N_sys`.

8.3 SUMMARY

In this chapter, we showed how we use the *Koord* Prover to verify inductive invariants, which do not necessarily require controller assumptions. We also formalized assumptions for this problem, to highlight how function summaries can be used to capture the notion of ground truth as we did with the *chk*. The three case studies for inductive invariant verification that we presented, including DMap in this chapter, show that our techniques are applicable across various types of multi-robot applications. In Chapter 9, we turn towards implementing *Koord* to enable simulation and deployment of *Koord* application programs.

CHAPTER 9: IMPLEMENTING KOORD : CYPHYHOUSE

This chapter discusses our *CyPhyHouse* toolchain [7], which includes an implementation of the *Koord* language, a simulator, and deployment interfaces. While the executable semantics reduces the gap between formal mathematical modeling and implementation, developing the entire *CyPhyHouse* toolchain provided several engineering challenges in achieving the system design.

In Section 9.5, we discuss how the language abstractions over platform-specific *controllers* are realized through actuator ROS topics, and how the *Koord* system obtains (real or simulated) information such as device positions through sensor ROS topics.

9.1 CYPHYHOUSE ARCHITECTURE: AN OVERVIEW

The *CyPhyHouse* runtime system refers to the collective management of hardware resources and software needed for execution of *Koord* programs, whether it be in simulation or deployment. Recall from Chapter 2 that a system running a *Koord* application has three parts: an application *program*, a *controller*, and a *plant*.

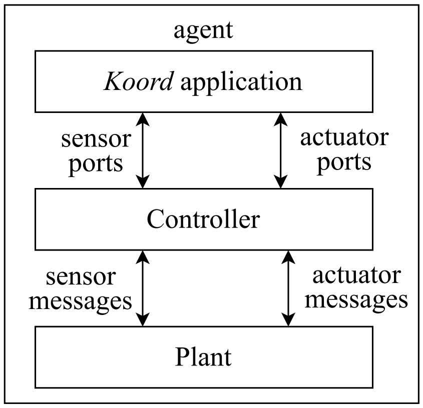

Figure 9.1: Overall System Architecture.

At runtime, the *Koord program* executes within the runtime system of a single robot. In a distributed system of multiple robots, a collection of programs execute on different robots that communicate using shared variables. Each compiled *Koord* program interacts with *CyPhyHouse* middleware via variables. In Section 9.4, we show how the middleware implements distributed shared memory(DSM) across robots.

The *plant* consists of the hardware platforms of the participating robots. The controller receives inputs from the program (through actuator ports), sends outputs back to the program (through sensor ports), and interfaces with the plant.

We developed a software-hardware interface (*middleware*) in Python 3.5 to support the three-plane architecture comprising the *Koord* runtime system.

As discussed in Chapter 3, our main insight in developing reliable robotics applications was that separation of distributed coordination logic, and low-level control would enable plugging in various verification techniques for the various components of a *Koord* application. Chapter 6 and Chapter 7 demonstrated a verification approach that exploits this separation of concerns. We had to ensure that the *CyPhyHouse* toolchain indeed maintained this separation in implementation.

The *Koord* toolchain has three main components:

- The *Koord* compiler, which accepts a *Koord* program as input and generates an executable python application,
- The *CyPhyHouse* middleware which interfaces with the compiled application includes the shared memory and provides the interfaces to the *CyPhyHouse* simulator and hardware platforms (robots),
- The *Koord* simulator, which provides a simulation environment with robot models in Gazebo.

The middleware includes distributed coordination logic, and the robot models in Gazebo and controllers used to drive actual robots include the control logic. We designed our middleware to be *modular*, to enable several replaceable implementations of the main language features, such as shared memory, mutual exclusion, and round synchronization. We designed *general* interfaces between the control logic and distributed coordination in the middleware to support robots with a variety of controller port abstractions. This *modular* design enables the portability of *Koord* applications across heterogeneous robots.

9.2 CYPHYHOUSE MIDDLEWARE HARNESS

Each *Koord* program compiles to Python application code, which is essentially a per-robot application *thread*, containing the declarations, and the distributed coordination logic in the form of conditional blocks controlled through the events' preconditions. We also added a termination feature to the application threads, to enable programmers to run their *Koord* applications for a limited number of rounds in simulation or deployment. We store each robot's worldview in an object called *global variable holder*, or GVH, which contains pointers to its corresponding robot's distributed shared memory, communication interfaces, its `pid`, and shared information from other robots in the *Koord* system.

9.3 THE KOORD COMPILER

The *Koord* compiler included with *CyPhyHouse* generates Python code for the application using all the supported libraries, such as the implementation of distributed shared variables using message passing over WiFi, motion automata of the robots, high-level collision, and obstacle avoidance strategies. The application then runs with the Python *middleware* for *CyPhyHouse*.

In our implementation of *Koord* in the *CyPhyHouse* toolchain, we enforce a Python-like indentation style, where every nested block is at a higher indent level. Note that the BNF grammar provided in the *Koord* syntax shown in Figure 3.1 does not include any indentation notions. We implemented the *Koord* compiler using Antlr (Antlr 4.7.2) in Java [62].

We added two terminals indent and dedent to the *Koord* syntax to facilitate indentation based parsing. We also added a logging feature to our implementation of *Koord* in *CyPhyHouse* for debugging purposes. One of the reasons for not including logging in the *Koord* formal semantics was that we did not want to include possible side effects due to I/O in the formal analysis, which is the primary objective of developing the executable semantics of *Koord* in $\mathbb{K}$.

Koord is statically typed. The parser creates a symbol table for all declared variables, and performs a table lookup on encountering a declared variable, and declares a parsing error if a variable does not match the type correctly. Type-inferencing and automatic casting may be implemented in the future, but as a side effect of Python code generation, we allow operations between permitted operations in Python. Strictly speaking, our formal semantics of *Koord* would not admit some of the programs permitted by our compiler. However, undeclared variables generate a parsing error in our implementation. The parser generates an abstract syntax tree consisting of basic control flow blocks, including variable declarations, variable initialization, event blocks, loops, and conditionals. For code generation, the AST is traversed using DFS after performing a topological sort on the nodes, and it directly generates the application python code. We do not perform any program analyses or optimizations on programs during compilation as we did not implement an intermediate representation.

We use ROS to handle the low-level interfaces with hardware. To communicate between the high-level programs and low-level controllers, we use Rospy, a Python client library for ROS, which enables the (Python) middleware to interface with ROS Topics and Services used for deployment or simulation.

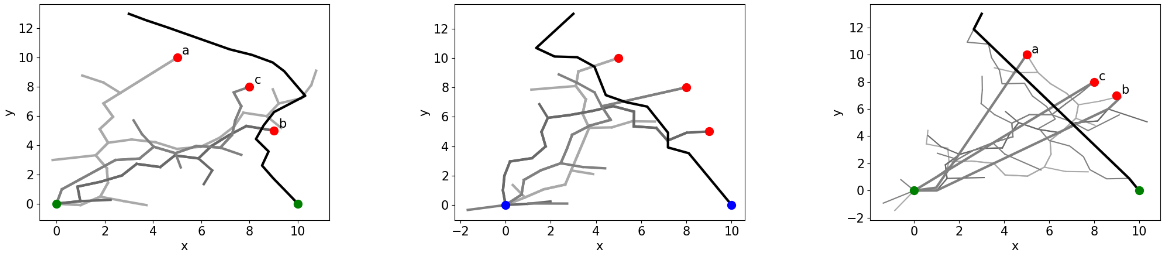

Figure 9.2: Different planners can work with the same code.

9.4 CYPHYHOUSE SHARED MEMORY AND COMMUNICATION

At a high level, the *CyPhyHouse* middleware propagates updates to a shared variable by one robot. These shared variable updates become visible to other robots in the next round. The correctness of a program relies on robots having consistent values of shared variables. When a robot updates a shared variable, the middleware uses message-passing to inform the other robots of the variable update. These changes should occur before the next round of computations.

CyPhyHouse implements the shared memory between robots through UDP messaging over WiFi. Any shared memory update translates to an update message which the robot broadcasts over WiFi. We assume the robots running a single distributed *Koord* application are running on a single network node, with little to no packet loss. However, the communication component of the middleware can be easily extended to support multi-hop networks as well.

The communication is handled by a separate thread, which has a pointer to the agent's global variable holder object.

9.5 CYPHYHOUSE INTERFACES FOR CONTROLLERS

The low-level control of a robot interfaces with the distributed coordination logic in the *Koord* program through the sensor and actuator ports. Given a robot with specific sensing and actuation capabilities, executing a *Koord* program on it requires an implementation of the interface in the *CyPhyHouse* toolchain.

For instance, if an application requires the robots to move, each robot uses an abstract class, *Motion automaton*, which must be implemented for each hardware model (either in deployment or simulation). This automaton subscribes to the required ROS Topics for positioning information of a robot, updates the *reached* flag of the motion module, and publishes to ROS topics for motion-related commands such as waypoint or path following. It also provides the programmer the ability to use different path planning modules as long as they support the interface functions. Figure 9.2 shows two robots executing the *same application* using different path planners. Figure 9.2 (*Left*) shows the *xy* plots of concurrently available paths during a round of the Delivery application using an RRT planner for two quadcopters. Figure 9.2 (*Middle*) shows the same configuration, where paths computed are not viable to be traversed concurrently. The green markers are current quadcopter positions, The black path is a fixed path, and the red points are unassigned task locations. Figure 9.2 (*Right*) shows the same scenarios under which paths cannot be traversed concurrently, except that a different RRT-based planner (with path smoothing) is used.

Motion automata mentioned earlier are simply an interface that can serve as a template for any interfaces between sensor and actuator ports. In practice, as a robot is added to the fleet, our design of the *Koord* semantics in $\mathbb{K}$ ensures that it is extensible (albeit with a blackbox implementation of the robot's dynamic behavior), and we can add the corresponding dynamic automata as an interface to the *CyPhyHouse* middleware.

9.6 PORTABILITY ACROSS EXECUTION PLATFORMS

Apart from the control components, all other components of the *CyPhyHouse* middleware are platform-agnostic. Our implementation allows any robot or system simulating or deploying a *Koord* program to use a configuration file (as shown in Figure 9.3) to specify the system configuration, and the runtime modules for each robot, including the dynamics-related modules, while using the same application code. It includes platform-agnostic settings for the robot, e.g., robot id (`pid`), device to run on (`on_device`), and the number of robots (`num_robots`), as well as platform-specific settings, e.g., path planners (`planner`) and position systems (`positioning_topic`). One of the main reasons for the high degree of portability is due to the Python implementation of the middleware, which enables execution without recompilation for every different platform.

9.7 CYPHYHOUSE MULTI-ROBOT SIMULATOR

CyPhyHouse includes a high-fidelity simulator for testing distributed *Koord* applications with a large number of heterogeneous robots in different scenarios. [1] Our modular middleware design allows us to separate the simulation of *Koord* applications and communications from the physical models for different platforms. Consequently, the compiled *Koord* applications, together with the communication modules, can run directly in the simulator—one instance

[1] a team led by Chiao Hsieh built the *CyPhyHouse* simulator.

```
robot:
    pid: 0
    on_device: hotdec_car
    motion_automaton: MoatTestCar
    ...
device:
    bot_name: hotdec_car
    bot_type: CAR
    planner: RRT_CAR
    positioning_topic:
        topic: vrpn_client_node/
        type: PoseStamped
    reached_topic:
        topic: reached
        type: String
    waypoint_topic:
        topic: waypoint
        type: String
    default_leader_pid = 0
    ...
num_robots: 3
```

Figure 9.3: A snippet of a sample configuration.

for each participating robot, and their simulated counterparts replace only the physical dynamics and the robot sensors. This flexibility allows programmers to test their *Koord* applications under different scenarios and with different robot hardware platforms. For example, one can use simpler physical models for early debugging of algorithms, and later simulate the same code with more accurate physics and heterogeneous platforms. Once the testing is satisfactory, the same code can be deployed on the actual platforms. The simulator can test different scenarios, with different numbers of (possibly heterogeneous) robots, with no modifications to the application code itself, instead directly modifying a configuration file as shown in Figure 9.4. A programmer with knowledge of distributed algorithms can develop simple protocols to write the computation logic for a coordination-based application for a multi-robot system. For instance, an application where each of the robots has some information about the positions of its neighbors and can set its targets to collectively form a shape with all the other robots (as shown in Figure 9.4). Figure 9.4(*Left*) shows the simulation of 9 drones running ShapeForm application, and Figure 9.4(*Middle*) shows the ShapeForm application on 16 drones. We specify different scenarios by changing the configuration file. Figure 9.4 (*Right*) shows a simulation of Delivery on heterogeneous robots. To our knowledge, this is the only simulator for distributed robotics providing such fidelity and flexibility.

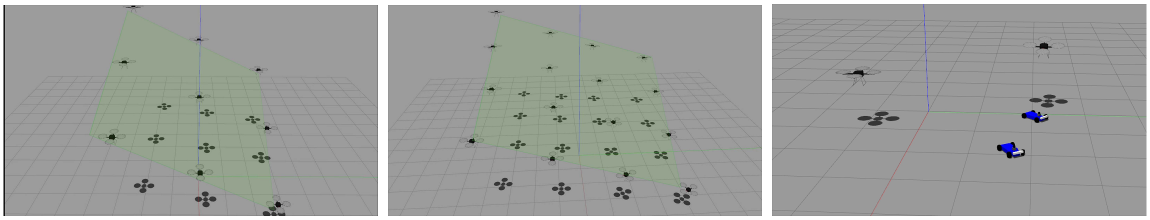

Figure 9.4: *CyPhyHouse* simulator running different scenarios with the same *Koord* application.

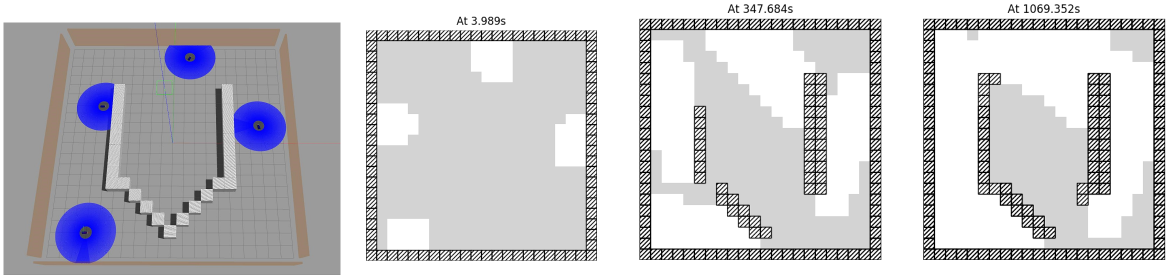

Figure 9.5: Four cars in a U-shaped world in simulator (*Left*). Visualization of the global map at three different time stamps (*Right*)

Figure 9.5 shows the stages of the collaborative map created by four robots of the U-shaped obstacle in the simulation environment using Mapping, described in Chapter 8.

Simulating *Koord* and Communication Our simulator spawns a process corresponding to each robot that encompasses all middleware threads to simulate the communication between robots faithfully. The communication handling threads in the individual processes can then send messages to each other through broadcasts within the local network. We support specifying distinct network ports for robots in the configuration file for simulating robots as multiple processes on a single machine. Since communication is through actual network interfaces, our work can be extended to simulate under different network conditions with existing tools.

Physical Models and Simulated World Our simulated physical world is developed based on Gazebo [44]. Furthermore, a simulated positioning system is provided to relay positions of simulated devices from Gazebo to middleware. For simulated devices, we integrate two Gazebo models from the Gazebo and ROS community, the car from MIT RACECAR project [41] and the quadcopter from the hector quadrotor project [56]. Further, we implement a simplified version of the position controller by modifying the provided default model. Users can choose between simplified models for faster simulation or original models for accuracy.

In addition to simulation, we also develop Gazebo plugins for visualization to help programmers replay logs recorded on real devices. The visualization can be either real-time or post-experiment: users may plot the movements or traces of models for real-time monitoring the progress; alternatively, programmers may visualize and analyze traces with Gazebo after experiments.

9.8 SIMULATOR PERFORMANCE ANALYSIS EXPERIMENTS AND RESULTS

Large scale simulations play an essential role in testing robotic applications and also in training machine learning modules for perception and control. Therefore, we performed experiments to measure the performance and scalability of the *CyPhyHouse* simulator. To study the performance of our simulator, we experiment with various scenarios consisting of different application *Koord* programs, increasing numbers of devices, or mixed device types. We then collect the usages of different resources and the number of messages in each scenario. Finally, we compare the statistics in resource usages and communications to study how our simulator can scale across different scenarios.

In our experiments, we use three *Koord* programs, including the example Delivery in Figure 7.2, a line formation program LineForm, and a program forming a square ShapeForm. For Delivery, we simulate with both cars and quadcopters to showcase the coordination between heterogeneous devices. For LineForm and ShapeForm, we use only quadcopters for simplicity to evaluate the growth of different statistics under increasing numbers of robots. For each experiment scenario with a timeout of 120 seconds, we collect the total message packets and packet length received by all robots, and sample the following resource usages periodically: Real-Time Factor (RT Factor, the ratio between simulated clock vs. wall clock), CPU percentage, Memory percentage, numbers of threads. All experiments are run on a workstation with 32 Intel Xeon Silver 4110 2.10GHz CPU cores and 32 GB main memory.

In Figure 9.6, we only show the average of each collected metric. Observe that for LineForm and ShapeForm. RT factor drops while all resource usages scale linearly with respect to the number of robots. Average number and size of packets received per second for each robot also grows linearly; hence, the sum of packets of all robots is quadratic in the number of robots. This quadratic message complexity is because both the size of each message for shared memory and the required number of messages are linear in the number of robots. One can further improve the communication complexity with a different distributed shared memory design.

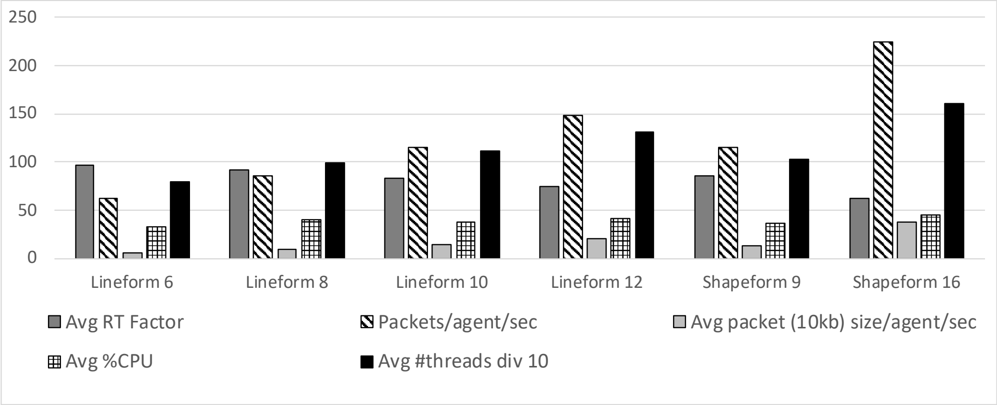

Figure 9.6: Resource usages and communications for ShapeForm and LineForm.

Figure 9.7 shows the results of using Delivery application on varying configurations of robots, to determine performance tradeoffs.

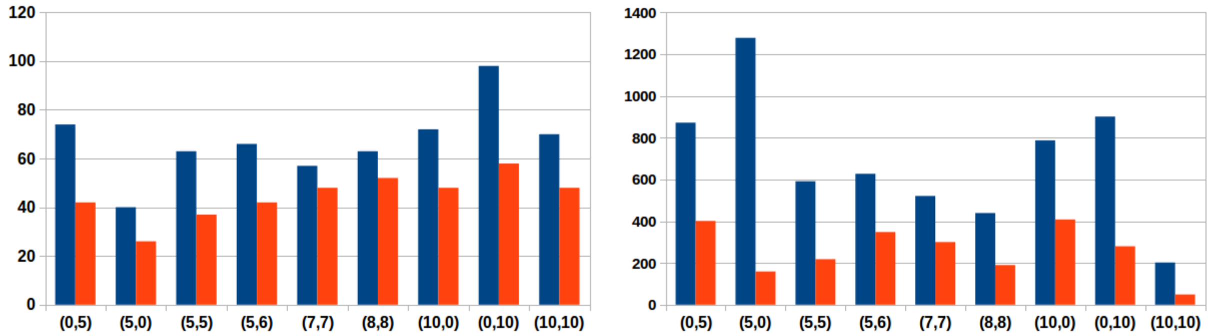

Figure 9.7: Performance evaluation of Delivery with CyPhyHouse Simulator for a 40 task sequence: the x axis refers to the number of robots, with $(cars, quadrotors)$. *(Left)* shows the total completion time in blue and and the time taken for conflict resolution in seconds in orange. *Right* shows the maximum distance (cm) travelled by a robot in blue and the minimum distance(cm) in orange.

Conflict resolution took more time with more robots. The completion time remained relatively stable. The maximum and minimum distances traveled were relatively similar, even with more robots. These results were affected by the non-determinism in the choice of paths and tasks by the robots and their previous relative positions when choosing the next task.

9.9 DEPLOYMENT SETUP

So far, we discussed the hardware-independent components of the *CyPhyHouse* toolchain; we now describe the experimental testbed we used to deploy our applications and the interaction of the different robots with the middleware[2].

9.10 VEHICLES

As previously mentioned, the CyPhyHouse framework targets heterogeneous robotics platforms. In order to demonstrate such capabilities, we have built both a car and a quadcopter. Note that the car has nonholonomic constraints, while the quadcopter has uncertain dynamics. In other standard settings, a roboticist would have to develop a separate application for each platform.

Quadcopter The quadcopter was assembled from off-the-shelf hardware, with a 40cm × 40cm footprint. The main computing unit consists of a Raspberry Pi 3 B+ along with a Navio2 deck for sensing and motor control. Ardupilot [4] handles stabilization and reference tracking. Between the *CyPhyHouse* middleware and Ardupilot, we include a hardware abstraction layer to convert setpoint messages from the *Koord* application into MAVLINK using the mavROS library ([78]), so Ardupilot can parse them. Since the autopilot was originally meant to use a GPS module, we also convert the current quadcopter position into the Geographic Coordinate System before sending it to the controller.

Car Similarly, the car platform uses off-the-shelf hardware based on the open-source MIT RACECAR project [41]. The computing unit consists of an NVIDIA TX2 board. In the car platform, instead of using Ardupilot to handle the waypoint following, we wrote a custom ROS node that uses the current position and desired waypoints to compute the input speed and steering angle using a Model Predictive Controller (MPC). The car has an electronic speed controller that handles low-level hardware control.

9.11 TEST ARENA AND LOCALIZATION

We performed our experiments in a 7m × 8m × 3m arena equipped with 8 Vicon cameras. The Vicon system allows us to track the position of multiple robots with sub-millimeter accuracy. However, we note that the position data can come from any source (for example, GPS, ultrawide-band, LIDAR), as long as all robots share the same coordinate system. While the motion capture system transmits all the data from a central computer, each vehicle only subscribes to its own position information.

[2] A team led by Joao Porto performed the hardware platform assembly and controller implementation.

All coordination and de-conflicting across robots is performed based on position information shared explicitly through shared variables in the *Koord* application. Aside from the Vicon localization system, the *Koord* middleware supports an interface with *any* localization system that generates the appropriate positioning ROS topics.

9.12 DEPLOYMENT INTERFACE WITH MIDDLEWARE

As mentioned earlier, the same application can be deployed using different path planners, which are associated with the platform-specific *motion automaton* through interfaces defined by the *CyPhyHouse* middleware. Both vehicles use RRT-based path planners [50] to compute a path to the next task. The car planner uses a bicycle model to compute the feasible paths, while the quadcopter planner assumes it can move in a straight line between points. The path generated is then forwarded to the robot via a ROS topic. The ROS topics required for positioning and setting waypoints of the vehicles were specified in the configuration. Each vehicle updates the *reached* topic when they reach a predefined ball around the destination. The car has nonholonomic constraints, while the quadcopter has uncertain dynamics, so in other standard settings, a roboticist would have to develop a separate application for each platform.

Real-Time Monitoring and Running Applications When an application executes on the platforms, the CyPhyHouse runtime framework records detailed timed logs, including program states, events, positions, sensor data, and message queues. These logs can be visualized and annotated to debug *Koord* application programs. We have also developed a simple 3D visualization in the Gazebo environment for real-time monitoring and for programmers to provide inputs to the applications.

Figure 9.8 shows a snapshot of Delivery deployment in the Intelligent Robotics Laboratory at the University of Illinois at Urbana Champaign, its visualization in Gazebo, and a closeup of the car and quadcopter we used to deploy this application.

9.13 AUTOMATICALLY LAUNCHING ON HETEROGENEOUS PLATFORMS

Without automation, the mundane task of compiling and running programs can become arduous when many devices are involved. It does become unmanageable and error-prone when different types of robot platforms have to get different versions of the executable, linked with correct libraries and IP addresses. Our *Launcher* program makes this a one-step automatic

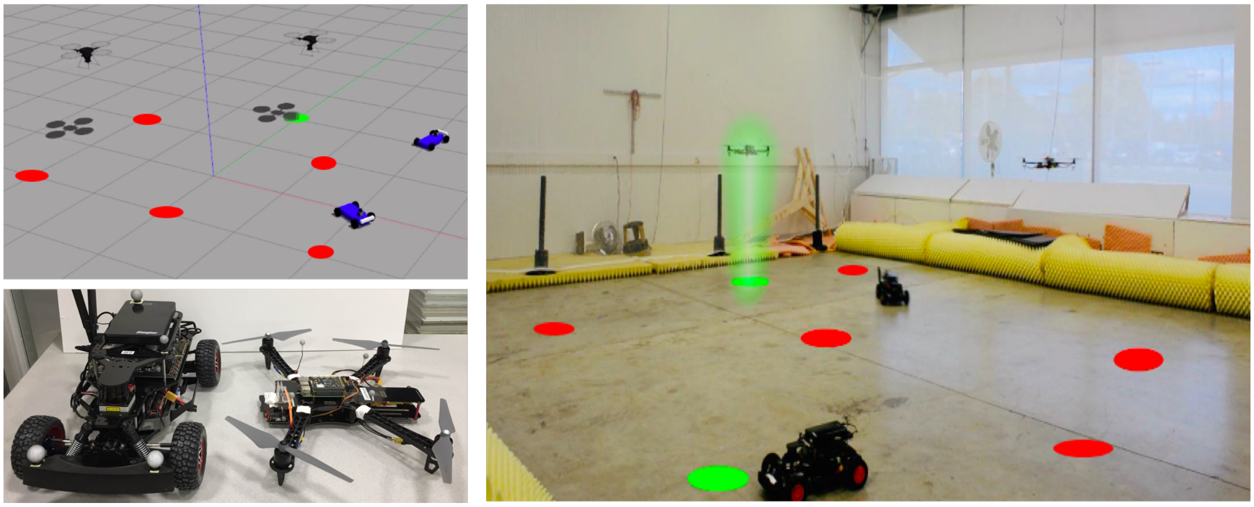

Figure 9.8: *Right:* Annotated snapshot of a distributed task allocation application deployed on four cars and drones using *CyPhyHouse* in our test arena. The red tasks are incomplete, and the green tasks are completed. *Left bottom:* different robotic platforms: F1/10 Car and quadcopter. *Left top:* Visualization of the same application running in *CyPhyHouse* simulator which interfaces with Gazebo.

process. It registers all the participating robots, compiles their application codes (using the above compiler and the correct platform-dependent libraries), loads the executables, sets up the indoor positioning system, and starts the application execution. In general, computing the membership of agents in a group of asynchronous processes is a well-studied hard problem [10]. Launching also involves the practical issue of discovering the IP addresses of the agents in the group.

9.14 EXPERIMENTS WITH DELIVERY ON UP TO FOUR VEHICLES

We performed several experiments involving the deployment of the Delivery application on cars and quadcopters. The measurement units and the coordinate system used by the indoor positioning system, the low-level controllers, and the Koord application program are all provided in meters. Our implementation of the quadcopter's waypoint follower stipulates that it reaches a location (task) provided it enters a $r = 30$cm ball around the location, and the indoor positioning system observes and transmits that fact. The car is considered to have reached a location if it enters a circle with radius $r = 25$cm around the specified location. The platform developer decides this parameter r, and it is not available to the programmer.

In our experiments with up to four vehicles, we found that there are fewer blocked paths with fewer robots, so each robot spends less time idling. However, this non-blocking effect is superseded by the parallelism gains obtained from having multiple robots. For example, three robots (two quadcopters and one car, or one quadcopter and two cars) show an average runtime of about 110 seconds for 20 tasks.

The average runtime for the same with four robots across 70 runs was about 90 seconds. We experience zero failures, provided the wireless network conditions satisfy the assumptions stated in Chapter 9. We discuss some of the specific experiments we performed with the Delivery application below.

Task Distribution Figure 9.9 (Left) shows the trajectories of a car and a quadcopter executing the Delivery app. The car visits the locations of all the ground level tasks ($z = 0$), and the quadcopter visits the locations of all the other ($z > 0$). The quadcopter takes off when the application starts and lands after all the tasks have been completed to avoid collisions. We added another car to the set of robots executing this app, requiring more coordination between the robots (as shown in in Figure 9.9 (Right)) that is handled automatically by the language. The programmer only needs to change the configuration file to reflect this change from 2 robots to 3.

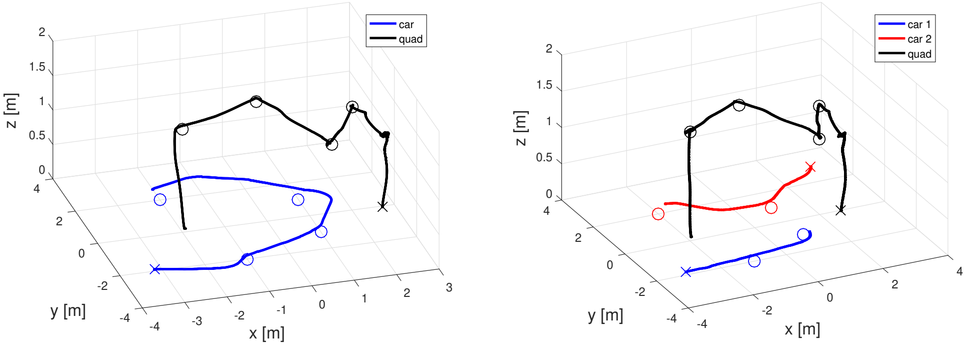

Figure 9.9: *Left:* trajectories for 1 car and 1 drone performing Delivery; *Right:* trajectories for 2 cars 1 drone. Circles represent the tasks given, while the crosses show the starting location of each robot.

Blocking Behavior Figure 9.11 demonstrates the efficacy of the task application itself in performing collision avoidance. For these experiments, we deployed the Delivery app shown in in Figure 7.2 on two cars, where the robots go to the first available task to which they can compute a path.

Figure 9.10 shows the computed (and eventually completed) paths. From Figure 9.11, we see that car 1 acquires the point at $\vec{x} = (2, -2)$ first and starts moving towards it at $t = 0.38$ s and car 2 only starts moving once car 1 has reached and released its path from the blocked routes.

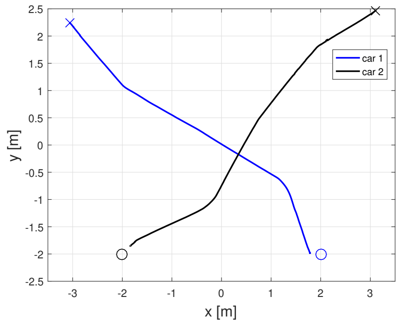

Figure 9.10: Trajectories for 2 cars in the *Blocking behavior* experiment.

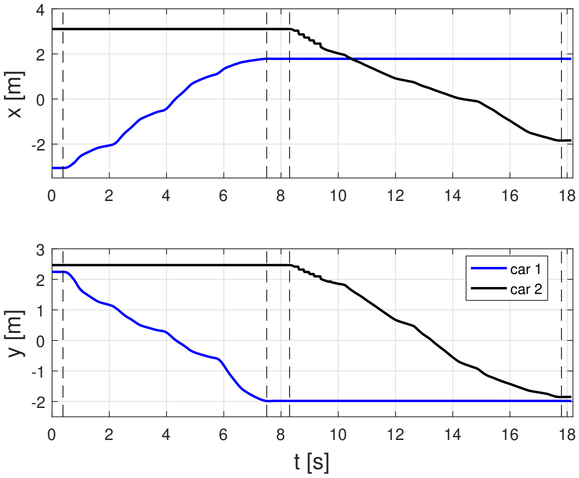

Figure 9.11: *Top:* x vs t for blocking paths for two cars. *Bottom:* y vs t for blocking paths for two cars. The dashed lines represent the starting and stopping times for the cars moving on their paths.

Repeated Tasks Here, we want to demonstrate that the system will robustly scale up with a longer list of waypoints while demonstrating the behavior exemplified in the previous two experiments. Since we have a limited test space, we gave the robots a list of repeating waypoints.

To see the blocking behavior for this experiment, we show a snippet of the positions of the two cars in Figure 9.12.

The resulting paths for two cars can be seen in Figure 9.13. We observed that the task distribution between the cars was roughly even during this experiment.

Robustness of Execution The Delivery application of Chapter 7 was run in over 100 experiments with different combinations of cars and quadcopters.

Figure 9.14 shows the (x, y)-trajectories of the vehicles in one specific trial run, in which two quadcopters and two cars were deployed. We can see concurrent movement when it is

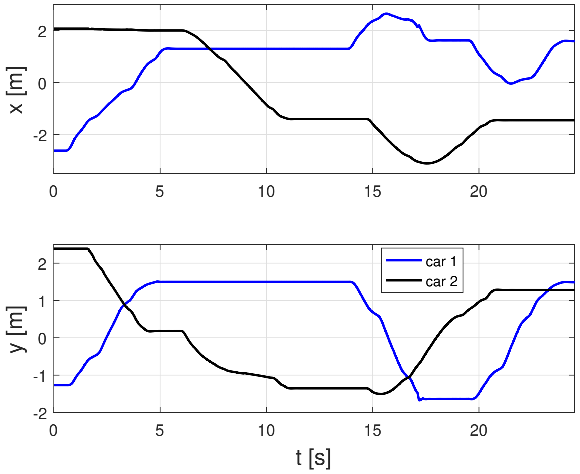

Figure 9.12: *Top: x* vs t for *Repeated tasks* for two cars. *Bottom: y* vs t for *Repeated tasks* for two cars.

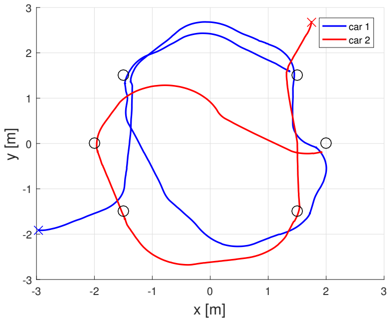

Figure 9.13: Trajectory for 2 cars in the *Repeated tasks* experiment.

safe (for example, at 13, 36, and 52 seconds), and only one robot moving or no robot moving when trying to compute a safe and collision-free path to a task.

Careful examination of the figure shows that all the performance requirements of Delivery are achieved, with concurrent movement when different robots have clear paths to tasks, safe separation, and robots getting blocked when there is no safe path found.

9.15 SUMMARY

In this chapter, we presented the *CyPhyHouse* software development and deployment toolchain for distributed robotic applications. We discussed our implementation of the *Koord* compiler, which uses the same BNF grammar that we presented in Chapter 3. We also presented details of the implementations of the shared memory and port abstractions and showed how our modular design of the *CyPhyHouse* toolchain allows plug-and-play features

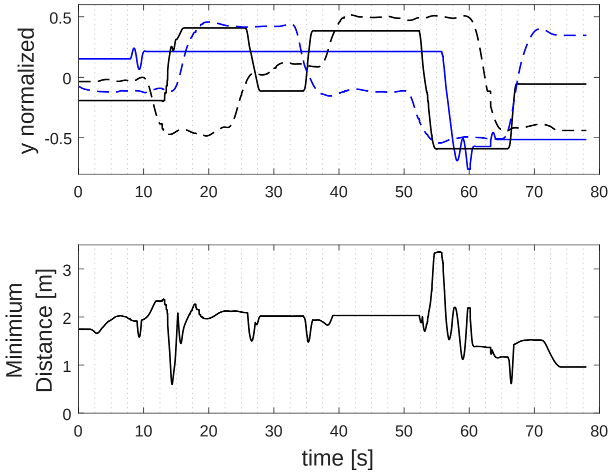

Figure 9.14: *Top* shows the y vs t trajectories of the vehicles during an execution of the task, and *bottom* shows the minimum distance between all robots. The vehicle positions in the *top* graph were normalized to improve visualization.

for different library functions, including path planners. We presented and profiled the high fidelity, scalable *CyPhyHouse* simulator that can execute and test instances of *Koord* applications with dozens of vehicles. We also showed how the *same* code could be simulated, and directly deployed on cars and drones with supporting platform-specific controllers. Finally, we performed a set of experiments using the Delivery application to gain insight into the performance of various configurations of robots and tasks in the Delivery application.

CHAPTER 10: CONCLUSIONS

In this thesis, we presented the *Koord* language and the *CyPhyHouse* software development and deployment toolchain for distributed robotic applications. It interfaces with and complements existing tools commonly used by roboticists, such as ROS, providing easy integration with popular platforms by almost any programmer. *Koord* and the *CyPhyHouse* toolchain enable programmers to develop and run distributed robotics applications in a hardware-independent fashion. *Koord* programs can be ported across platforms automatically with minimal effort from the application developer, and our high fidelity simulation can provide a valuable testing and debugging environment. Our goal in developing *Koord* was to provide a programming methodology for reliable distributed robotics that can exploit different analysis techniques while not requiring complete domain expertise.

While still in the development stages, *CyPhyHouse* has been used by more than 25 individuals for programming, simulating, and testing other applications like formation flight, and surveillance. Our experiences suggest the toolchain can indeed lower the barrier for entry into the distributed robotics. Our case studies with *Koord* demonstrate that DRAs with sensing, actuation, path planning, collision avoidance, and multi-robot coordination, can be succinct and amenable to formal analysis.

The *Koord* programmer only needs to understand *Koord*'s shared memory semantics, and the sensor and actuator port abstractions. On the other hand, the hardware engineer will need to validate that the target hardware platform indeed meets the port abstractions through testing. The symbolic execution of *Koord* programs can effectively automate the analysis of inductive invariants of the distributed coordination logic. Distributed robotics applications may have nondeterministic behaviors. We found that inductive invariants preserved during program transitions across *every event execution by any agent* can be completely verified by our approach. Further, the framework allows one to plug-in reachability analysis to validate/falsify controller assumptions for platform-dependent controllers.

10.1 CONTRIBUTION SUMMARY

Programming Language Abstractions for DRAs We explored the trade-off between the level of abstraction of the minutiae of multi-robot systems and achieved a simple yet expressive language design enabling a versatile set of multi-robot applications.

We introduced an abstract interface of *sensor* and *actuator ports* through which a *Koord* program interacts with its environment. The program can read from sensor ports to receive updates about its environment; furthermore, it can write to actuator ports to direct the low-level controllers and actuators. Beyond the names and types of these ports, the abstract interface may specify additional *controller assumptions* that these ports should satisfy. Application developers get to use these assumptions to reason about the *Koord* application, while platform engineers ensure that these assumptions are met when implementing the interface for each platform. These interfaces allow deploying and simulating the same *Koord* program on heterogeneous devices without any alteration, thus shortening the test-debug-deployment cycle.

We provided a *distributed shared memory (DSM)* construct for *Koord* applications on different robots to communicate with each other. This shared memory construct makes *Koord* applications very succinct. Moreover, it raises the level of abstraction for the programmers beyond sockets, message queues, and ROS topics and services [68].

A Formal Semantics of the Koord Language We developed the executable $\mathbb{K}$ semantics [70] of *Koord*. To our knowledge, this is the first formalization of a programming language for distributed cyber-physical systems, which has also been deployed on actual platforms. Our executable semantics of *Koord* in $\mathbb{K}$ assumes a *synchronous round-by-round computation model for the distributed system.* Each round lasts for a fixed period, and all robots synchronize at the beginning of each round. This synchronous model is a restrictive but standard model for distributed systems [52, 6]. While it does not eliminate concurrency control, it significantly simplifies programming and verification. *Koord* also provides constructs for mutual exclusion as additional mechanisms for concurrency control with shared memory mentioned above. We showed how this model can be implemented under typical synchrony assumptions for multi-robot networks.[1]

Verification approaches for inductive invariants for DRAs We presented a $\mathbb{K}$ symbolic execution-based formal verification methodology, which does *not* require explicit dynamical models of the platform. In practice, a detailed model of the platform (e.g., dynamic models for cars, wheel friction, engine torque.) may not be available. The *Koord* semantics is parameterized so that any available model or an actual blackbox executable for the platform can be plugged in. Our formal analyses facilitate both inductive invariant checking and state-space exploration with blackbox dynamics.

Using our formal analysis of *Koord* semantics, we identified and separated platform-independent and platform-dependent *proof obligations* for three benchmark applications. We proposed an approach to discharge the former obligations with existing SMT solvers

[1] Bounded message delays and clocks with bounded drifts.

automatically. On the one hand, platform-dependent proof obligations can suggest the infeasibility of implementing such systems, when they are difficult to discharge or quickly violated in simulation or testing. On the other hand, these proof obligations can serve as contracts for sensors, drivers, and operating system modules outside the purview of application developers. A useful outcome of this methodology is a list of formal assumptions for the platform or implementation-specific components.

We implemented a suite of benchmarks and used both our formal analysis tools of inductive invariant verification and bounded model checking to verify them or find counterexamples. As expected, the state space expands exponentially with the number of agents due to the inherent non-determinism of distributed CPS captured by our system, which is a factor for the running time of both our approaches. However, inductive invariant checking scales much better than explicit state bounded model checking.

The CyPhyHouse Toolchain for Koord Application Simulation and Deployment

Aside from the theoretical and formal analysis, we performed hardware experiments to confirm the assumptions we made while developing the language semantics for *Koord*. Our compiler translates *Koord* programs to Python application code, and device-independent middleware implements the memory model and module interfaces. We achieved distributed shared memory through round-based synchronous message passing. We created a simulation environment in Gazebo for testing *Koord* applications and validating assumptions, specifically on robot-environment behavior. The simulation environment only differs from a hardware deployment environment in the dynamic models of the physical platforms, but all software components are identical.

The simulator enables the user to test their discrete-event loop with simple motion models to test and debug the application program logic without incurring the cost of hardware deployment in case of buggy programs. The simulator also serves as a visualization tool as it can be used to plot the behavior of any program variables or controller variables.

10.2 FUTURE DIRECTIONS OF RESEARCH

Structured, Extensible Design It is difficult to expect that any language, including controller assumptions, can cover vastly different types of robots (which are constantly evolving). To that end, our design of *Koord* on top of the $\mathbb{K}$ semantic framework gives a flexible way to extend our language and tailor it to specific types of robots on demand.

We can explore further whether our port abstractions indeed support applications for robots with diverse sensing and actuation capabilities such as LIDAR, RADAR, and cameras for (possibly mobile) object detection. Another type of robotic applications that we can potentially use *Koord* for involves robots with arms for object manipulation.

Fault Tolerance We assumed a lack of failures in our formal model of *Koord* applications. While the *CyPhyHouse* implementation of *Koord* supports infrastructure for detecting dynamic leaves and joins, our formal semantics and analysis framework does not. Implementations of existing algorithms [36, 37] for fault tolerance for dynamic leaves and joins can be expressed in the system semantics in the future. For general-purpose distributed systems, even without crash failures, robots can suffer from delays and network failures, which may result in inconsistent shared memory. Our model includes a strong consistency assumption, where the robots receive all updates to the shared memory from the previous round. We can include weaker requirements of consistency in the *Koord* semantics and investigate how our decomposed verification approach can be adapted to these new semantics.

Verification of Liveness and Self-Stabilization We only explored the verification of invariants in this thesis. However, the correctness of distributed robotics applications can also rely on liveness properties, which indicate progress and self-stabilization. Typically, verifying such properties requires computing an appropriate *ranking function* [76]. Identifying whether a separation of distributed coordination and control can help verify such properties as well as a natural future direction of work.

Dynamic Neighborhoods One of the simplifying assumptions we made was that all robots are aware of the set of participating robots, which remains constant. The Actor model [3] is a formalism for concurrent computing, which treats actors as computational entities. Actors respond to message passing by making local decisions, sending more messages, and potentially even creating more actors. Actors can also only communicate with actors they know addresses for, thus motivating an idea of *neighborhoods*. A neighborhood of an actor A can be defined as a set of actors whose addresses are known to A. Taking inspiration from this model, we can adopt the idea of neighborhoods for robots, where a robot can only communicate with other robots in its neighborhood and does not know the identities of all the robots in the system. Neighborhoods can be determined by factors such as geographical proximity and connections to WiFi nodes. In the future, we can extend *Koord* with the communication semantics of actor models for distributed systems to support more realistic assumptions for the set of robots which can communicate with each other to support communication across wide geographical regions and multi-hop networks.

User Studies Our formal semantics and verification framework aside, the main objective of this thesis has been to make robotics programming easier and more accessible to a general programmer without specific robotics experience. In the future, user studies with programmers of varied experiences can allow us to determine whether we are indeed progressing towards this objective.

.

CHAPTER 11: REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1996.

[3] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[4] ArduPilot Development Team. Ardupilot.

[5] E. Asarin, O. Bournez, T. Dang, and O. Maler. Approximate reachability analysis of piecewise-linear dynamical systems. In B. Krogh and N. Lynch, editors, *Hybrid Systems: Computation and Control*, volume 1790 of *LNCS*, pages 20–31. Hybrid Systems: Computation and Control, 2000.

[6] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley &#38; Sons, Inc., USA, 2004.

[7] Anonymous Author. Anonymous paper. In *to appear in ICRA 2020*, page 6 pages, Paris,France, 2019. IEEE.

[8] Stanley Bak and Parasara Sridhar Duggirala. Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, pages 173–178. ACM, 2017.

[9] Rahul Balani, Lucas F. Wanner, Jonathan Friedman, Mani B. Srivastava, Kaisen Lin, and Rajesh K. Gupta. Programming support for distributed optimization and control in cyber-physical systems. In *Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, ICCPS '11, pages 109–118, Washington, DC, USA, 2011. IEEE Computer Society.

[10] Kenneth P Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[11] José-Luis Blanco. *Contributions to Localization, Mapping and Navigation in Mobile Robotics*. PhD thesis, PhD. in Electrical Engineering, University of Malaga, nov 2009.

[12] Timo Blender, Thiemo Buchner, Benjamin Fernandez, Benno Pichlmaier, and Christian Schlegel. Managing a mobile agricultural robot swarm for a seeding task. In *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society*, pages 6879–6886. IEEE, 2016.

[13] V.D. Blondel, J.M. Hendrickx, A. Olshevsky, and J.N. Tsitsiklis. Convergence in multiagent coordination consensus and flocking. In *Proceedings of the Joint forty-fourth IEEE Conference on Decision and Control and European Control Conference*, pages 2996–3000, 2005.

[14] Brandon Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völp, and André Platzer. Formally verified differential dynamic logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 208–221, New York, NY, USA, 2017. ACM.

[15] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. Veriphy: Verified controller executables from verified cyber-physical system models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 617–630, New York, NY, USA, 2018. ACM.

[16] M. Branicky. Multiple lyapunov functions and other analysis tools for switched and hybrid systems. *IEEE Transactions on Automatic Control*, 43:475–482, 1998.

[17] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[18] Miguel Campusano and Johan Fabry. Live robot programming: The language, its implementation, and robot API independence. *Science of Computer Programming*, 133:1 – 19, 2017.

[19] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *Computer Aided Verification*, pages 258–263. Springer, 2013.

[20] Alexander Cunningham, Manohar Paluri, and Frank Dellaert. Ddf-sam: Fully distributed slam using constrained factor graphs. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3025–3030. IEEE, 2010.

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and

Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[22] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe asynchronous event-driven programming. *SIGPLAN Not.*, 48(6):321–332, June 2013.

[23] Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A Seshia. Drona: A framework for safe distributed mobile robotics. In *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS)*, pages 239–248. IEEE, 2017.

[24] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. Psync: A partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 400–415, New York, NY, USA, 2016. ACM.

[25] Parasara Sridhar Duggirala and Sayan Mitra. Lyapunov abstractions for inevitability of hybrid systems. In *The 15th International Conference on Hybrid Systems: Computation and Control (HSCC 2012), Beijing, China.*, 2012.

[26] Parasara Sridhar Duggirala, Sayan Mitra, and Mahesh Viswanathan. Verification of annotated models from executions. In *EMSOFT*, 2013.

[27] Chuchu Fan, Bolun Qi, and Sayan Mitra. Data-driven formal reasoning and their applications in safety analysis of vehicle autonomy features. *IEEE Design & Test*, 35(3):31–38, 2018.

[28] Chuchu Fan, Bolun Qi, Sayan Mitra, and Mahesh Viswanathan. DryVR: Data-driven verification and compositional reasoning for automotive systems. In *Computer Aided Verification (CAV)*, July 2017.

[29] Ansgar Fehnker and Franjo Ivancic. Benchmarks for hybrid systems verification. In Rajeev Alur and George J. Pappas, editors, *HSCC*, volume 2993 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2004.

[30] 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 Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011.

[31] David Gauthier, Paul Freedman, Gregory Carayannis, and Alfred Malowany. Interprocess communication for distributed robotics. *IEEE Journal on Robotics and Automation*, 3(6):493–504, 1987.

[32] Mario Gerla, Eun-Kyu Lee, Giovanni Pau, and Uichin Lee. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. In *2014 IEEE world forum on internet of things (WF-IoT)*, pages 241–246. IEEE, 2014.

[33] Sukumar Ghosh. *Distributed systems: an algorithmic approach.* Chapman and Hall/CRC, 2014.

[34] Seth Gilbert, NancyA. Lynch, and AlexanderA. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23:225–272, 2010.

[35] Lars Grüne and Jürgen Pannek. Nonlinear model predictive control. In *Nonlinear Model Predictive Control*, pages 45–69. Springer, 2017.

[36] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In *International conference on reliable software technologies*, pages 38–57. Springer, 1996.

[37] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[38] Ge Guo and Wei Yue. Autonomous platoon control allowing range-limited sensors. *IEEE Transactions on vehicular technology*, 61(7):2901–2912, 2012.

[39] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[40] Taylor Johnson and Sayan Mitra. A small model theorem for rectangular hybrid automata networks. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, 2012.

[41] S. Karaman, A. Anders, M. Boulet, J. Connor, K. Gregson, W. Guerra, O. Guldner, M. Mohamoud, B. Plancher, R. Shin, and J. Vivilecchia. Project-based, collaborative, algorithmic robotics for high school students: Programming self-driving race cars at mit. In *2017 IEEE Integrated STEM Education Conference (ISEC)*, pages 195–203, March 2017.

[42] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata.* Synthesis Lectures on Computer Science. Morgan Claypool, November 2005. Also available as Technical Report MIT-LCS-TR-917.

[43] Marius Kloetzer and Calin Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Transactions on Automatic Control*, 53(1):287–297, 2008.

[44] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, Sep. 2004.

[45] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics*, 25(6):1370–1381, 2009.

[46] Alexander B. Kurzhanski and Pravin Varaiya. Ellipsoidal techniques for reachability analysis. In *HSCC*, pages 202–214, 2000.

[47] Michal Kvasnica, Pascal Grieder, Mato Baotić, and Manfred Morari. Multi-parametric toolbox (mpt). In *Hybrid systems: computation and control*, pages 448–462. Springer, 2004.

[48] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[49] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[50] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning, 1998.

[51] Nancy Lynch, Roberto Segala, Frits Vaandrager, and H. B. Weinberg. Hybrid I/O automata. In T. Henzinger R. Alur and E. Sontag, editors, *Hybrid Systems III*, volume 1066 of *LNCS*, New Brunswick, New Jersey, October 1995. Springer-Verlag.

[52] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., Cambridge, 1996.

[53] Nancy A Lynch. *Distributed algorithms.* Elsevier, 1996.

[54] Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Hybrid I/O automata revisited. In M.D. Di Benedetto and A.L. Sangiovanni-Vincentelli, editors, *Proceedings Fourth International Workshop on Hybrid Systems: Computation and Control (HSCC'01),* Rome, Italy, volume 2034 of *LNCS*, pages 403–417. Springer, March 2001.

[55] M. Mesbahi and Magnus Egerstedt. *Graph-theoretic Methods in Multiagent Networks.* Princeton University Press, 2010.

[56] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar von Stryk. Comprehensive simulation of quadrotor uavs using ros and gazebo. In Itsuki Noda, Noriaki Ando, Davide Brugali, and James J. Kuffner, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, pages 400–411, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[57] Pieter J Mosterman, David Escobar Sanabria, Enes Bilgin, Kun Zhang, and Justyna Zander. A heterogeneous fleet of vehicles for automated humanitarian missions. *Computing in Science & Engineering*, 16(3):90, 2014.

[58] César Muñoz, Víctor Carreño, and Gilles Dowek. *Formal Analysis of the Operational Concept for the Small Aircraft Transportation System*, pages 306–325. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[59] Adithyavairavan Murali, Tao Chen, Kalyan Vasudev Alwala, Dhiraj Gandhi, Lerrel Pinto, Saurabh Gupta, and Abhinav Gupta. Pyrobot: An open-source robotics framework for research and benchmarking. *arXiv preprint arXiv:1906.08236*, 2019.

[60] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52 –60, aug. 1991.

[61] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. *A Survey on Domain-Specific Languages in Robotics*, pages 195–206. Springer International Publishing, Cham, 2014.

[62] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[63] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[64] D. Pickem, P. Glotfelter, L. Wang, M. Mote, A. Ames, E. Feron, and M. Egerstedt. The robotarium: A remotely accessible swarm robotics research testbed. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1699–1706, May 2017.

[65] Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. Buzz: An extensible programming language for self-organizing heterogeneous robot swarms. *CoRR*, abs/1507.05946, 2015.

[66] J Norberto Pires and JMG Sá Da Costa. Object-oriented and distributed approach for programming robotic manufacturing cells. *Robotics and computer-integrated manufacturing*, 16(1):29–42, 2000.

[67] Stephen Prajna and Ali Jadbabaie. Safety verification of hybrid systems using barrier certificates. In *In Hybrid Systems: Computation and Control*, pages 477–492. Springer, 2004.

[68] Morgan Quigley, Ken Conley, Brian P. 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*. IEEE, 2009.

[69] Redmond R Shamshiri, Cornelia Weltzien, Ibrahim A Hameed, Ian J Yule, Tony E Grift, Siva K Balasundram, Lenka Pitonakova, Desa Ahmad, and Girish Chowdhary. Research and development in agricultural robotics: A perspective of digital farming, 2018.

[70] Grigore Rosu and Traian Florin Serbanuta. K overview and simple case study. In *Proceedings of International K Workshop (K'11)*, volume 304 of *ENTCS*, pages 3–56, Illinois, June 2014. Elsevier.

[71] Giovanni Russo and Jean-Jacques E Slotine. Symmetries, stability, and control in nonlinear systems and networks. *Physical Review E*, 84(4):041929, 2011.

[72] Hussein Sibai, Navid Mokhlesi, Chuchu Fan, and Sayan Mitra. Multi-agent safety verification using symmetry transformations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–190. Springer, 2020.

[73] David St-Onge, Vivek Shankar Varadharajan, Guannan Li, Ivan Svogor, and Giovanni Beltrame. ROS and buzz: consensus-based behaviors for heterogeneous teams. *CoRR*, abs/1710.08843, 2017.

[74] Mia N Stevens and Ella M Atkins. Multi-mode guidance for an independent multicopter geofencing system. In *16th AIAA Aviation Technology, Integration, and Operations Conference*, page 3150, 2016.

[75] Sebastian Thrun et al. Robotic mapping: A survey. *Exploring artificial intelligence in the new millennium*, 1(1-35):1, 2002.

[76] John N. Tsitsiklis. On the stability of asynchronous iterative processes. *Theory of Computing Systems*, 20(1):137–153, December 1987.

[77] Alphan Ulusoy, Stephen L Smith, Xu Chu Ding, Calin Belta, and Daniela Rus. Optimality and robustness in multi-robot path planning with temporal logic constraints. *The International Journal of Robotics Research*, 32(8):889–911, 2013.

[78] Vladimir Ermakov. mavros.

[79] Brian C Williams, Michel D Ingham, Seung H Chung, and Paul H Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE*, 91(1):212–237, 2003.

[80] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M Murray. Receding horizon control for temporal logic specifications. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, pages 101–110. ACM, 2010.

[81] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard M Murray. Tulip: a software toolbox for receding horizon temporal logic planning. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*, pages 313–314. ACM, 2011.

[82] Damien Zufferey. The REACT language for robotics, 2017.