ABSTRACTIONS FOR SAFETY ASSURANCE OF AUTONOMOUS SYSTEMS

BY

CHIAO HSIEH

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 Urbana-Champaign, 2023

Urbana, Illinois

Doctoral Committee:

Professor Sayan Mitra, Chair
Professor Madhusudan Parthasarathy
Associate Professor Sasa Misailovic
Assistant Professor Stanley Bak, Stony Brook University

Abstract

An autonomous system has components to perceive the environment, control the hardware, and communicate with other agents. Ideally, the formal analysis of the closed-loop system would have access to explicit models of all components. However, explicit models of perception and dynamics are often unavailable or intractable for formal verification. Instead, executable models for these components are available for simulation and testing in practice. In this thesis, we will present our compositional verification framework for certifying and assuring safety for systems with combinations of distributed communication, learning-enabled perception, and black-box dynamics. Our insight is to construct abstractions of these components for the end-to-end system-level safety analysis; then we empirically validate each component against its abstraction by sampling executable models.

For distributed robotic applications, we focus on constructing the abstraction for heterogeneous motion dynamics. Our notion of *port assumptions* for dynamics decomposes the safety assurance into two steps: (a) the formal safety proof of the distributed system using port assumptions, and (b) the validation of port assumptions using data-driven reachability analyses. We learned that this compositional reasoning generalizes to both synchronous and asynchronous communications between heterogeneous vehicles. We are able to derive the collision avoidance guarantee in a distributed delivery application using our Koord framework for shared variable communication between ground vehicles and quadrotors. We apply the same idea on asynchronous message passing-based Unmanned Air-traffic Management protocols (UTM) and verify safe separations between quadrotors and fixed-wing airplanes.

We also study autonomous systems with visual perception enabled by deep neural networks (DNNs). Our main insight is to search for ground truth-based approximate abstractions for perception. Our notion of approximate abstractions bypasses the challenges in the formal specification of perception and high dimensional image domains, and the precision of the approximate abstraction can be estimated empirically. We study both single agent and multi-agent systems including three practical vision-based autonomous systems: (a) a lane tracking system for an autonomous vehicle, (b) a corn row following system for an agricultural robot, and (c) a vision-based multi-agent swarm formation. We are able to provide end-to-end assurance for all systems and examine the precision using simulations.

In summary, our compositional verification framework outlines a pragmatic path to provide safety assurance of autonomous systems. We formalize (approximate) abstractions for compositional verification and develop approaches to search for abstractions. This allows us

to formally verify as many components in the system as possible. For other components that are intractable for formal verification, we search for abstractions of these components and rigorously validate the abstractions via testing and data-driven verification. We are able to gain safety assurance using abstractions in all our case studies and validate the abstractions with high-fidelity Gazebo and AirSim simulations.

Table of Contents

Chapter 1: Introduction

Autonomous systems are being increasingly deployed in safety-critical applications, such as manufacturing [1], agriculture [2, 3], delivery [4], and transportation [5, 6]. For example, there are more than two million drones registered for recreational and commercial uses in the United States since 2021 [7]. *System-level* (also called *end-to-end*) safety assurance is important for such autonomous systems to achieve acceptable levels of human and property safety, meet safety laws and regulations, and gain public trust. Not surprisingly, there is now a growing momentum for regulating learning-enabled autonomous systems from government [8] and corporate bodies [9]. Regulatory agencies from many industries including aerospace [10], automotive [11], robotic surgery, and manufacturing, are creating processes and guidelines. NASA and FAA are actively developing traffic management systems for the safe operations of the unmanned aircraft [12].

Current approaches to ensure system-level safety mainly rely on large-scale simulations and field tests. However, the required amount of simulations and tests can be prohibitively large because unsafe behaviors of autonomous systems are rare events by nature. For example, to validate if the autonomous vehicles can achieve the same performance as human drivers, i.e., a fatality rate of 1.09 deaths per 100 million miles, it is estimated that driving for approximately 8.8 billion miles is required [13].

Formal verification addresses the rarity of unsafe behaviors and provides a rigorous way of obtaining strong assurance. Formal techniques aim to mathematically check that all behaviors of a system, described using a *formal model*, satisfy the requirements given as *formal system-level specifications*. This requires the existence of formal descriptions for both the specification and the model of the implementation. Such descriptions are usually given in a unified mathematical framework with a formal semantics that associates a mathematical interpretation with each description, and the framework would support checking if the interpretation of the model conforms to the specification. Formal verification has been successfully applied for the correctness of hardware design [14] as well as software design [15].

In particular, we advocate the use of compositional verification using *abstractions* of components. In essence, if we can find abstractions to capture all behaviors of components, and we show that the system with the components replaced by their abstractions satisfies the system-level specifications, then the original system satisfies the system-level specifications. Frameworks such as *assume-guarantee* reasoning [16, 17, 18], hybrid input/output automata [19, 20, 21], model checking combined with reachability analysis [22, 23], and contract-based design [24] propose different abstractions and compositional reasoning ap-

proaches to verify autonomous systems. There are however unique challenges in applying compositional verification on autonomous systems.

1.1 CHALLENGES IN COMPOSITIONAL VERIFICATION OF AUTONOMOUS SYSTEMS

Certifying autonomous systems with compositional verification relies on analyzing all system components and their interactions. Ideally, the compositional verification would have access to models of all system components, and the formal analysis tools would support all models. In reality, some components in autonomous systems are still beyond the capability of state-of-the-art formal approaches. For example, the formal methods community has identified key challenges and limitations on formally verifying safety with learning-enabled components [25, 26]. Several recent research papers are starting to tackle this problem [27, 28, 29, 30]. See Section 1.4 for a more detailed discussion of related approaches. In this thesis, we studied several autonomous systems including distributed coordination (Chapter 4), unmanned aircraft traffic management (Chapter 5), smart manufacturing (Chapter 6), autonomous vehicle with vision (Chapter 7), and vision-based formation systems (Chapter 8). Our experiences suggest that we can design a *good* decomposition to apply formal verification on as many components as possible and obtain statistical guarantees for components currently intractable for formal analyses.

More precisely, we consider that an autonomous system consists of several components given by the system architecture. Figure 1.1 shows an example autonomous vehicle, and Figure 1.2 shows its system architecture with four components. A decomposition is to decompose the system into *modules*, and each module can group one or multiple components. We propose that a *good* decomposition should ease the following tasks for compositional verification:

(1) For components amenable to formal analyses, the decomposition defines modules that are easier to extract mathematical models for existing formal approaches.

(2) If a module must include components intractable for formal analyses, the decomposition helps derive abstractions of the module from system-level specifications. The abstractions should define ground-truth input/output behaviors for modules (or *test oracles* [31]) so that statistical assurance can be obtained through testing or data-driven verification.

These criteria are inspired by the recent progress in *data-driven verification* to provide statistical safety guarantees for black-box vehicle dynamics. The DryVR framework [23] pro-

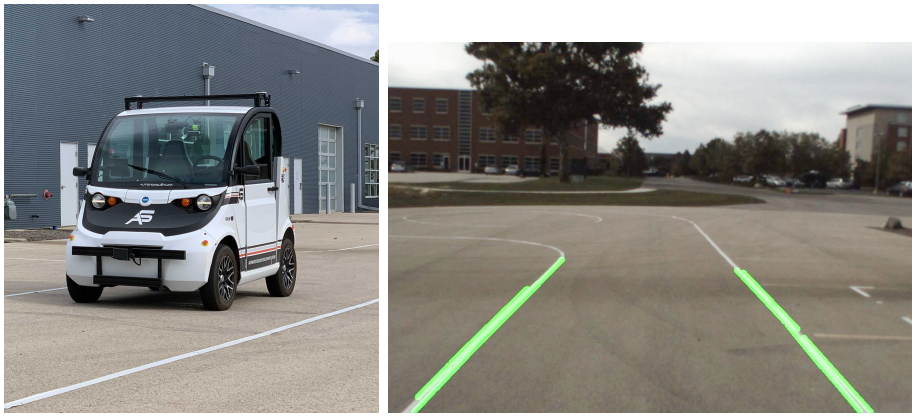


Figure 1.1: Vision-based lane detection component on an autonomous vehicle.

poses the setup of the white-box discrete transition graph combined with the black-box continuous dynamics for hybrid system verification. Following the same insight, our KOORD framework [32, 33] shows that the verification of distributed robotics can be decomposed into platform-independent coordination and platform-dependent motion dynamics. This decomposition allows the application of different analyses more suitable for individual components.

In addition, one should carefully consider the following components when designing the decomposition:

(1) dynamics of the hardware platform, such as the motion dynamics of drones and ground vehicles, and

(2) perception using high dimensional sensing, such as computer vision and deep neural networks (DNNs) with cameras or LiDAR scanning.

Here we elaborate on the challenges due to dynamics and perception components and how a good decomposition can address them.

- *Environmental Uncertainties.* It is known that extracting complete formal models of certain components is difficult due to unaccounted environmental uncertainties. Consider an autonomous vehicle in Figure 1.1 as an example. It is difficult to account for all external factors, such as the surface friction of the road, the slope, the lighting conditions, the temperatures, etc., that can affect the motion dynamics and perception components of the vehicle. Especially, DNN-based perception functions are known to be susceptible to adversarial external perturbations [34]. Interestingly, if we group perception with dynamics as a module and consider the evolution of the dynamics, the DNN-based perception could benefit from the smoothness of natural signals, and be more robust to occasional misclassifications [35].

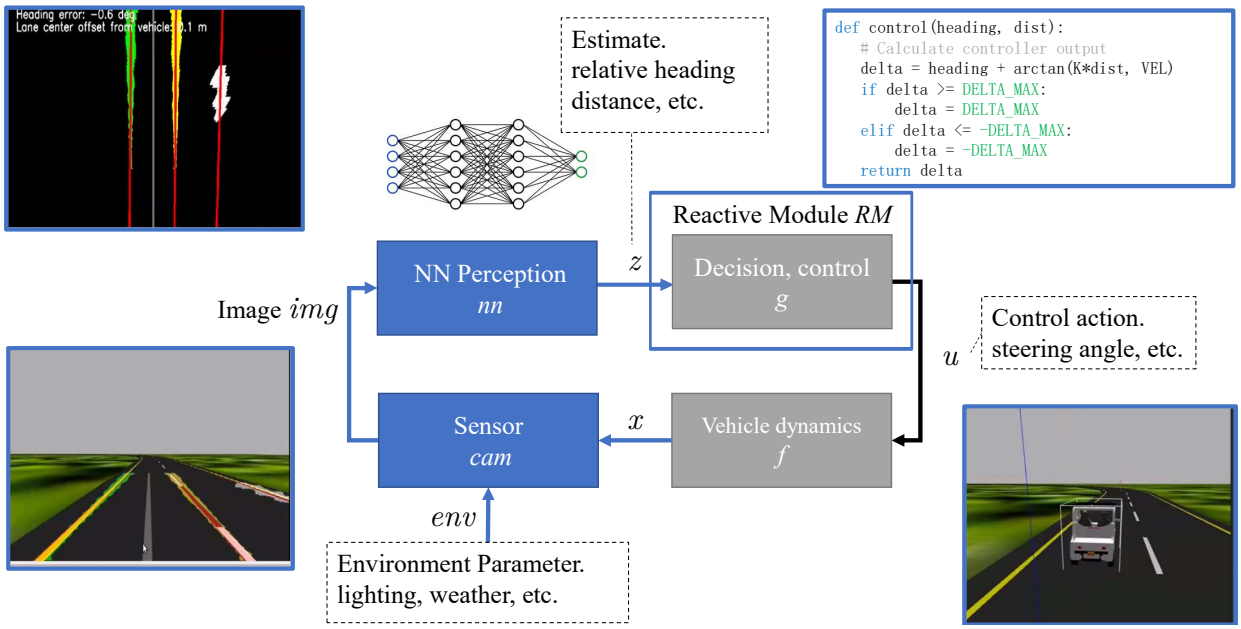


Figure 1.2: System architecture of an autonomous vehicle with camera and DNN-based perception.

- *High Dimensional Interfaces.* The decomposition can significantly affect the scalability problem in formal verification. For example, if we directly decompose the system into four modules for four components in Figure 1.2. the interface between the camera component and the NN perception component will worsen the scalability problem for its high dimensional image space: the very deep convolution neural network (VGGNet) [36] and the deep residual networks (ResNet) [37] commonly used for visual perception have an input space of RGB images with 224×224 pixels, which, if modeled naively, could encode $256^{224\times224\times3} \approx 10^{362000}$ possible inputs. Alternatively, if we choose a different decomposition which groups the camera component and the NN perception component as a module, the number of dimensions of the interface is significantly smaller.

- *Hard-to-Formalize Tasks.* The decomposition can help formalize the correctness of a module. The combination of environmental uncertainties and high dimensional space leads to the difficulty in defining the desired behavior of a component. For instance, Figure 1.2 shows a NN perception component for lane detection in autonomous vehicles. To define the correctness for this component requires a mathematical formulation over the set of images. The lane lines in each image may look very different under variations of lighting conditions, textures of the roads, colors of the lane markers, etc. In general, formally specifying such perceptual tasks is difficult due to ambiguity [38]. On the

other hand, the specification for a position estimation task is obvious. The correctness on estimating the position of the lane is well-defined given the ground-truth position of the car. Hence, if we group the camera sensor and the NN perception as a module for position estimation, the specification for this module is unambiguous.

1.2 IMPORTANCE OF ABSTRACTIONS FOR REASONING ABOUT SAFETY

Our insight to provide safety assurance via compositional verification is based on one crucial observation: both dynamics and perception are used to capture *the behavior of the environment* under which the autonomous system is operating. Modeling autonomous systems as *reactive systems* [16, 39], or equivalently *open systems* [40], presents a promising way of addressing the above-mentioned challenges. Figure 1.3 summarizes the compositional reasoning steps as a proof tree enabled by modeling autonomous systems as reactive systems. From the reactive systems' perspective, one considers the overall system as the composition of a *reactive module* (denoted as RM) interacting with a real environment (denoted as $REnv$) and check if the system (denoted as $RM \| REnv$) satisfies desired safety properties (denoted as $[\![Safe]\!]$). The real environment $REnv$ is used to represent components discussed in Section 1.1 that are intractable for formal reasoning, and a *model of the environment* (denoted as Env), i.e., the assumptions on the behavior of the real environment, is used to simplify the real environment $REnv$. If the model Env is an *exact abstraction* (or over-approximation) of the real environment $REnv$ (denoted as $REnv \preceq Env$), and if the *abstract system* (denoted as $RM \| Env$) obtained by substituting the real environment $REnv$ with the abstraction Env is verified (denoted as $RM \| Env \models [\![Safe]\!]$), then we can infer the safety of the original system.

$$\dfrac{\dfrac{RM \models_k [\![Inv]\!] \qquad Env \models_{\Delta T} [\![Inv]\!] \qquad \models [\![Inv \Rightarrow Safe]\!]}{RM \| Env \models [\![Safe]\!]} \qquad REnv \preceq Env}{RM \| REnv \models [\![Safe]\!]}$$

Figure 1.3: Proof tree for the compositional verification of the system composed of the reactive module RM and the real environment $REnv$. The proof is to find a model of environment Env that is an abstraction of $REnv$ and show both RM and Env preserves a given invariant $[\![Inv]\!]$, and the invariant $[\![Inv]\!]$ ensures the safety $[\![Safe]\!]$.

Nevertheless, the verification of the abstract system ($RM \| Env \models [\![Safe]\!]$) as a whole is likely intractable, and hence we apply compositional reasoning to decompose the verification of the abstract system into verifying the reactive module and the model of environment separately. Here we assume an *inductive invariant* (denoted as $[\![Inv]\!]$) is available that can

ensure the system-level safety (denoted as $\models [\![Inv \Rightarrow Safe]\!]$). This allows us to apply existing induction-based verification techniques on the reactive module (denoted as $RM \models_k [\![Inv]\!]$).

Therefore, the main proof obligations to complete the compositional reasoning are to ensure that (1) the model of environment is an abstraction of the real environment (denoted as $REnv \preceq Env$), and (2) the model of environment preserves the invariant (denoted as $Env \models_{\Delta T} [\![Inv]\!]$). However, it is often intractable to rigorously prove that *Env* is an exact abstraction of the real environment *REnv*. For example, if *REnv* includes the perception model implemented with deep learned neural networks, showing that *Env* over-approximates *REnv* still suffers from the state explosion and external uncertainty problems. We instead explore a compromise: an approximate model *Env* is constructed which still preserves the invariant ($Env \models_{\Delta T} [\![Inv]\!]$), but the abstraction relationship between *Env* and *REnv* can have an error, and this error can be estimated arbitrarily precisely with high probability using data-driven verification or simulations. That is, we check $REnv \preceq Env$ empirically and provide probability estimates. We call this approximate model *Env* a *safe abstraction* of the real environment *REnv*.

This thesis therefore focuses on systematically searching for safe abstractions. The designed approaches cover a wide variety of autonomous systems including single-agent and multi-agent systems in combinations with vision-based perception, heterogeneous dynamics, synchronous and asynchronous communications.

1.3 CONTRIBUTIONS OF THE THESIS

In this thesis, we propose a general formalism, CPReact, for modeling and verifying a general class of autonomous systems. We separate the reactive module *RM* from the environment *Env* following the "good decomposition" criteria outlined in Section 1.1 and provide a compositional approach to formally verify the end-to-end system. The nondeterminism in our CPReact framework allows us to model different types of distributed systems under different communication channels as the reactive module, and we show that the verification of the reactive module can be achieved using existing approaches, such as model checking or concurrent program verification, from software verification and distributed computing literature.

We show that the safety verification of the environment can be reduced to finding a model of the environment *Env* representing the assumptions on the real environment *REnv*. To reiterate, we search for a model of the environment satisfying two criteria:

- *Safe.* The invariant preserving the system safety is preserved under the assumptions

made by the model ($Env \models_{\Delta T} [\![Inv]\!]$).

- *Abstraction.* The model of the environment over-approximates the real environment, i.e., the model simulates all possible behaviors of the real environment ($REnv \preceq Env$).

We further elaborate on our approaches for finding safe abstractions of the dynamics and the perception.

Abstraction of Dynamics in Distributed Robotics Our CPReact model generalizes the Koord language framework in [32] to deal with different distributed systems under different communication models. The CPReact model can model not only synchronous communication via shared variables in Koord but also asynchronous communication via message passing, and this allows us to support completely different application domains.

In Chapter 4, we describe the separation of coordination logic from low level motion dynamics in the CPReact model. It decomposes the verification of the whole system into two independent proof tasks: (1) the verification of discrete transitions defined by Koord programs with shared variables, and (2) the verification of continuous state transitions defined by the dynamics. The verification of discrete transitions can be rigorously proven with existing program analysis tools. The notion of *controller port assumptions* abstracts the heterogeneous platform-specific dynamics and reduce the verification of continuous state transitions to reachability analysis. That is, we have designed controller port assumptions to over-approximate the set of all transient states reached in a continuous transition in order to prove the system safety, and controller port assumptions can be validated against a particular motion dynamics model. It further enables us to apply data-driven reachability analysis via DryVR to discharge the proof without explicit models.

In Chapter 5, we follow the same decomposition and apply compositional verification on Unmanned Aircraft Traffic Management systems (UTM), SkyTrakx [41]. Different from the Koord programs, UTM protocols rely on asynchronous message passing communication. Nevertheless, we are able to specify abstractions of dynamics represented as operation volumes (OV), and we use DryVR to check if dynamics models follow OVs.

In Chapter 6, we further showcase how to model smart manufacturing systems which are distributed systems with agents doing nonidentical tasks. We provide the compositional modeling and simulation/testing for smart manufacturing systems, and depict a path forward to formally verify distributed systems with nonidentical agents and dynamics.

Approximate Abstraction of Vision-based Perception In Chapter 7, we propose a practical method for reasoning about the safety of systems with vision-based perception

base on our paper [30]. Our method is based on systematically constructing the Approximate Abstract Perception (AAP) for the model of environment from system-level safety requirements, data, and program analysis of the reactive modules that are downstream from perception. These approximations have some desirable properties like being low-dimensional and tractable for existing verification tools. Further, the closed-loop system, with the approximation substituting the actual perception model, is verifiably safe.

Establishing a formal relationship between the actual perception and the AAPs remains well beyond available verification techniques. However, we do provide a useful empirical measure of their closeness called *precision*. Overall, our method can trade off the size of the approximation against precision. We apply the method to two significant case studies (1) a vision-based lane tracking controller for an autonomous vehicle and (2) a controller for an agricultural robot. We show how the generated approximations for each system can be composed with the downstream modules and be verified using program analysis tools like CBMC. Detailed evaluations of the impacts of size, and the environmental parameters (e.g., lighting, road surface, plant type) on the precision of the generated approximations suggest that the approach can be useful for realistic autonomous systems.

In Chapter 8, we further study the vision-based drone formation system to showcase how to analyze a distributed autonomous system with vision-based perception. To our knowledge, we provide the *first* approach to provide safety assurance for realistic vision-based distributed control systems with abstractions.

High-Fidelity Simulations In addition to the theoretical contributions, we developed and integrated high-fidelity simulation models for data-driven analysis and validation in every case study. The simulation models include a variety of motion dynamics for quadrotors [42], racecars [43], fixed-wing airplanes [44], smart manufacturing systems [45], electric golf carts [46], as well as photo-realistic camera images from AirSim [47] for vision-based perception. Our simulation code is open-source and available online at `https://cyphyhouse.github.io/` for Gazebo simulations and `https://publish.illinois.edu/aproximated-abstract-perception/` for AirSim simulations.

1.4 RELATED WORKS

In this section, we review the literature on formal verification of autonomous and cyber-physical systems. This is a vast and growing area, we do not include the vast literature on the testing and statistical methods for autonomous systems, and we refer the readers to a comprehensive review [48] published in 2023. We focus our discussions on works that provide

formal modeling and verification techniques for systems with distributed communications, learning-enabled components, and vision-based perception.

For modeling autonomous systems, there are many domain specific languages for designing robotic applications. A detailed survey in [49] identified 137 publications on domain-specific modeling languages that target core robotics concerns, but only few provide the formal verification and validation capabilities as discussed in [33]. We will discuss these formal modeling and verification frameworks in Section 1.4.1.

For the verification of systems with learning-enabled components, we start with the closely related works that deal with vision-based perception in Section 1.4.2. We then discuss the works on verifying *neural feedback systems* using NN controllers in Section 1.4.3.

Additionally, we summarize the rapid advancements on the verification tools for NNs as an isolated component in Section 1.4.4, and we discuss how we may integrate these tools into our approach for the system-level verification.

1.4.1 Formal Verification Tools for Distributed Robotics

There are several frameworks that support formal reasoning for distributed robotics. Overall, instead of a monolithic formal model and single verification approach, the community has recognized that there is a need to support a variety of formal techniques to address different components, such as communication, coordination, planning, and motion control.

The UCLID5 framework [50, 51] is designed to support models involving combinations of hardware and software. Their *multi-modal* language supports various models including transition systems, sequential programs, and concurrent systems along with diverse specifications such as invariant, temporal logic, hyper-properties, and simulation relations. This enables one to tackle verification problems for systems with heterogeneous components and varied specifications. UCLID5 emphasizes on the use of syntax-guided and inductive synthesis (SyGuS) [52] to automate steps in modeling and verification and applies satisfiability modulo theories (SMT) solvers and SMT-based verification methods. Both our CPReact model and UCLID5 are solving the same sub-problems of finding inductive invariant and synthesizing abstractions of components. In comparison, we utilize reachability analysis and constrained optimization that are more suitable for models with continuous time and state spaces. It will be interesting to combine our approach with SMT-based syntheses from UCLID5 to deal with hybrid systems.

DRONA [53] is a framework for multi-robot motion planning built on P language [54] for partially distributed system, and it has been deployed on drones. Our Koord language, CPReact model, and the underlying CyPhyHouse tool-chain [55] aims to be more gen-

eral, 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 manually finding safe abstractions. DRONA's model checker explores reachable states up to a given depth (number of transitions from an initial state). The induction proof enabled by our CPReact modeling framework achieves unbounded verification.

VeriPhy [56] is a modeling framework based on differential dynamic logic [57] and its theorem proving engine KeYmaera X [58]. It relies heavily on differential dynamic logic, and this poses difficulties on integrating different formal techniques such as reachability analysis for dynamics.

1.4.2 Verification on Vision-Based Control Systems

Several works start to apply formal verification on vision-based control systems including VerifAI [27, 59] by Ghosh et al., [28] by Katz et al., NNLander-VeriF [29], and [60] by Păsăreanu et al. These works and our approach all generate a simpler model (abstraction or not) of the vision-based perception component using ground-truth information either from simulators or labeled datasets.

VerifAI [27] and related publications [61, 62] provide a comprehensive framework to *falsify* a closed-loop system with ML-based perception. Their techniques focus on the falsification of the system specification including fuzz testing, simulation, neural network redesign, counterexample guided data augmentation, syntheses of hyperparameters and model parameters. Our work on AAPs (Chapter 7) provides a safe approximation and complements the falsification approaches of VerifAI. Further, the Counter Example Guided Inductive Synthesis-based approach in [59] uses VerifAI to find counterexamples of the closed-loop system, and it synthesizes a controller as well as learns a surrogate model for the simulator and perception components at the same time. On the other hand, our approach by design reuses existing Lyapunov stability and barrier certificate proof techniques for inferring perception models, and decouple the design of the controller from the perception models.

[28], [29], and our work on AAPs are similar in spirit to the white paper [18] by Păsăreanu et al. In [18], a compositional approach is proposed based on inferring abstractions/contracts for learning-enabled components. The compositional approach enables separate component verification with specialized tools, e.g., one can use software model checking for a discrete-time controller, hybrid model checking for the plant component in an autonomous system, and DNN analysis for the perception module. [28] in particular trains generative adversarial networks to produce a simpler network. This simpler network transforms states and envi-

ronment parameters to estimates similar to our AAP. NNLander-VeriF in [29] approximates the perception by encoding the camera sensors and computer vision components as a NN using geometric models for 3D-vision [63]. It then verifies NN for perception along with NN controllers for an autonomous landing system using NN verification tools (see Section 1.4.4). In comparison, our work provides an intelligible set-valued function to approximate the perception.

Most recently, [60] proposed a probabilistic abstraction for formal probabilistic analyses. Similar to our AAPs, their approach replaces the camera and the network with a compact probabilistic abstraction built from the confusion matrices computed for the DNN on a representative image data set. The abstraction is then integrated to a probabilistic system model, such as Discrete Time Markov Chains, and obtain system level probabilistic safety guarantees. Our approach differs in that our construction of abstractions is in a safety guided way. We derive abstractions with respect to not only the image data set but also the system level safety property; thus we provide the more traditional worst case safety guarantee for our approximated system in contrast to their probabilistic guarantees.

1.4.3 Verification of Closed-Loop Neural Feedback Systems

In the past three years (2020-2023), there is a focus on the analysis of *neural feedback systems* with neural network controllers including verification [64, 65, 66, 67], reachability analysis [68, 69, 70, 71, 72], statistical model checking [73], and synthesis [74]. [66, 67] specifically focus on developing and verifying a neural network replacement for ACAS-Xu collision avoidance decision tables. Such controller NNs are typically much smaller than the DNNs used for perception. Therefore, in order to apply these techniques, we can derive AAPs for the DNN-based perception pipeline and encode AAPs as smaller NNs or equivalent models, and we may reuse the techniques for neural feedback systems to verify the approximated closed-loop system.

1.4.4 Neural Network Robustness Verification

Motivated by safety criticality of autonomous systems, the problem of verifying neural networks has received keen attention. Recently, there are plenty of works on verifying an isolated neural network such as ReLuplex [75], NNV [64], Verisig [70], and the line of works in the ETH Robustness Analyzer for Neural Networks (ERAN) including AI^2 [76], DeepPoly [77], and the PRIMA framework [78]. Lately, the α, β-CROWN project [79, 80] advanced the NN verification research significantly. α, β-CROWN was able to solve more than

a dozen benchmarks proposed by industry and prevailed in the neural network verification competitions. We refer readers to the summary reports of the verification of neural network competitions in 2021 and 2022 [81, 82] for a complete list. It is however challenging to use these NN verification tools to analyze the DNN-based perception component with respect to the system-level safety. Existing NN verification tools for image processing DNNs currently only verifies *local robustness* against perturbations on a single camera image [81, 82]. These tools check if a DNN is robust against adversarial perturbations on selected image inputs. The NN verification results therefore only hold for a small neighborhood around the given inputs, and this is insufficient to ensure the robustness over the entire image domain required by the end-to-end safety of the autonomous system. Formal analysis over the entire image domain, coined as *global robustness*, is still beyond the capabilities of existing NN verification techniques.

Our work on finding AAPs decomposes the system-level safety to abstractions for modules. We can further check local robustness using NN verification tools to verify if the perception component satisfies AAPs for selected images. To further complete the verification of the system, more research on verifying the global robustness of DNN-based perception is needed.

1.5 ORGANIZATION OF THE THESIS

This thesis is organized as follows:

(1) In Chapter 2, we present our CPReact modeling and verification framework based on reactive systems and induction proofs.

(2) In Chapter 3, we formally define the abstractions for environments and the synthesis problem of safe abstractions.

(3) In Chapter 4, we present our CPReact model for analyzing distributed Koord applications with synchronous communication, and we present our approach to search for safe abstractions of the dynamic models using data-driven reachability analysis.

(4) In Chapter 5, we present our SkyTrakx tool for the same CPReact model-based safety analysis on Unmanned Aircraft Traffic Management protocols with asynchronous message passing communications and communication delays.

(5) In Chapter 6, we study the compositional modeling and testing for smart manufacturing system with non-identical agents and dynamic models.

(6) In Chapter 7, we present our approach to search for safe abstractions of vision-based perception using constrained optimization solvers.

(7) In Chapter 8, we analyze distributed drone formations with vision-based positioning components which combines a distributed system with vision-based perception.

(8) Lastly, we conclude and discuss ongoing research and future directions in Chapter 9.

Chapter 2: Models and Proof Techniques for Autonomous Systems

In this thesis, we design a formal modeling framework for autonomous systems, named CPReact, based on the reactive system models in [16, 83]. We consider an autonomous system as the interaction between a *physical environment* and a *reactive module*. The reactive module maintains an internal state and interacts with the environment via *percept* variables as input and *feedback* variables as output. This separates the reactive module representing the known components, such as software controllers, from the environment representing the incomplete or unknown components, such as black-box vehicle dynamics, complex vision, or a partially observable environment in general. Our approach then divides the safety verification of the autonomous system into (1) checking that the reactive module preserves the safety and (2) searching for a *safe abstraction*, a model representing assumptions on the incomplete physical environment.

In this chapter, we will first provide basic definitions for expressing the states via variables and expressions in Section 2.1. We then formalize a CPReact system model as the composition of a physical environment and a reactive module in Section 2.2. In Section 2.3, we provide the induction-based safety analysis for CPReact.

2.1 VARIABLES, VALUATIONS, EXPRESSIONS, AND PREDICATES

We follow the notations in the two textbooks [83, 84] and use a set of variables over arbitrary *data types* and the *valuations* of these variables to model the state-space of autonomous systems. For commonly used types, we denote by $\mathbb{B}$, $\mathbb{N}$, $\mathbb{R}$, and $\mathbb{R}_{\geq 0}$ the sets of Boolean values, natural numbers, real values and nonnegative reals, respectively. An *enumerated type* contains a finite number of symbolic constants; an example of such a type is the set $\{\texttt{ON}, \texttt{OFF}\}$ with two values.

For a set of variables X, a *valuation* over X is a function q such that for each variable $\mathtt{x} \in X$, $q(\mathtt{x})$ is a value belonging to the respective domain of the variable $\mathtt{x}$. When X is finite with a cardinality of $k \in \mathbb{N}$, we also denote a valuation $q = \{\mathtt{x}_1 \to v_1, \mathtt{x}_2 \to v_2, \ldots, \mathtt{x}_k \to v_k\}$ to explicitly enumerate the value v_i of each variable $\mathtt{x}_i \in X$. We let $\mathcal{Q}_X$ denote the set of all possible valuations of X (over the respective types). Also, for any set of variables X, we let X' denote a fresh set of primed variables corresponding to variables in X, i.e., $X' \stackrel{\texttt{def}}{=} \{\mathtt{x}' \mid \mathtt{x} \in X\}$, and the type for $\mathtt{x}'$ is the same as the type for $\mathtt{x}$. For simplification purposes, we consider that the valuations over X' are interchangeable with the valuations over X, and the set of all valuations $\mathcal{Q}_{X'}$ is the same as $\mathcal{Q}_X$.

An *expression* e is constructed using variables in X, constants, and primitive operations over types corresponding to these variables. We assume all expressions are *well-typed* by construction. Given a valuation q, we denote the *evaluation* of the expression e over q as $\mathit{eval}(e, q)$. A *predicate* *Pred* over the set of variables X is a Boolean expression; hence, the evaluation of a predicate $\mathit{eval}(\mathit{Pred}, q)$ is either true or false. In addition, we use the notion, $[\![\mathit{Pred}]\!]_X$, to denote the subset of $\mathcal{Q}_X$ where *Pred* evaluates to true, in other words, we can define $[\![\mathit{Pred}]\!]_X$ as shown in Formula (2.1):

$$[\![\mathit{Pred}]\!]_X \stackrel{\texttt{def}}{=} \{q \in \mathcal{Q}_X \mid \mathit{eval}(\mathit{Pred}, q)\} \tag{2.1}$$

In addition, we use several shorthand notations for the valuations of disjoint sets of variables. Given multiple pairwise disjoint sets of variables, $X_1, X_2, \ldots$, we use $\mathcal{Q}_{X_1,X_2,\ldots}$ as the shorthand of all possible valuations for the product set, $\mathcal{Q}_{X_1} \times \mathcal{Q}_{X_2} \times \ldots$, and we denote a valuation as $(q_1, q_2, \ldots) \in \mathcal{Q}_{X_1,X_2,\ldots}$ with $q_1 \in \mathcal{Q}_{X_1}$, $q_2 \in \mathcal{Q}_{X_2}$, and so on. Similarly, we use $[\![\mathit{Pred}]\!]_{X_1,X_2,\ldots}$ as the subset of $\mathcal{Q}_{X_1,X_2,\ldots}$ where *Pred* evaluates to true, and we use $[\![\mathit{Pred}]\!]$ without denoting the sets of variables when there is no ambiguity. We further use the notion $\mathit{Pred}[X'_i/X_i]$ to denote the predicate constructed by replacing the variables $\mathtt{x}_i \in X_i$ in *Pred* with the primed version of variables $\mathtt{x}'_i \in X'_i$.

Example 2.1. Let us consider how to model the states of an autonomous vehicle with its position and the current time. We can define a set of two variables $X = \{\mathtt{pos}, \mathtt{clk}\}$ in which `pos` represents the position and `clk` represents the clock. For simplicity, we assume that `pos` is of the type $\mathbb{R}$ to represent a 1D position, and `clk` is of the type $\mathbb{R}_{\geq 0}$ to represent the time. A state of the vehicle is then a valuation, e.g., $q_1 = \{\mathtt{pos} \rightarrow 1.1, \mathtt{clk} \rightarrow 0.6\} \in \mathcal{Q}_X$, denoting that `pos` maps to 1.1 meter and `clk` maps to 0.6 seconds. Let us then consider an example predicate in Formula (2.2) over the set of variables X:

$$\mathit{Pred} \stackrel{\texttt{def}}{=} (\mathtt{pos} \geq 0 \wedge\ \mathtt{clk} \geq 0.5) \tag{2.2}$$

We can easily see, if we evaluate the predicate over the valuation q_1, we know $\mathit{eval}(\mathit{Pred}, v_1)$ is true because $(1.1 \geq 0 \wedge\ 0.6 \geq 0.5)$ evaluates to true when variables are substituted with their values. Similarly, we can consider a different valuation, $q_2 = \{\mathtt{pos} \rightarrow -1.1, \mathtt{clk} \rightarrow 0.1\}$ and show that $\mathit{eval}(\mathit{Pred}, q_2)$ evaluates to false. Then by definition, $[\![\mathit{Pred}]\!]_X$ denotes a set of all valuations where `pos` is non-negative and `clk` is at least 0.5.

Formula (2.3) shows how to construct $\mathit{Pred}[X'/X]$ by replacing `pos` with $\mathtt{pos}'$ and `clk` with $\mathtt{clk}'$.

$$\mathit{Pred}[X'/X] \stackrel{\texttt{def}}{=} (\mathtt{pos}' \geq 0 \wedge\ \mathtt{clk}' \geq 0.5) \tag{2.3}$$

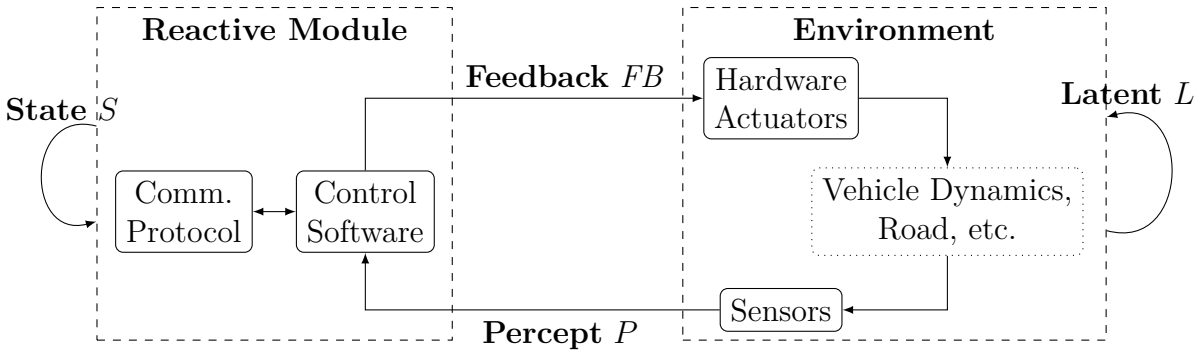


Figure 2.1: A simplified autonomous vehicle architecture with components separated into the reactive module *RM* and the environment *Env*.

2.2 SYSTEMS INTERACTING WITH ENVIRONMENTS

We formalize an autonomous system as the interaction between two modules — the reactive module *RM* modeling the software programs interacting with the environment *Env*. The reactive module and environment communicate with each other in *discrete time steps* in rounds. In each round, the percepts are fed to the reactive module, which in turn computes both an update to its state and feedback to the environment. Here we use the system architecture of an autonomous vehicle as an example (Figure 2.1). *RM* is the control software of the car; *Env* includes the car's hardware platform, the road, as well as the surrounding environment that influences the behavior of the car. Note that vehicle dynamics are part of the environment, and the feedback from the program is used to effect only certain variables, such as an update to the target position for driving the vehicle. The state of the environment such as the position of the vehicle or the visible area of the road then changes according to the feedback. This completes the round, and new percepts are fed to the reactive module to start the next round.

More precisely, we now specify *RM* and *Env* using a set of variables over arbitrary domains, with state-space being the valuations of these variables, a set of initial states, and a transition relation describing (potentially nondeterministic) changes to these variables. We start with the interfacing variables between *RM* and *Env*. Let us fix a set of *percept variables* P that captures some attributes of reality in the environment. In an autonomous driving setting, for example, this set of percepts would be variables that give the position of the car in some fixed coordinate system, e.g., the ego vehicle's coordinates, the lay of the road, the motion of pedestrians and vehicles nearby, etc. Let us also fix a set of *feedback FB* variables that captures the feedback the *RM* gives to effect changes to the environment. Again, in the autonomous vehicle example, this feedback can be variables for control of brakes,

acceleration, and steering angle of the vehicle.

We start with the modeling of an environment. Let us fix an environment *Env* with a set of latent variables L and the percept variables P. Let us assume an initial state predicate $Init_{Env}$, which defines a set of initial valuations $[\![Init_{Env}]\!]_{L,P}$ for latent and percept variables, as well as a transition relation $[\![T_{Env}]\!]_{L,P,FB,L',P'}$. Let $\mathcal{Q}_{L,P}$ denote the set of states of the environment and we will denote particular states as a pair (l, p). A transition $((l, p), fb, (l', p'))$ denotes that the environment, when in state (l, p) and reading feedback fb, can transition to (l', p'), and give the program the percept p'.

We then model the reactive module *RM*. Let the reactive module *RM* have a state-space defined by a set of variables S. Let $\mathcal{Q}_S$ denote the set of states of the reactive module, the initial state predicate $[\![Init_{RM}]\!]_S \subseteq \mathcal{Q}_S$ defines possible initial states. The transition relation is a relation $[\![T_{RM}]\!]_{S,P,S',FB'}$. A transition of the form (s, p, s', fb') means that the system, when in state s, reading a percept value p, can transition to state s', and give the feedback fb' to the environment.

Now we model the autonomous system as the composition of *RM* and *Env* synchronized in *rounds*. In each round, *RM* first transitions to next state, and then *Env* transitions to next state to complete a round. Formally, the global behavior of the system is defined over *configurations*, i.e., the system state of the composition. The set of possible configurations is $Turn \times \mathcal{Q}_{P,FB,L,S}$, where we introduce an enumerated type $Turn = \{\texttt{prog}, \texttt{env}\}$ for modeling whether it is the turn of the environment or the reactive module to move. The transition is defined as follows:

- There is a transition from $(\texttt{prog}, p, fb, l, s)$ to $\left(\texttt{env}, p, fb', l, s'\right)$ if $(s, p, s', fb') \in [\![T_{RM}]\!]$
- There is a transition from $(\texttt{env}, p, fb, l, s)$ to $(\texttt{prog}, p', fb, l', s)$ if $((l, p), fb, (l', p')) \in [\![T_{Env}]\!]$

The initial set of configurations are composed of the initial states of *Env* and *RM*, and the system starts from the `prog` turn so that *RM* produces the first feedback fb_0 based on the initial percept p_0. Formally, the initial set of configurations can be defined as a predicate $Init_{Sys}$ in Formula (2.4):

$$[\![Init_{Sys}]\!] \stackrel{\texttt{def}}{=} \{(\texttt{prog}, p_0, fb_0, l_0, s_0) \mid (l_0, p_0) \in [\![Init_{Env}]\!] \wedge s_0 \in [\![Init_{RM}]\!]\} \tag{2.4}$$

Finally, the execution of the system is a sequence of configurations alternating between the `prog` turn and the `env` turn following the above definition. That is, given an initial configuration $(\texttt{prog}, p_0, fb_0, l_0, s_0) \in [\![Init_{Sys}]\!]$, an execution α is a sequence of configurations

show in Formula (2.5):

$$\begin{aligned}\alpha \stackrel{\texttt{def}}{=} &(\texttt{prog}, p_0, fb_0, l_0, s_0) \xrightarrow{RM} (\texttt{env}, p_0, fb_1, l_0, s_1) \xrightarrow{Env} \\ &(\texttt{prog}, p_1, fb_1, l_1, s_1) \xrightarrow{RM} (\texttt{env}, p_1, fb_2, l_1, s_2) \xrightarrow{Env} \\ &(\texttt{prog}, p_2, fb_2, l_2, s_2) \cdots\end{aligned} \tag{2.5}$$

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 [19]. Such models are also standard for hybrid automata, where computation is faster than physical movements [20].

Example 2.2. In this example, we consider a simplistic autonomous system driving a vehicle back and forth between a home base and a work location. We let $p_h \in \mathbb{R}$ denote the position of the home base and $p_w \in \mathbb{R}$ denote the position of the work location. Here we first give an example environment *Env* based on Example 2.1 with the two variables, `pos` representing the position and `clk` to get the current time. We let the environment only provides the vehicle position for perception, that is, $P = \{\texttt{pos}\}$, and there is a clock variable modeling the current time, i.e., $L = \{\texttt{clk}\}$. We consider the feedback to drive the vehicle is specified as a target position `tgt`, so the set of feedback variables $FB = \{\texttt{tgt}\}$. We can specify the initial predicate as Formula (2.6):

$$Init_{Env} \stackrel{\texttt{def}}{=} (\texttt{pos}{=}p_h) \tag{2.6}$$

and it defines the set of initial valuations as Formula (2.7):

$$[\![Init_{Env}]\!]_{L,P} \stackrel{\texttt{def}}{=} \{(\{\texttt{clk}{\rightarrow}\ t\},\ \{\texttt{pos}{\rightarrow}\ p_h\}) \mid t \in \mathbb{R}_{\geq 0}\} \tag{2.7}$$

which specifies that the vehicle starts at the home base with no initial velocity, and the clock may be at any time.

The transition relation $[\![T_{Env}]\!]_{L,P,FB,L',P'}$ is then used to model how the position and clock evolves given the target position. For illustration purposes, we consider a simplistic example model as follows. (1) The reading of the clock is strictly increasing, (2) the vehicle stops when reach the target, and (3) the vehicle moving towards the target may pass beyond the target position up to a maximum *percentage overshoot* ρ, the ratio of the overshoot distance divided by the distance to the target position. We can formalize this simple model as the

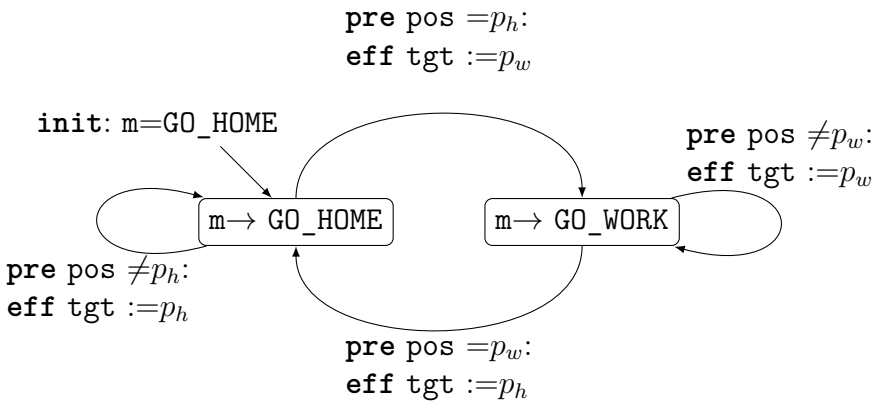


Figure 2.2: State machine of the reactive module RM in Example 2.2 based on the notations in the textbook [84]. Nodes represent discrete modes of the system. A node representing the initial mode is annotated with the initial condition (**init**: ...). Edges represent transitions between two modes. Each edge is annotated with a precondition (**pre** ...:) specifying the enabling condition of the transition, and an effect (**eff** ...) setting new values to variables.

predicate in Formula (2.8):

$$T_{Env} \stackrel{\text{def}}{=} \left(\begin{array}{l} \texttt{clk} < \texttt{clk}' \wedge \\ (\texttt{tgt} = \texttt{pos} \Rightarrow \texttt{pos}' = \texttt{pos}) \wedge \\ (\texttt{tgt} \neq \texttt{pos} \Rightarrow 0 \leq \frac{\texttt{pos}' - \texttt{pos}}{\texttt{tgt} - \texttt{pos}} \leq (1+\rho)) \end{array} \right) \tag{2.8}$$

which states that (1) the value of `clk` is increasing, (2) if the vehicle is already at the target position `tgt`, the position stays the same (`pos`′=`pos`), and (3) the ratio between the actual position change (`pos`′−`pos`) and the desired position change (`tgt`−`pos`) is between 0 to $(1+\rho)$, the maximum change due to overshooting.

We now describe the reactive module RM driving the vehicle back and forth between the home base and the work location. Figure 2.2 illustrates the logic of RM as a state machine. At a high level, RM is alternating between two modes, going to the home base (`GO_HOME`) and going to the work location (`GO_WORK`). In the `GO_HOME` mode, RM checks if the vehicle has reached the position of the home base or not. If it has not reached yet (**pre** `pos`≠p_h), it repeatedly sets the target position to be the home base (**eff** `tgt` :=p_h). Otherwise (**pre** `pos`=p_h), it transits to `GO_HOME` mode, and it sets the target position to be the work location (**eff** `tgt` :=p_w). The transitions from the `GO_WORK` mode are similar, except the home base and work location are swapped. Formally, we first identify the set of variables. RM can perceive the current position of the vehicle `pos` and set a target position `tgt` as the feedback. In addition, it uses a state variable `m` with an enumerate type {`GO_HOME`, `GO_WORK`} to denote whether the vehicle is going to the home base or the work

location. Therefore, the sets of variables are state variables, $S = \{\texttt{m}\}$, percept variables, $P = \{\texttt{pos}\}$, and feedback variables, $FB = \{\texttt{tgt}\}$.

As shown in Figure 2.2, the initial set of states of RM is defined by the predicate $Init_{RM}$ in Formula (2.9):

$$Init_{RM} \stackrel{\texttt{def}}{=} (\texttt{m} = \texttt{GO\_HOME}) \tag{2.9}$$

which specifies the set of initial valuations $[\![Init_{RM}]\!]_S$ as the singleton set $\{\{\texttt{m} \to \texttt{GO\_HOME}\}\}$ with the only valuation $\{\texttt{m} \to \texttt{GO\_HOME}\}$.

We then need to translate the reactive module RM to a predicate T_{RM} to encode the transition relation $[\![T_{RM}]\!]_{S,P,S',FB'}$. We skip the detail of this translation here, and we will discuss the translation for reactive modules implemented in our Koord language [32] in Chapter 4. Following the precondition (**pre**) and effect (**eff**) of the edges in Figure 2.2, T_{RM} can be expressed as the predicate in Formula (2.10):

$$T_{RM} \stackrel{\texttt{def}}{=} \bigwedge \begin{pmatrix} (\texttt{m}{=}\texttt{GO\_HOME} \wedge \texttt{pos}{\neq}p_h) \Rightarrow (\texttt{m}'{=}\texttt{GO\_HOME} \wedge \texttt{tgt}'{=}p_h) \\ (\texttt{m}{=}\texttt{GO\_HOME} \wedge \texttt{pos}{=}p_h) \Rightarrow (\texttt{m}'{=}\texttt{GO\_WORK} \wedge \texttt{tgt}'{=}p_w) \\ (\texttt{m}{=}\texttt{GO\_WORK} \wedge \texttt{pos}{\neq}p_w) \Rightarrow (\texttt{m}'{=}\texttt{GO\_WORK} \wedge \texttt{tgt}'{=}p_w) \\ (\texttt{m}{=}\texttt{GO\_WORK} \wedge \texttt{pos}{=}p_w) \Rightarrow (\texttt{m}'{=}\texttt{GO\_HOME} \wedge \texttt{tgt}'{=}p_h) \end{pmatrix} \tag{2.10}$$

Finally, we give an example execution of the composition of RM and Env. We choose $(\texttt{prog}, \{\texttt{pos} \to p_h\}, \{\texttt{tgt} \to p_h\}, \{\texttt{clk} \to 0\}, \{\texttt{m} \to \texttt{GO\_HOME}\})$ as the initial configuration which satisfies $\{\texttt{m} \to \texttt{GO\_HOME}\} \in [\![Init_{RM}]\!]_S$ and $(\{\texttt{clk} \to 0\}, \{\texttt{pos} \to p_h\}) \in [\![Init_{Env}]\!]_{L,P}$ at the same time. We also use the following values, the home location $p_h = 0$, the work location $p_w = 100$, and the percentage overshoot $\rho = 0.05$. An execution starting from this initial configuration is given as Formula (2.11) below:

$$\begin{aligned}
& (\texttt{prog}, \{\texttt{pos} \to 0\}, \{\texttt{tgt} \to 0\}, \{\texttt{clk} \to 0\}, \{\texttt{m} \to \texttt{GO_HOME}\}) \\
\xrightarrow{RM}\ & (\texttt{env}, \{\texttt{pos} \to 0\}, \{\texttt{tgt} \to 100\}, \{\texttt{clk} \to 0\}, \{\texttt{m} \to \texttt{GO_WORK}\}) \\
\xrightarrow{Env}\ & (\texttt{prog}, \{\texttt{pos} \to 105\}, \{\texttt{tgt} \to 100\}, \{\texttt{clk} \to 0.1\}, \{\texttt{m} \to \texttt{GO_WORK}\}) \\
\xrightarrow{RM}\ & (\texttt{env}, \{\texttt{pos} \to 105\}, \{\texttt{tgt} \to 100\}, \{\texttt{clk} \to 0.1\}, \{\texttt{m} \to \texttt{GO_WORK}\}) \\
\xrightarrow{Env}\ & (\texttt{prog}, \{\texttt{pos} \to 100\}, \{\texttt{tgt} \to 100\}, \{\texttt{clk} \to 1\}, \{\texttt{m} \to \texttt{GO_WORK}\}) \\
\xrightarrow{RM}\ & (\texttt{env}, \{\texttt{pos} \to 100\}, \{\texttt{tgt} \to 0\}, \{\texttt{clk} \to 1\}, \{\texttt{m} \to \texttt{GO_HOME}\}) \\
\xrightarrow{Env}\ & (\texttt{prog}, \{\texttt{pos} \to 0\}, \{\texttt{tgt} \to 0\}, \{\texttt{clk} \to 10\}, \{\texttt{m} \to \texttt{GO_HOME}\}) \\
\xrightarrow{RM}\ & (\texttt{env}, \{\texttt{pos} \to 0\}, \{\texttt{tgt} \to 100\}, \{\texttt{clk} \to 10\}, \{\texttt{m} \to \texttt{GO_WORK}\}) \\
\xrightarrow{Env}\ & \ldots
\end{aligned} \tag{2.11}$$

Notice that the feedback variable tgt and the state variable m are updated with new values

only after each program turn transition of RM, and the latent and percept variables, `clk` and `pos`, are updated only after each environment turn transition of *Env*. We further observe that the execution can be very unrealistic. For example, if we look at the changes on the clock variable `clk`, the duration for traveling to a target position can be arbitrarily short or long. This is because our simplistic environment model only requires that the value of the clock to strictly increase.

Example 2.2 shows how to model an autonomous driving system with CPReact, i.e., the interaction between a reactive module and a simple environment. In Section 2.3, we will discuss how to analyze the safety of the CPReact model based on induction and apply the analysis on the system in Example 2.2. In addition, we see that the execution of this example system can be unrealistic because of the simple environment model, so in Chapter 3, we will discuss how to incorporate and analyze with more realistic environment models.

2.3 SYSTEM-LEVEL SAFETY ASSURANCE VIA INDUCTIVE INVARIANCE

The problem of assuring safety of the system can be stated as follows: given an autonomous system *Sys* on a state space $\mathcal{Q}_{Sys}$, we would like to check that all *reachable configurations* from initial configurations of the system preserve a safety specification *Safe*. For example, a common kind of safety requirement for autonomous vehicles is the *geofencing* property which requires the vehicle to always stay within regions of interest, i.e., geofences.

More precisely, let us denote the set of reachable configurations as *Reach*.[1] For every configuration $(turn, p, fb, l, s) \in Reach$, the system is said to satisfy a safety predicate *Safe* if $(p, s) \in [\![Safe]\!]_{P,S}$ holds. Note that the predicate *Safe* is defined over only percept variables P of *Env* and state variables S of RM. This is to constrain over only variables observable from RM and avoid using latent variables of the autonomous systems. In general, we may also choose to include sets of latent and feedback variables for defining the safety predicate.

Example 2.3. In this example, we specify a safety requirement for the autonomous system in Example 2.2 in Section 2.2. For simplicity, we consider a geofence defining a region of 1D positions $[p_{lo}, p_{hi}]$ with a lower bound position $p_{lo} \in \mathbb{R}$ and an upper bound position $p_{hi} \in \mathbb{R}$. We assume the geofence should cover the home and work locations, p_h and p_w. Especially, the geofence should not be violated even when the vehicle overshoots. Formally, given the percentage overshoot $0 \leq \rho < 1$, we require $p_h, p_w \in [\frac{p_{lo}+\rho \cdot p_{hi}}{1+\rho}, \frac{\rho \cdot p_{lo}+p_{hi}}{1+\rho}]$. The safety predicate

[1] The set of reachable configuration *Reach* can be very complex and may not be expressible as a predicate over configurations.

Safe of this geofence is expressed as Formula (2.12):

$$Safe \stackrel{\texttt{def}}{=} (p_{lo} \leq \texttt{pos} \leq p_{hi}) \tag{2.12}$$

which defines the set of valuations over the percept variables $P = \{\texttt{pos}\}$ and state variables $S = \{\texttt{m}\}$ as Formula (2.13):

$$[\![Safe]\!]_{P,S} \stackrel{\texttt{def}}{=} \left\{ (\{\texttt{pos} \to p\}, \{\texttt{m} \to mode\}) \middle| \begin{array}{l} p_{lo} \leq p \leq p_{hi} \\ \wedge\ mode \in \{\texttt{GO\_HOME}, \texttt{GO\_WORK}\} \end{array} \right\} \tag{2.13}$$

The safety property effectively requires the vehicle to stay within the geofence no matter when it is going home or going to work.

It is in general hard to check whether a given predicate *Safe* is preserved in all reachable configurations. We will describe the two main proof techniques used across the thesis, namely induction and abstraction refinement, and we derive specialized proof tactics for our CPReact model described in Section 2.2. We will focus on the induction proofs using inductive invariant in this section, and talk about abstraction and refinement based on simulation relations in Chapter 3.

The typical approach to prove the safety specification is to find a stronger notion of *inductive invariant* that is easier to verify. An inductive invariant for proving safety is a predicate *Inv* that satisfies the following properties:

- It includes all the initial configurations, $[\![Init_{Sys}]\!] \subseteq [\![Inv]\!]$.
- All configurations in the invariant preserve the safety specification, i.e., for every configuration $(turn, p, fb, l, s) \in [\![Inv]\!]$, the percept value and the state of RM, $(p, s) \in [\![Safe]\!]_{P,S}$ holds.
- For every configuration $(turn, p, fb, l, s)$ satisfying *Inv*, i.e., $(turn, p, fb, l, s) \in [\![Inv]\!]$, for every next configuration $(turn', p', fb', l', s')$ that $(turn, p, fb, l, s)$ can transition to, $(turn', p', fb', l', s') \in [\![Inv]\!]$ also holds.

It is easy to see (by induction on the length of executions) that *Reach* is always a subset of the invariant $[\![Inv]\!]$, and the system is proven safe since all configurations in $[\![Inv]\!]$ are safe. We summarize the above and provide a theorem with the set of configurations, and an equivalent formulation using predicates and expressions.

Theorem 2.1. *Given a reactive module RM with the environment model Env, a predicate $Init_{Sys}$ specifying the initial set of configurations, and a safety predicate Safe over percept*

and state variables of RM, an inductive invariant *Inv proves the safety Safe if for any* $turn \in \{\texttt{prog}, \texttt{env}\}, p, p' \in \mathcal{Q}_P, fb, fb' \in \mathcal{Q}_{FB}, l, l' \in \mathcal{Q}_L$ *and* $s, s' \in \mathcal{Q}_S$*, we can prove the validity of all the following proof obligations (POs):*

$$(turn, p, fb, l, s) \in [\![Init_{Sys}]\!] \implies (turn, p, fb, l, s) \in [\![Inv]\!] \tag{Init}$$

$$(turn, p, fb, l, s) \in [\![Inv]\!] \implies (p, s) \in [\![Safe]\!]_{P,S} \tag{Safe}$$

$$(\texttt{prog}, p, fb, l, s) \in [\![Inv]\!] \wedge (s, p, s', fb') \in [\![T_{RM}]\!] \implies (\texttt{env}, p, fb', l, s') \in [\![Inv \wedge Aux]\!] \tag{IndProg}$$

$$(\texttt{env}, p, fb, l, s) \in [\![Inv \wedge Aux]\!] \wedge ((p, l), fb, (p', l')) \in [\![T_{Env}]\!] \implies (\texttt{prog}, p', fb, l', s) \in [\![Inv]\!] \tag{IndEnv}$$

where Aux can be any predicate over configurations.

Proof. PO (*Init*) and PO (*Safe*) are directly translated from proving initial configurations and safety. We further break down the property of the inductive invariant according to the transition relations of *RM* and *Env*.

- For every configuration $(\texttt{prog}, p, fb, l, s) \in [\![Inv]\!]$ and for every program transition $(s, p, s', fb') \in [\![T_{RM}]\!]$, every next configuration $(\texttt{env}, p, fb', l, s') \in [\![Inv \wedge Aux]\!]$ holds.
- For every configuration $(\texttt{env}, p, fb, l, s) \in [\![Inv \wedge Aux]\!]$ and for every environment transition $((p, l), fb, (p', l')) \in [\![T_{Env}]\!]$, every next configuration $(\texttt{prog}, p', fb, l', s) \in [\![Inv]\!]$ holds.

The first ensures that $[\![Inv \wedge Aux]\!]$ holds at the end of the program turn, i.e., the beginning of the environment turn. Because $[\![Inv \wedge Aux]\!]$ is equivalent to $[\![Inv]\!] \cap [\![Aux]\!]$ by expanding the definition, $[\![Inv]\!] \cap [\![Aux]\!]$ is a subset of $[\![Inv]\!]$; thus the invariant $[\![Inv]\!]$ holds. The second ensures that, starting from any configuration in $[\![Inv \wedge Aux]\!]$, $[\![Inv]\!]$ holds after the environment turn. Therefore, proving (*IndProg*) and (*IndEnv*) is sufficient to show that *Inv* is an inductive invariant. QED.

Remark 2.1. Observe that the auxiliary predicate *Aux* helps strengthen the invariant *Inv* after the program turn transitions. This allows us to encode additional constraints based on our understanding of the reactive module *RM*. We will explain and show the usefulness of the auxiliary predicate *Aux* later in Example 2.4.

Corollary 2.2. *The proof obligations in Theorem 2.1 are valid if and only if all predicates below are* unsatisfiable*, that is, there is no valuation so that the predicates evaluate to true:*

$$Init_{Sys} \wedge \neg Inv \tag{InitUnsat}$$

$$Inv \wedge \neg Safe \tag{SafeUnsat}$$

$$Inv \wedge T_{RM} \wedge \neg(Inv \wedge Aux)[S'/S, FB'/FB] \tag{IndProgUnsat}$$

$$Inv \wedge Aux \wedge T_{Env} \wedge \neg Inv[L'/L, P'/P] \tag{IndEnvUnsat}$$

Proof. This is a direct result by reducing the validity of each proof obligation to the satisfiability of the negation of the proof obligation. QED.

Theorem 2.1 divides the induction proof into four proof obligations, and Corollary 2.2 provides a method to discharge these four proof obligations with satisfiability solvers, such as Satisfiability Modulo Theories (SMT) solvers [85, 86]. Theorem 2.1 has two important features: (1) separated proof obligations for the program turn PO (*IndProgUnsat*) and the environment turn PO (*IndEnvUnsat*), and (2) an auxiliary predicate *Aux* to strengthen the invariant after the program turn. The first feature is possible because the reactive module *RM* and the environment *Env* update disjoint sets of variables. In the program turn transition, the values of percept and latent variables for *Env* remain unchanged, and hence only state variables S and feedback variables *FB* are replaced. Similarly, in the environment turn transition, the values of state and feedback variables for *RM* remain unchanged, and only latent variables L and percept variables P are replaced. As a result, the satisfiability query to discharge PO (*IndProgUnsat*) is independent of the query to discharge PO (*IndEnvUnsat*). The second feature is from the observation that *RM* guides the entire system towards safer configurations in general. Hence, after the program transition, we may require that the system configuration is a subset of safer configurations more restricted than the invariant, and this in turn can help prove that the invariant holds in the environment turn. We will show in Example 2.4 how PO (*IndProgUnsat*) and PO (*IndEnvUnsat*) are checked separately, and how the auxiliary predicate helps prove the proof obligations.

Example 2.4. In this example, we verify the autonomous system in Example 2.2 against the geofence property defined in Example 2.3. We consider the invariant candidate *Inv* that is the same as the geofence property in Formula (2.12), and we specify it as Formula (2.14):

$$Inv \stackrel{\texttt{def}}{=} (p_{lo} \leq \texttt{pos} \leq p_{hi}) \tag{2.14}$$

which requires the position of the vehicle stays within the geofence. We then check the four proof obligations for in Theorem 2.1. Here, we apply Corollary 2.2 and manually check the unsatisfiability.

We first instantiate the proof obligation PO (*InitUnsat*) as Formula (2.15):

$$(\texttt{vel}{=}0 \wedge \texttt{pos}{=}p_h \wedge \texttt{m}{=}\texttt{GO\_HOME}) \wedge \neg\, (p_{lo} \leq \texttt{pos} \leq p_{hi}) \tag{2.15}$$

Because $p_h \in [\frac{p_{lo}+\rho \cdot p_{hi}}{1+\rho}, \frac{\rho \cdot p_{lo}+p_{hi}}{1+\rho}]$, the predicate $\texttt{pos}{=}p_h$ contradicts with $\neg(p_{lo} \leq \texttt{pos} \leq p_{hi})$. As a result, there is no valuation for pos, and we derive that the predicate is unsatisfiable.

The proof obligation PO (*SafeUnsat*) is unsatisfiable because *Inv* is the same as the geofence *Safe*; thus $Safe \wedge \neg Inv$ is trivially unsatisfiable. To prove the remaining two proof obligations, PO (*IndProgUnsat*) and PO (*IndEnvUnsat*), we use the following auxiliary predicate in Formula (2.16):

$$Aux \stackrel{\texttt{def}}{=} (\texttt{tgt}{=}p_h \vee \texttt{tgt}{=}p_w) \tag{2.16}$$

which indicates the assigned target position is either the home or work location.

With the auxiliary predicate *Aux*, the proof obligation for the program turn transition, PO (*IndProgUnsat*), is the following predicate in Formula (2.17):

$$\begin{array}{ll}
 & p_{lo} \leq \texttt{pos} \leq p_{hi} \quad \cdots Inv \\
\hline
\wedge & (\texttt{m}{=}\texttt{GO_HOME} \wedge \texttt{pos}{\neq}p_h) \Rightarrow (\texttt{m}'{=}\texttt{GO_HOME} \wedge \texttt{tgt}'{=}p_h) \\
\wedge & (\texttt{m}{=}\texttt{GO_HOME} \wedge \texttt{pos}{=}p_h) \Rightarrow (\texttt{m}'{=}\texttt{GO_WORK} \wedge \texttt{tgt}'{=}p_w) \quad \cdots T_{RM} \\
\wedge & (\texttt{m}{=}\texttt{GO_WORK} \wedge \texttt{pos}{\neq}p_w) \Rightarrow (\texttt{m}'{=}\texttt{GO_WORK} \wedge \texttt{tgt}'{=}p_w) \\
\wedge & (\texttt{m}{=}\texttt{GO_WORK} \wedge \texttt{pos}{=}p_w) \Rightarrow (\texttt{m}'{=}\texttt{GO_HOME} \wedge \texttt{tgt}'{=}p_h) \\
\hline
\wedge & \neg\, (p_{lo} \leq \texttt{pos} \leq p_{hi} \wedge (\texttt{tgt}'{=}p_h \vee \texttt{tgt}'{=}p_w)) \quad \cdots \neg(Inv \wedge Aux)[S'/S, FB'/FB]
\end{array} \tag{2.17}$$

We can prove Formula 2.17 is unsatisfiable by examining every clause inside the predicate $(Inv \wedge Aux)[S'/S, FB'/FB]$. The invariant $p_{lo} \leq \texttt{pos} \leq p_{hi}$ is trivially preserved because the pos variable is not updated during the program turn transition. The auxiliary predicate $(\texttt{tgt}'{=}p_h \vee \texttt{tgt}'{=}p_w)$ also holds because the updated value of $\texttt{tgt}'$ is either p_h or p_w in every possible transition in T_{RM}. Therefore, the over all predicate is unsatisfiable with the negation of $(Inv \wedge Aux)[S'/S, FB'/FB]$. Interested readers can also encode the predicate as an SMT formula and query an SMT solver to check the unsatisfiability.[2]

Lastly, we have to inspect the proof obligation for the environment transition. The proof

[2]For SMT solving, additional axioms are required to select p_h and p_w from $[\frac{p_{lo}+\rho \cdot p_{hi}}{1+\rho}, \frac{\rho \cdot p_{lo}+p_{hi}}{1+\rho}]$.

obligation PO (*IndEnvUnsat*) is instantiated as Formula (2.18):

$$
\begin{array}{lll}
 & p_{lo} \leq \texttt{pos} \leq p_{hi} & \cdots Inv \\
\hline
\wedge & (\texttt{tgt}{=}p_h \vee \texttt{tgt}{=}p_w) & \cdots Aux \\
\hline
\wedge & \texttt{clk} < \texttt{clk}' & \\
\wedge & \texttt{tgt} = \texttt{pos} \Rightarrow \texttt{pos}' = \texttt{pos} & \cdots T_{Env} \\
\wedge & \texttt{tgt} \neq \texttt{pos} \Rightarrow \ 0 \leq \frac{\texttt{pos}' - \texttt{pos}}{\texttt{tgt} - \texttt{pos}} \leq (1+\rho) & \\
\hline
\wedge & \neg(p_{lo} \leq \texttt{pos}' \leq p_{hi}) & \cdots \neg Inv[L'/L, P'/P]
\end{array}
\tag{2.18}
$$

We argue the unsatisfiability by considering the worst value of the new position $\texttt{pos}'$. The worst case scenario happens when the vehicle has to travel the furthest distance with the maximum overshoot. Formally, from the invariant *Inv*, the value of the old position $\texttt{pos}$ is within $[p_{lo}, p_{hi}]$. From the auxiliary predicate *Aux*, the value of the target position $\texttt{tgt}$ is either p_h or p_w, and both p_h and p_w are within $[\frac{p_{lo}+\rho \cdot p_{hi}}{1+\rho}, \frac{\rho \cdot p_{lo}+p_{hi}}{1+\rho}]$. We therefore choose the worst value of the old position $\texttt{pos}$ to be p_{lo}, and the value of $\texttt{tgt}$ to be $\frac{\rho \cdot p_{lo}+p_{hi}}{1+\rho}$.

The value of the new position $\texttt{pos}'$ is constrained by the maximum percentage overshoot and derived as Formula (2.19):

$$
\begin{array}{lll}
 & 0 \leq \dfrac{\texttt{pos}' - \texttt{pos}}{\texttt{tgt} - \texttt{pos}} \leq (1+\rho) & \cdots \text{Evaluate with } \{\texttt{pos} \rightarrow p_{lo},\ \texttt{tgt} \rightarrow \dfrac{\rho \cdot p_{lo} + p_{hi}}{1+\rho}\} \\
\iff & 0 \leq \dfrac{\texttt{pos}' - p_{lo}}{\frac{\rho \cdot p_{lo}+p_{hi}}{1+\rho} - p_{lo}} \leq (1+\rho) & \cdots \text{Divide by } 1+\rho \\
\iff & 0 \leq \dfrac{\texttt{pos}' - p_{lo}}{p_{hi} - p_{lo}} \leq 1 & \\
\iff & p_{lo} \leq \texttt{pos}' \leq p_{hi} &
\end{array}
\tag{2.19}
$$

We can see Formula (2.19) contradicts with $\neg Inv[L'/L, P'/P]$ in Formula (2.18). Hence, we have shown the unsatisfiability.

Notice that the target position $\texttt{tgt}$ is constrained in the auxiliary predicate *Aux* and not in the invariant *Inv*. As a result, if the auxiliary predicate *Aux* is not used, the proof now has to show *Inv* holds even when the specified target position $\texttt{tgt}$ is out of the geofence. We know it is impossible to still ensure the invariant under these unreasonable target positions, and hence it shows the importance of encoding the constraints guaranteed by the program turn transition as the auxiliary predicate *Aux*.

We have demonstrated in Example 2.4 how to systematically apply our induction proof by solving the four proof obligations in Theorem 2.1. In particular, we use Corollary 2.2 to automatically discharge proof obligations with the help of existing satisfiability solvers.

Overall, we have shown how to use an inductive invariant to verify the safety of a CPReact model composed of a reactive module RM and an environment model Env.

2.4 SUMMARY

In this chapter, we presented an overview of the CPReact framework to model an autonomous system as the interaction between a physical environment and a reactive module and analyze the safety of the system via induction proofs. We highlighted how our approach decomposes the safety verification of the system into independent proof obligations for the reactive module and the physical environment, and we outlined the formal reasoning that allow us to apply satisfiability solving for the induction proof on the program turn and the environment turn transitions.

We walked through the modeling and analyses of an illustrative autonomous vehicle system, provided safety assurance with respect to a simple environment model. We have also identified a major concern that the environment model such as the one in Example 2.2 is unrealistic. The safety proof for the simplistic environment may not provide any guarantee over a realistic environment. To address this concern, we study how to connect a simpler environment model with a more realistic environment model and in turn extend the safety guarantee to the more realistic environment in Chapter 3.

Chapter 3: Safe Abstractions of Real Environments

In CPREACT, we assume that a model of the environment *Env* is available and expressible as a predicate T_{Env} to encode the transition relation $[\![T_{Env}]\!]_{L,P,FB,L',P'}$ over the latent variables L, the percept variables P, the feedback variables FB, and the primed versions of latent and percept variables L' and P'. However, recall our autonomous example in Section 2.2, we use the model of the environment *Env* to enclose the components such as the car's hardware platform, perception algorithms, the road, and other external factors that may influence the behavior of the car, and we have discussed in Section 1.1 why extracting a tractable formal model *Env* directly from the real environment can be extremely challenging. Thus, the classical approach of proving the safety of systems that interact with the environment proceeds in two steps:

(1) Prove the simpler *abstract model* of the environment, *Env*, *simulates* all possible behaviors of the real world environment *REnv*.

(2) Prove the safety of the system with respect to this model of the environment *Env*.

The completion of this proof shows that the system interacting with *any* environment *REnv* that can be simulated by *Env* will be safe. In Chapter 2, we have discussed how to prove the safety with respect to an environment model *Env*. The model *Env* however may not simulate the real world environment *REnv*. Another direction is to observe the behaviors of the real world environment *REnv* and construct an abstract model *Env*, but this model *Env* may not guarantee the safety of the system. It is therefore crucial to find a *safe abstraction*, i.e., an abstract model of the real world environment *REnv* that proves the system-level safety at the same time.

In this chapter, we hence focus on finding a safe abstraction of the real-world physical environment. In Section 3.1, we formally define the condition when a model of environment is an abstraction of a physical environment, and show the safety in the model of environment implies the safety in the physical environment. In Section 3.2, we define the synthesis problem for *safe abstractions* of the physical environment, and outline an approach to find safe abstractions using reachability analyses.

3.1 SYSTEM SAFETY USING ABSTRACTIONS OF ENVIRONMENTS

Now we formally outline the technique for proving systems safe against the real environment *REnv*. The real environment *REnv* has precisely the same structure as the en-

vironment model *Env* described in Section 2.2. It has a set of latent variables LR of the real environment, an initial set of states $\Theta_{REnv} \subseteq \mathcal{Q}_{LR,P}$, and a transition relation $\mathcal{T}_{REnv} \subseteq \mathcal{Q}_{LR,P,FB,LR',P'}$. Typically, the set of latent variables and transitions of the model are much simpler than real world environments, and can be seen as *assumptions* made of the real world. In the automated vehicle context, for example, vehicle dynamics may model nondeterministic skidding of a vehicle up to some degree (without modeling the wetness of the road precisely), making formal reasoning much easier. To explicitly show this distinction between *Env* and *REnv*, we use $[\![Init_{Env}]\!]$ and $[\![T_{Env}]\!]$ for *Env* which are expressible as predicates $Init_{Env}$ and T_{Env}, and we use Θ_{REnv} and $\mathcal{T}_{REnv}$ for *REnv* which are not necessarily expressible as predicates.

Consider an environment model *Env* with variables L and P interacting with a system, and assume that the system interacting with this model *Env* (over variables L and P) satisfies a safety condition $[\![Safe]\!]_{P,S}$, established using an invariant $[\![Inv]\!]$. Now under conditions that relate the model of the environment *Env* and the real environment *REnv*, namely a *simulation relation*, we can argue that the system working with the real environment *REnv* will continue to be safe. If we can find such a simulation relation, we say *Env simulates REnv*, *Env* is an *abstraction* of *REnv*, and *REnv* is a *refinement* of *Env*. We formally define the simulation relation in Definition 3.1.

Definition 3.1. A relation $\sim$ between the states of the environment $\mathcal{Q}_{LR,P}$ and the states of the environment model $\mathcal{Q}_{L,P}$ is a *simulation relation* if the following three conditions hold. (below, $lr, lr' \in \mathcal{Q}_{LR}$, $l, l' \in \mathcal{Q}_L$, and $p, p' \in \mathcal{Q}_P$).

- If $p = p'$, then $(lr, p) \sim (l, p')$.
- For every initial state of the real environment, there is an initial state of the model of the environment that it is related to, i.e., for every lr, p, if $(lr, p) \in \Theta_{REnv}$ holds, there is some l such that $(l, p) \in [\![Init_{Env}]\!]$ holds and $(lr, p) \sim (l, p)$.
- Let $(lr, p) \sim (l, p)$, let fb be feedback, and let $(lr, p, fb, lr', p') \in \mathcal{T}_{REnv}$ hold. Then there is some l' such that $(l, p, fb, l', p') \in [\![T_{Env}]\!]$ holds and $(lr', p') \sim (l', p')$.

The first condition says that states of the real environment and the environment model must share the same perception valuations. The second demands that every initial state of the environment is related to some initial state of the environment model. And the third demands that from any pair of states that are similar, the environment model should be able to simulate every move of the environment, and reach similar states.

Theorem 3.1. *Let a reactive module RM with the environment model Env satisfy an invariant Inv, which in turn proves a property Safe. Let REnv be a real environment that Env simulates REnv. Then the environment REnv working with the system RM is guaranteed to preserve the property Safe.*

Proof. Let $\sim$ be a simulation relation between *Env* and *REnv*. We can show by induction that on n that for every reachable configuration $(turn, p, fb, lr, s)$ reached with *REnv* in n steps, there is a reachable configuration $(turn, p, fb, l, s)$ such that $(lr, p) \sim (l, p)$. The base case is easy, and the induction step involving a system move as the last move are trivial. For the induction step involving an environment move as the last move, the fact that the simulation relation guarantees a move of the environment model that simulates the real environment ensures the property. QED.

Theorem 3.1 motivates finding an environment model *Env* and then proving *Env* simulates the real environment *REnv*. However, searching for a simulation relation between arbitrary *Env* and *REnv* is difficult in general; we therefore look for a sufficient condition on the structure of *Env* and *REnv* so that a simple simulation relation is guaranteed. We restrict that latent variables of *Env* must also be used by the real environment *REnv*, i.e. $LR = L \cup R$. where R is the set of latent variables only for *REnv*, and the valuations of *REnv* is denoted as $lr = (l, r) \in \mathcal{Q}_{L,R}$. Under this restricted setting, we provide a sufficient condition as proof obligations in Proposition 3.2.

Proposition 3.2. *Let Env be an environment model with latent variables L, percept variables P, initial states $[\![Init_{Env}]\!]_{L,P}$ and transition relation $[\![T_{Env}]\!]_{L,P,FB,L'P'}$. Let REnv be a real environment with latent variables $LR = L \cup R$ and $L \cap R = \emptyset$, the same percept variables P, initial states Θ_{REnv}, and the transition relation $\mathcal{T}_{REnv}$. If for any $l, l' \in \mathcal{Q}_L, r, r' \in \mathcal{Q}_R, p, p' \in \mathcal{Q}_P$, and $fb \in \mathcal{Q}_{FB}$, the following two proof obligations are valid:*

$$(l, r, p) \in \Theta_{REnv} \implies (l, p) \in [\![Init_{Env}]\!] \qquad (SimInit)$$

$$(l, r, p, fb, l', r', p') \in \mathcal{T}_{REnv} \implies (l, p, fb, l', p') \in [\![T_{Env}]\!] \qquad (SimTrans)$$

then the model of the environment Env simulates the environment REnv.

Proof. We define a relation $\sim$ such that every state (l, r, p) is related to the state (l, p), i.e, $(l, r, p) \sim (l, p)$. It is straightforward to prove $\sim$ is a simulation relation for *REnv* and *Env* because (1) by construction, whenever $(l, r, p) \sim (l, p')$, $p = p'$, (2) the first proof obligation ensures that all initial states are simulated, and (3) the second proof obligation ensures that, for any feedback *fb* and every *REnv* transition from (l, r, p) to (l', r', p'), there is an

Env transition from (l, p) to the new state (l', p') which simulates (l', r', p') by construction. QED.

Corollary 3.3. *Given two models of environments Env_1 and Env_2 sharing the same set of latent variables L, let $Env_1 \wedge Env_2$ denote the model of environment with the initial predicate $Init_{Env_1} \wedge Init_{Env_2}$ and the transition relation predicate $T_{Env_1} \wedge T_{Env_2}$, the model $Env_1 \wedge Env_2$ is an abstraction of $\mathcal{T}_{REnv}$ if and only if both Env_1 and Env_2 are abstractions of $\mathcal{T}_{REnv}$.*

Proof. Following Proposition 3.2, the proof obligations for Env_1 and Env_2 combined are equivalent to the proof obligations for $Env_1 \wedge Env_2$. QED.

For demonstration, we consider a slightly more realistic environment model as the real environment *REnv*, and compare it with the simple model *Env* from Example 2.2. We will show that *Env* simulates *REnv* using Proposition 3.2, and therefore by Theorem 3.1, the safety of the system is guaranteed even if we replace *Env* with *REnv*.

Example 3.1. In this example, *REnv* shares the same set of percept variables $P = \{\texttt{pos}\}$ for the vehicle position, and the same set of latent variables $L = \{\texttt{clk}\}$ for the current time. In addition, *REnv* has a latent variable $R = \{\texttt{vel}\}$ representing the velocity of the vehicle that is not a latent variable of the model *Env*. We can specify an initial predicate as Formula (3.1):

$$Init_{REnv} \stackrel{\texttt{def}}{=} (\texttt{pos} = p_h \wedge \texttt{vel} = 0) \tag{3.1}$$

and it defines the set in Formula (3.2):

$$\Theta_{REnv} \stackrel{\texttt{def}}{=} [\![Init_{REnv}]\!]_{L,R,P} \stackrel{\texttt{def}}{=} \{(\{\texttt{clk} \to t\}, \{\texttt{vel} \to 0\}, \{\texttt{pos} \to p_h\}) \mid t \in \mathbb{R}_{\geq 0}\} \tag{3.2}$$

which requires the vehicle to start at the home location p_h with no initial velocity. We then define the transition relation of *REnv* following simplified laws of motion below.

(1) The clock increases by at most a period $\Delta T \in \mathbb{R}_{\geq 0}$ for sampling each new position,

(2) The velocity is proportional to the distance to the target with a gain $\kappa \in \mathbb{R}_{\geq 0}$ bounded by $\kappa \leq \frac{1+\rho}{\Delta T}$ to limit the maximum overshoot where ρ is the percentage overshoot.

(3) the change of the vehicle position is the velocity multiplying the elapsed time.

Formally, the transition relation $\mathcal{T}_{REnv} \stackrel{\texttt{def}}{=} [\![T_{REnv}]\!]$ can be specified with a predicate in Formula (3.3):

$$T_{REnv} \stackrel{\texttt{def}}{=} \begin{pmatrix} 0 < \texttt{clk}' - \texttt{clk} \leq \Delta T \wedge \\ \texttt{vel}' = \kappa * (\texttt{tgt} - \texttt{pos}) \wedge \\ \texttt{pos}' = \texttt{pos} + \texttt{vel}' * (\texttt{clk}' - \texttt{clk}) \end{pmatrix} \tag{3.3}$$

We now check the two proof obligations, PO (*SimInit*) and PO (*SimTrans*), to prove *Env* simulates *REnv*. Because Θ_{REnv} and $\mathcal{T}_{REnv}$ can be specified with predicates $Init_{REnv}$ and T_{REnv}, we alternatively prove the following two predicates are both unsatisfiable:

$$Init_{REnv} \wedge \neg Init_{Env} \qquad (SimInitUnsat)$$

$$T_{REnv} \wedge \neg T_{Env} \qquad (SimTransUnsat)$$

Recall from Example 2.2, we instantiate PO (*SimInitUnsat*), namely $Init_{REnv} \wedge \neg Init_{Env}$, as $(\texttt{pos} = p_h \wedge \texttt{vel}{=}0) \wedge \neg(\texttt{pos}{=}p_h)$ and this is trivially unsatisfiable.

We instantiate PO (*SimTransUnsat*) as Formula (3.4):

$$\begin{array}{lll} & 0 < \texttt{clk}' - \texttt{clk} \leq \Delta T & \\ \wedge & \texttt{vel}' = \kappa{*}(\texttt{tgt} - \texttt{pos}) & \cdots T_{REnv} \\ \wedge & \texttt{pos}' = \texttt{pos} + \texttt{vel}'{*}(\texttt{clk}' - \texttt{clk}) & \\ \hline \wedge & \neg \left(\begin{array}{l} \texttt{clk} < \texttt{clk}' \wedge \\ (\texttt{tgt} = \texttt{pos} \Rightarrow \texttt{pos}' = \texttt{pos}) \wedge \\ (\texttt{tgt} \neq \texttt{pos} \Rightarrow 0 \leq \frac{\texttt{pos}' - \texttt{pos}}{\texttt{tgt} - \texttt{pos}} \leq (1+\rho)) \end{array} \right) & \cdots \neg T_{Env} \end{array} \tag{3.4}$$

We prove each clause of T_{Env} separately and complete the proof using Corollary 3.3. We find that $\texttt{clk} < \texttt{clk}'$ is implied by $0 < \texttt{clk}' - \texttt{clk} \leq \Delta T$, and $\texttt{tgt} = \texttt{pos} \Rightarrow \texttt{pos}' = \texttt{pos}$ is valid because the velocity $\texttt{vel}'$ is 0 when $\texttt{tgt} = \texttt{pos}$.

Finally, starting from $\texttt{pos}' = \texttt{pos} + \texttt{vel}'{*}(\texttt{clk}' - \texttt{clk})$ in T_{REnv}, we derive Formula (3.5) when $\texttt{tgt} \neq \texttt{pos}$:

$$\begin{array}{lll} & \texttt{pos}' = \texttt{pos} + \texttt{vel}'{*}(\texttt{clk}' - \texttt{clk}) & \\ \Longleftrightarrow & \texttt{pos}' - \texttt{pos} = \kappa{*}(\texttt{tgt} - \texttt{pos}){*}(\texttt{clk}' - \texttt{clk}) & \because \texttt{tgt} \neq \texttt{pos} \\ \Longrightarrow & \frac{\texttt{pos}' - \texttt{pos}}{\texttt{tgt} - \texttt{pos}} = \kappa{*}\ (\texttt{clk}' - \texttt{clk}') & \because 0 < \texttt{clk}' - \texttt{clk} \leq \Delta T \\ \Longrightarrow & 0 \leq \frac{\texttt{pos}' - \texttt{pos}}{\texttt{tgt} - \texttt{pos}} \leq \kappa * \Delta T & \because 0 \leq \kappa \leq \frac{1+\rho}{\Delta T} \text{ by design} \\ \Longrightarrow & 0 \leq \frac{\texttt{pos}' - \texttt{pos}}{\texttt{tgt} - \texttt{pos}} \leq (1+\rho) & \end{array} \tag{3.5}$$

We conclude that the last clause of T_{Env} is also implied by T_{REnv}. Hence, $\neg T_{Env}$ contradicts with T_{REnv}, and PO (*SimTransUnsat*) is unsatisfiable.

Example 3.1 shows that the simple environment model *Env* is the abstraction of a more realistic environment model *REnv* using Proposition 3.2, and following Theorem 3.1, we guarantee the safety of the CPReact system using the real environment *REnv*. This means a simplistic environment model *Env* exhibiting unrealistic executions can still be used in the induction proof as long as it is able to simulate the realistic environment *REnv*. We will

discuss in Section 3.2 how this allows us to incorporate techniques that are suitable for analyzing and abstracting the realistic environment *REnv*.

3.2 SYNTHESIZING SAFE ABSTRACTIONS FOR SAFETY

We now reiterate through the theoretical results presented in previous sections and introduce the synthesis problem for *safe abstractions of environments* in the CPReact framework. Recall from Section 2.3, we assume the reactive module *RM* and the model of physical environment *Env* are given as initial and transition relations predicates for analyzing a CPReact system model, and by Theorem 2.1, showing the safety of the CPReact system requires the model *Env* to preserve the inductive invariant *Inv*, i.e., to show the proof obligation PO (*IndEnv*) holds. Then, to carry over the safety guarantee to a real physical environment *REnv*, the model *Env* must be an abstraction of *REnv* by Theorem 3.1 in Section 3.1. In short, the safety verification under the CPReact framework is to search for a model of the environment *Env* that is a *safe abstraction* of the real environment *REnv*. More precisely, the model *Env* satisfies the following:

- *Safe*: *Env* preserves the inductive invariant *Inv*.[3]
- *Abstraction*: *Env* is an abstraction of the real physical environment *REnv*.

Solving this synthesis problem is in general as difficult as verifying the autonomous system. This thesis focuses on synthesizing safe abstractions for a practical class of environments—continuous-time dynamical systems—which is studied extensively by the control theory community. In the rest of this section, we first provide the definition of continuous-time dynamical systems. We then define the *guard-reachset assumptions* and show that they are abstractions of continuous-time dynamical systems by construction. Lastly, we demonstrate on the autonomous vehicle example using the reachability analysis tool, DryVR [23].

3.2.1 Continuous-Time Dynamical Systems

Given a state space $\mathcal{X} \subseteq \mathbb{R}^n$, an input space $\mathcal{U} \subseteq \mathbb{R}^m$, and an output space $\mathcal{Z} \subseteq \mathbb{R}^k$, a *dynamical system* is defined by a function $f : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$ that is locally Lipschitz in the first argument and a function $h : \mathcal{X} \to \mathcal{Z}$ that calculates the perception output based on the current state. We call f a dynamic function and h an output function. The state and the input of the system will evolve continuously and discretely in time, respectively. Let

[3]For simplicity, we assume the inductive invariant *Inv* is given in the synthesis problem.

$\Delta T \in \mathbb{R}_{\geq 0}$ denote a time bound to define a finite time interval $[0, \Delta T]$, we assume the input will not change during the time interval.

We define $\xi : \mathcal{X} \times \mathcal{U} \times \mathbb{R}_{\geq 0} \to \mathcal{X}$ to be the function that generates *trajectories* of the system. For any $x_0 \in \mathcal{X}$ and $u \in \mathcal{U}$, $\xi(x_0, u, \cdot)$ is the trajectory that starts from the initial state x_0 and follows the input u. The trajectory should satisfy two conditions: (1) $\xi(x_0, u, 0) = x_0$, and (2) For all $t \in \mathbb{R}_{\geq 0}$,

$$\frac{d}{dt}\xi(x_0, u, t) = f(\xi(x_0, u, t), u) \tag{3.6}$$

We say that $\xi(x_0, u, t)$ the state of the system at time t when it starts from x_0 with input u, and $h(\xi(x_0, u, t))$ provides the perceived output of the system at time t. Given an initial state $x_0 \in \mathcal{X}$ and a constant input $u \in \mathcal{U}$, the trajectory $\xi(x_0, u, \cdot)$ exists and is the unique solution of the ordinary differential equation (ODE) in Equation (3.6) since f is locally Lipschitz [87]. Further, the dynamical system is *invariant under time translation* by construction, so given any trajectory starting at any time $t \in \mathbb{R}_{\geq 0}$ in a time interval $[t, t + \Delta T] \subseteq \mathbb{R}_{\geq 0}$ with $x_0 = \xi(x, u, t)$ evolving to $x_1 = \xi(x, u, t + \Delta T)$, there is a trajectory in the time interval $[0, \Delta T]$ with the same initial value $x_0 = \xi(x_0, u, 0)$ and $x_1 = \xi(x_0, u, \Delta T)$. In this thesis, we therefore work with bounded-time solutions of ODEs of a finite time interval $[0, \Delta T]$ with a constant input u. We now define the transition relation of the environment *REnv* based on a given dynamical system.

Definition 3.2. Given a dynamical system of Equation (3.6) over the state space $\mathcal{X}$, the input space $\mathcal{U}$, and the output space $\mathcal{Z}$ with the solution ξ and the output function h, we define an environment *REnv* with two latent variables $\mathtt{x}, \mathtt{clk} \in L$, a percept variable $\mathtt{z} \in P$, and a feedback variable $\mathtt{u} \in FB$. $\mathtt{x}$ is of type $\mathcal{X}$, $\mathtt{clk}$ is of type $\mathbb{R}_{\geq 0}$, $\mathtt{u}$ is of type $\mathcal{U}$, and $\mathtt{z}$ is of type $\mathcal{Z}$. The *dynamical system-based transition relation* $\mathcal{T}_{REnv} \subseteq \mathcal{Q}_{L,P,FB,L',P'}$ is defined as Formula (3.7):

$$\mathcal{T}_{REnv} \stackrel{\mathtt{def}}{=} \left\{ (l, p, fb, l', p') \;\middle|\; \begin{array}{c} 0 \leq l'(\mathtt{clk}) - l(\mathtt{clk}) \leq \Delta T \wedge \\ p(\mathtt{z}) = h(l(\mathtt{x})) \wedge p'(\mathtt{z}) = h(l'(\mathtt{x})) \wedge \\ l'(\mathtt{x}) = \xi(l(\mathtt{x}), fb(\mathtt{u}), l'(\mathtt{clk}) - l(\mathtt{clk})) \end{array} \right\} \tag{3.7}$$

where $l, l' \in \mathcal{Q}_L, p, p' \in \mathcal{Q}_P$ and $fb \in \mathcal{Q}_{FB}$.

In simple terms, the transition relation $\mathcal{T}_{REnv}$ relates two states $l(\mathtt{x})$ and $l'(\mathtt{x})$ as well as the measured output $p(\mathtt{z})$ and $p'(\mathtt{z})$ of the dynamical system at any two time points $l(\mathtt{clk})$ and $l'(\mathtt{clk})$ when their time difference is within ΔT. As a result, we can assume the input is a constant $fb(\mathtt{u})$, and because the dynamical system is invariant under time translation, the new state is $l'(\mathtt{x}) = \xi(l(\mathtt{x}), fb(\mathtt{u}), l'(\mathtt{clk}) - l(\mathtt{clk}))$.

Observing Definition 3.2, $\mathcal{T}_{REnv}$ might not be expressible as a predicate because (1) dynamical systems do not have a closed-form solution ξ in general, and (2) the output function h can be too complex to be specified as an expression. This poses a challenge to prove the proof obligations in Proposition 3.2 for checking if a model *Env* is an abstraction of *REnv*. We therefore need to integrate a different technique, reachability analyses, for *REnv* based on a dynamical system.

3.2.2 Guard-Reachset Assumptions of Dynamical Systems

Over three decades of research on verification of complex dynamical and hybrid systems [83, 84, 88] has led to the creation of powerful techniques to compute reachable states, namely *reachability analyses*, for linear [89, 90], nonlinear [91, 92, 93], and black-box systems [23]. Depending on the type and availability of the dynamical systems, these tools can be used for validating if a model of environment *Env* is indeed an abstraction of a real environment *REnv*. Here, we introduce the formal definition of *reachsets* as the output from querying a reachability analysis tool.[4] We then provide the definition of *guard-reachset assumption* constructed from reachsets and use the assumption to derive abstract environment models. At the end of the section, we briefly mention how the traces from the CyPhyHouse simulator [55] together with the reachability analysis tool, DryVR [23], could be used to search for the abstract environment model for various environments or find violations to the system invariant.

Definition 3.3. The *reachset*, i.e., the set of reachable states, of a continuous-time dynamical system of Equation (3.6) in the time interval $[0, \Delta T]$ and starting from an initial set $\mathcal{X}_0 \subseteq \mathcal{X}$ with a set of inputs $\mathcal{U}_0 \subseteq \mathcal{U}$ is defined as Formula (3.8):

$$Reachset(\mathcal{X}_0, \mathcal{U}_0, [0, \Delta T]) \stackrel{\texttt{def}}{=} \{\xi(x_0, u, t) \mid x_0 \in \mathcal{X}_0 \wedge u \in \mathcal{U}_0 \wedge t \in [0, \Delta T]\} \tag{3.8}$$

Computing reachsets exactly is undecidable in general [94]. Reachability analysis algorithms therefore compute bounded-time over-approximations of the reachsets instead. These algorithms commonly depend on particular set representations, such as support functions and polyhedra [90], ellipsoids [95], zonotopes [96], star-shaped sets [89], etc., for efficient approximations of reachable sets, and these set representations can be easily specified as predicates. We use reachability analysis tools with their set representations to help construct a pair of predicates, called a *guard-reachset assumption*, in Definition 3.4 below, and

[4] We do not include the algorithms for computing reachsets in this thesis. We encourage interested readers to read the textbook [84] and references therein for a better understanding of reachability analysis algorithms.

then we construct an abstraction of the real environment *REnv* using this guard-reachset assumption.

Definition 3.4. A *guard-reachset assumption* is defined by a pair of predicates $\langle G, RS \rangle$ s.t.

- The guard predicate G is over only variables in L, P, and FB.
- The reachset predicate RS is over only variables in L' and P'.
- Given a dynamical system of Equation (3.6) with the initial set $\mathcal{X}_0 \subseteq \mathcal{X}$, the set of inputs $\mathcal{U}_0 \in \mathcal{U}$, the reachset computation function *Reachset*, and the output function h, for $l, l' \in \mathcal{Q}_L, p, p' \in \mathcal{Q}_P$, and $fb \in \mathcal{Q}_{FB}$, the guard-reachset assumption satisfies the following two proof obligations:

$$(l, p, fb) \in [\![G]\!]_{L,P,FB} \implies (l(\mathtt{x}) \in \mathcal{X}_0 \wedge p(\mathtt{z}) = h(l(\mathtt{x})) \wedge fb(\mathtt{u}) \in \mathcal{U}_0) \qquad (Guard)$$

$$(l'(\mathtt{x}) \in Reachset(\mathcal{X}_0, \mathcal{U}_0, [0, \Delta T]) \wedge p'(\mathtt{z}) = h(l'(\mathtt{x}))) \implies (l', p') \in [\![RS]\!]_{L',P'} \qquad (Reachset)$$

In simple terms, the guard predicate G defines a set of valuations where each valuation maps to an initial state in $\mathcal{X}_0$ of the dynamical system, and the reachset predicate RS defines a set of valuations which over-approximates the reachset of the dynamical system under any feedback input $fb(\mathtt{u}) \in \mathcal{U}_0$.

Proposition 3.4. *Given a dynamical system of Equation* (3.6)*, let $\mathcal{T}_{REnv}$ be the dynamical system-based transition relation, and let $\langle G, RS \rangle$ be a guard-reachset assumption of the same dynamical system. The transition relation $[\![G \Rightarrow RS]\!]_{L,P,FB,L',P'}$ constructed from the predicate $(G \Rightarrow RS)$ is an abstraction of $\mathcal{T}_{REnv}$.*

Proof. We show that the proof obligation PO (*SimTrans*) in Proposition 3.2 holds. By expanding Definition 3.2, 3.3, and 3.4, we derive that every valuation l satisfying the guard $[\![G]\!]_{L,P,FB}$ is mapping $\mathtt{x}$ to an initial state $l(\mathtt{x}) \in \mathcal{X}_0$. Then, every new state $l'(\mathtt{x})$ on the time-bounded trajectory, $l'(\mathtt{x}) = \xi(l(\mathtt{x}), fb(\mathtt{u}), l'(\mathtt{clk}) - l(\mathtt{clk}))$, is inside $Reachset(\mathcal{X}_0, u, [0, \Delta T])$ and thus in the over-approximation $[\![RS]\!]_{L',P'}$. Now if the old valuation l is violating the guard, the new valuation $l'(\mathtt{x})$ can simulate any state value by definition. As a result, $[\![G \Rightarrow RS]\!]_{L,P,FB,L'P'}$ can simulate every transition of $\mathcal{T}_{REnv}$. QED.

Remark 3.5. We can query reachability analyses more than once to obtain multiple guard-reachset assumptions $\langle G_1, RS_1 \rangle, \langle G_2, RS_2 \rangle, \ldots, \langle G_n, RS_n \rangle$. The predicate $[\![\bigwedge_{i=1}^{n} (G_i \Rightarrow RS_i)]\!]$ is also an abstraction of $\mathcal{T}_{REnv}$ by Corollary 3.3.

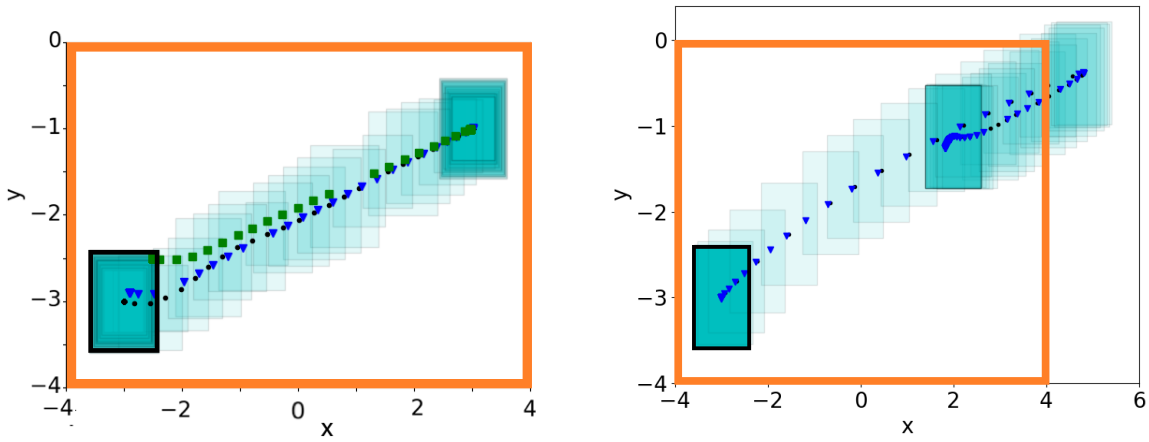


Figure 3.1: Reachsets for the F1/10$^{\text{th}}$ racecar model (*Left*) and the Hector Quadrotor model (*Right*) by DryVR. In both plots, the initial set is the **black** rectangle in the lower left corner, the large **orange** rectangle is the invariant region specifying the geofence between (-4, -4) and (4, 0). and the reachset is the union of all **blue-green** rectangles. The reachset is computed from traces starting at an initial position inside the black rectangle and traveling to the target position at (3, -1). As a side note, the blue-green color around the initial and target positions is darker because more blue-green rectangles are overlapping, and this is due to the slower speed when the vehicle is starting to move or stops moving.

Proposition 3.4 essentially states that abstractions can be generated from the reachability analysis tool. To further ensure the generated abstractions are safe, we demonstrate with a simple *generate-then-verify* approach. In this example, we first generate a guard-reachset assumption from the reachability analysis tool DryVR [23] and derive $[\![G \Rightarrow RS]\!]$ as the abstraction. Then, we check if the proof obligation PO (*IndEnv*) in Theorem 2.1 holds for the derived abstraction and prove the safety of the autonomous system.

DryVR has been used to analyze automotive and aerospace control systems [97]. DryVR uses numerical simulations to learn the sensitivity of the trajectories of the vehicle to changes in initial conditions, with a certain confidence level. Then it uses this sensitivity and additional simulations to either prove the unsafe states are unreachable (in our case, no violation of the geofence in Example 2.3) or find a counter-example. Under certain robustness assumptions, this process is also guaranteed to terminate. We used our CyPhyHouse simulator to generate traces of a dynamical system, for instance, a vehicle or a quadcopter moving from a set of initial conditions to a target position. From these traces, DryVR computes the reachsets. In Example 3.2 below, we will illustrate how to use DryVR to generate abstractions with respect to realistic vehicle motions in our CyPhyHouse simulator and visualize the abstractions with respect to the invariant.

Example 3.2. Recall from Example 2.4, it is proven that, after the program transition, the target position is always either the home or work location, and this is specified as the auxiliary predicate $Aux \stackrel{\texttt{def}}{=} (\texttt{tgt}{=}p_h \vee \texttt{tgt}{=}p_w)$. We would like to show the invariant representing the geofence $Inv \stackrel{\texttt{def}}{=} (p_{lo} \leq \texttt{pos} \leq p_{hi})$ is preserved after the environment transition. Additionally, to account for overshoot, we require the home or work location is sufficiently far away from the boundary of the geofence, that is, $p_h, p_w \in [\frac{p_{lo}+\rho \cdot p_{hi}}{1+\rho}, \frac{\rho \cdot p_{lo}+p_{hi}}{1+\rho}]$ where ρ is the maximum percentage overshoot $0 \leq \rho < 1$.

Here we use the CyPhyHouse simulator to generate traces of two types of vehicles, a car using the MIT F1/10$^{\text{th}}$ race car model [43] and a drone using the Hector Quadrotor model [42], from which DryVR computes the reachsets. Figure 3.1 shows the outputs from DryVR performed on the car (*Left*) and on the drone (*Right*). In both plots, the large orange rectangle represents the invariant region specifying the geofence between (-4, -4) and (4, 0). We let the maximum percentage overshoot be $\rho = \frac{1}{3}$, and select an initial position in the black rectangle around (-3, -3) and set the target position at (3, -1). We collect simulation traces of each vehicle then compute the reachsets with DryVR. The reachsets are shown as the union of all blue-green rectangles.

We observe that for the same set of the initial positions and the same target, the drone has a larger reachset than the car. The drone may temporarily go out of the geofence due to severe overshoot, so we did not find a safe abstraction of the drone dynamics. On the other hand, the reachability analysis shows the car stays within the invariant, and hence the constructed guard-reachset assumption is a safe abstraction.

We note that the guards in the complete guard-reachset assumption should cover all reasonable initial positions within the geofence *Inv* as well as all possible target positions constrained within *Aux*, otherwise the proof obligation PO (*IndEnv*) will never hold because a configuration outside the guards can transit to any configuration. To reduce the amount of simulation traces covering all pairs of initial and target positions, we can exploit the fact that the motion of quadrotor is symmetric under translations, planar reflections, and rotations. Therefore, using Theorem 10 from [98] and as shown in [99], the computed reachsets can be translated and rotated to cover all choices of initial and target positions.

3.3 SUMMARY

In this chapter, we formally defined abstractions of the environment based on the simulation relation between the model *Env* and the real environment *REnv*, and we formalize the more important *safe abstractions* which preserve the invariant in environment turn transitions and subsequently prove the system-level safety. In addition, we outlined the synthesis

problem of safe abstractions. We showed that existing reachability analysis tools can search for safe abstractions of the environment, and we demonstrated how to use DryVR to find safe abstractions for our autonomous vehicle example in the realistic Gazebo simulator. In the rest of the thesis, we instantiate the CPReact framework to analyze two categories of autonomous systems, distributed robotics systems and systems using vision-based perception.

Chapter 4: Abstraction and Verification with Koord Semantics

In this chapter, we demonstrate that the compositional reasoning on the reactive module and the environment can be integrated into existing software analyses for distributed robotics systems. We extend the inductive invariant-based proof in Chapter 2 and instantiate the proof obligations in Theorem 2.1 for multi-agent systems with synchronous communication. We show that the instantiated proof obligations can be discharged with existing program verification tools that check inductive invariants for distributed systems.

Specifically, we apply our CPReact framework for reformulating the Koord language and program verification methods presented in our conference paper [32].[5] The Koord language is the modeling and programming language for implementing distributed robotic applications (DRAs) in the CyPhyHouse project [55]. The Koord compiler allows generating code that can be and has been directly deployed on aerial and ground vehicle platforms in Intelligent Robotics Lab[6] and simulated with the CyPhyHouse simulator. The executable formal semantics for Koord in the $\mathbb{K}$ framework [100] further enables the formal verification and validation on any DRA implemented in Koord. The formal analysis methods presented in [32, 33] verify invariant properties via computing the reachable configurations through program and environment transitions.

We claim that any program implemented with the Koord language is an instance of a CPReact model; we therefore are able to apply the proof techniques discussed in Chapter 2 for unbounded verification. In addition, we illustrate how to reuse the existing formal analysis methods for Koord to discharge the proof obligations for CPReact. In Section 4.1, we first connect the Koord executable semantics in [32, 33] with the CPReact system model. In Section 4.2, we set up the induction proof for Koord applications for unbounded verification based on Theorem 2.1. In other words, we derive the three proof obligations from Theorem 2.1 for a Koord program including (1) PO (*Init*) for initial configurations, (2) PO (*IndProg*) for *platform-independent* program transitions, and (3) PO (*IndEnv*) for *platform-dependent* environment transitions. In Section 4.3, we reuse the symbolic execution for Koord and discharge the proof obligations for program transitions using Satisfiability Modulo Theory (SMT) solvers according to Corollary 2.2. Lastly, we show in Section 4.4 that the notion of *controller port assumptions* proposed in [32] is an instance of guard-reachset assumptions in Definition 3.4, and we apply reachability analyses to check the proof obligations for environment transitions according to Proposition 3.4.

[5] This a joint work with Dr. Ritwika Ghosh and Prof. Sasa Misailovic published in the International Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA) in 2020.

[6] Website: `https://robotics.illinois.edu/lab/`

4.1 CONNECTING KOORD SEMANTICS WITH CPREACT EXECUTIONS

In this section, we show that any KOORD program is an instance of a CPREACT model by deriving the conversion from *system configurations* and *execution rounds* of a KOORD program to the configurations and transition relations of a CPREACT model.

4.1.1 KOORD System Configurations to CPREACT Configurations

Our first task is to match the configuration of a KOORD application from Chapter 3 in [33] to the configuration of a CPREACT model defined in Section 2.2. We achieve this by listing all variables in the configuration of a KOORD application, and we then match them to the latent and percept variables of *Env* as well as the state and feedback variables of *RM* in a CPREACT model. At a high level, a KOORD application is running on a distributed system of multiple robots, so a configuration of a KOORD application, namely a *system configuration*, consists of a collection of *robot configuration* as well as shared variables for communications between robots, and each robot configuration is used to specify the semantics of each participating robot. Each participating robot would have its own set of module ports and local variables, along with a local copy of shared variables.

Formally, given a system of N robots with identifiers from $\texttt{ID} = \{0, \ldots, N-1\}$, the *system configuration* in KOORD semantics is a tuple $\boldsymbol{c} = (\{\boldsymbol{L}_i\}_{i \in \texttt{ID}}, gm, \tau, turn)$ where

(1) $\{\boldsymbol{L}_i\}_{i \in \texttt{ID}}$ or $\{\boldsymbol{L}_i\}$ in short is an indexed set of *robot configurations*–one for each participating robot. $\boldsymbol{L}_i$ refers to the configuration of the i-th robot in the system.

(2) $gm \in \mathcal{Q}_{GVar}$ is the *global memory*, mapping shared variables $GVar$ to values.

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

(4) $turn \in \{\texttt{prog}, \texttt{env}\}$ determines whether program or environment transitions are being processed.

For each robot, let $MVar$ be the set of local variables and copies of shared variables, let $Sens$ be the set of sensor ports, and let $Acts$ be the set of actuator ports.[7] A *robot configuration* is a tuple $\boldsymbol{L} = (lm, sp, ap, turn)$ where

(1) $lm \in \mathcal{Q}_{MVar}$ is its *local memory* for both local variables and copies of shared variables.

(2) $sp \in \mathcal{Q}_{Sens}$ is the mapping of sensor ports to values.

[7] In [32, 33], sensor and actuator ports are aggregated as controller ports.

(3) $ap \in \mathcal{Q}_{Acts}$ is the mapping of actuator ports to values.

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

For readability, we use the dot (".") notation to access components of system and robot configurations. For example, $\boldsymbol{L}_i.lm$ means accessing the local context lm in the robot configuration $\boldsymbol{L}_i$ of the i-th robot. We further use the notation $\{\boldsymbol{L}_i.lm\}_{i\in\texttt{ID}}$ or in short $\{\boldsymbol{L}_i.lm\}$ to denote the indexed set of the local memories lm of all robots $\{\boldsymbol{L}_0.lm, \boldsymbol{L}_1.lm, \ldots, \boldsymbol{L}_{N-1}.lm\}$.

Table 4.1: Conversion of sets of variables from KOORD to CPREACT.

CPREACT			KOORD
Turn	$turn$	$=$	$turn$
Percept Variables	P	$=$	$\{\tau\} \cup \bigcup_{i\in\texttt{ID}} \boldsymbol{L}_i.Sens$
Feedback Variables	FB	$=$	$\bigcup_{i\in\texttt{ID}} \boldsymbol{L}_i.Acts$
State Variables of RM	S	$=$	$GVar \cup \bigcup_{i\in\texttt{ID}} (\boldsymbol{L}_i.MVar \cup \{\boldsymbol{L}_i.turn\})$
Latent Variables of Env	L	$=$	$\emptyset$

Table 4.1 summarizes the conversion of different sets of variables from a KOORD application to a CPREACT model. It is straightforward to match the $turn$ variable because it is in both KOORD and CPREACT. We consider that the global time τ is observed from the environment and hence a percept variable. The sensor ports of all participating robots are percept variables, and the actuator ports of all robots are feedback variables. The state variables of RM include all variables in the global memory and all local memories. In addition, the turn variable for each robot is considered as a state variable. Finally, it is assumed in KOORD semantics that all environment states are observable through sensor ports; thus the latent variables of Env is an empty set.

```
1  using M: // Motion module
2    sensors: Point pos
3    actuators: Point tgt
4
5  allread: Point x[N]
6  init:
7    x[pid] = M.pos
8  Avg:
9    pre: ¬ (pid = N − 1 ∨ pid = 0)
10   eff:
11       M.tgt := (x[pid+1] + x[pid−1]) / 2
12       x[pid] := M.pos
13 Skip:
14   pre: pid = N − 1 ∨ pid = 0
```

Figure 4.1: KOORD program LineForm for a set of robots to form a line.

Example 4.1. We use the distributed line formation case study from Chapter 6 in [33] as an example. Figure 4.1 shows the KOORD program, LineForm, for distributed line formation.

In LineForm, every robot calculates the next target position by averaging the positions of its predecessor and successor except for robot 0 and robot $N-1$, and it shares its current position through the shared variable `x`. and this averaging and sharing rule is specified in the `Avg` event. On the other hand, robot 0 and robot $N-1$ stay at the same position in the `Skip` event.

We now dive into the detail of specifying the system and robot configurations. The set of global variables has one variable $GVar = \{\texttt{x}\}$. For each robot i, the domain of local memory has one variable $\boldsymbol{L}_i.MVar = \{\boldsymbol{L}_i.\texttt{x}\}$ because it contains a copy of the global variable. The sensor ports contain one variable $\boldsymbol{L}_i.Sens = \{\boldsymbol{L}_i.\texttt{M.pos}\}$, and the actuator ports contains one variable $\boldsymbol{L}_i.Acts = \{\boldsymbol{L}_i.\texttt{M.tgt}\}$. According to the conversion in Table 4.1, we derive Equation 4.1:

$$\begin{aligned} P &= \{\tau\} \cup \{\boldsymbol{L}_i.\texttt{M.pos}\}_{i\in\texttt{ID}} \\ FB &= \{\boldsymbol{L}_i.\texttt{M.tgt}\}_{i\in\texttt{ID}} \\ S &= \{\texttt{x}\} \cup \{\boldsymbol{L}_i.\texttt{x}\}_{i\in\texttt{ID}} \cup \{\boldsymbol{L}_i.turn\}_{i\in\texttt{ID}} \\ L &= \emptyset \end{aligned} \tag{4.1}$$

Below we give a concrete example configuration of the CPReact model. Let the number of robots $N = 3$; then the set of identifiers is $\texttt{ID} = \{0, 1, 2\}$. `pid` is a reserved word in Koord for referring to each robot's own identifier. We assume the Point type represents 2D positions. An initial configuration $(\texttt{prog}, p_0, fb_0, l_0, s_0)$ satisfying the initial predicate $\texttt{x}[\texttt{pid}] = \texttt{M.pos}$ is given as Equation 4.2:

$$\begin{aligned} p_0 &= \{\boldsymbol{L}_0.\texttt{M.pos}\rightarrow (0,0), \boldsymbol{L}_1.\texttt{M.pos}\rightarrow (0,1), \boldsymbol{L}_2.\texttt{M.pos}\rightarrow (2,2), \tau\rightarrow 0\} \\ fb_0 &= \{\boldsymbol{L}_0.\texttt{M.tgt}\rightarrow (0,0), \boldsymbol{L}_1.\texttt{M.tgt}\rightarrow (1,1), \boldsymbol{L}_2.\texttt{M.tgt}\rightarrow (2,2)\} \\ l_0 &= \emptyset \\ s_0 &= \{\texttt{x}\rightarrow [(0,0), (0,1), (2,2)], \\ &\quad \boldsymbol{L}_0.turn\rightarrow \texttt{prog}, \boldsymbol{L}_0.\texttt{x}\rightarrow [(0,0), (0,1), (2,2)], \\ &\quad \boldsymbol{L}_1.turn\rightarrow \texttt{prog}, \boldsymbol{L}_1.\texttt{x}\rightarrow [(0,0), (0,1), (2,2)], \\ &\quad \boldsymbol{L}_2.turn\rightarrow \texttt{prog}, \boldsymbol{L}_2.\texttt{x}\rightarrow [(0,0), (0,1), (2,2)]\} \end{aligned} \tag{4.2}$$

4.1.2 Koord Transition Rules to CPReact Transition Relations

Given a Koord application of N robots, we define the following transition relations according to the semantic rules in [33], and build the transition relations $[\![T_{RM}]\!]$ and $[\![T_{Env}]\!]$. According to the Koord semantics rules and the conversion of variables in Table 4.1, we can model the program turn with the transition relation $[\![T_{RM}]\!]$ over the state, percept, and

feedback variables, namely S, P and FB. Similarly, we can capture the environment turn with the transition relation $[\![T_{Env}]\!]$ over the latent, percept, and feedback variables, namely L, P and FB. We first built the transition relation $[\![T_{RM}]\!]$ for the program turn as follows:

(1) $[\![T_{i,ev}]\!]$ represents the transition relation of the robot i executing an event $ev \in Events$ where the event ev consists of the precondition $Cond$ and effect statements $Body$. Formally, $[\![T_{i,ev}]\!]$ is defined as Formula 4.3:

$$\begin{aligned}[\![T_{i,ev}]\!] \stackrel{\texttt{def}}{=} \{((gm, \boldsymbol{L}_i.lm), \boldsymbol{L}_i.sp, (gm', \boldsymbol{L}'_i.lm), \boldsymbol{L}'_i.ap) \mid \\ [\![Cond]\!]_{GVar,\boldsymbol{L}_i} \wedge \langle gm, \boldsymbol{L}_i, Body\rangle \rightarrow_{stmt} \langle gm', \boldsymbol{L}'_i, \cdot\rangle\}\end{aligned} \tag{4.3}$$

Note that the robot i reads the global and local memory as well as sensor ports, namely gm, $\boldsymbol{L}_i.lm$ and $\boldsymbol{L}_i.sp$, it then transits to the new global and local memory and provides new actuator port values $\boldsymbol{L}'_i.ap$.

(2) $[\![T_{i,RM}]\!]$ represents the transition relation of robot i executing any event in $Events$. Formally, $[\![T_{i,RM}]\!]$ is defined as Formula 4.4:

$$[\![T_{i,RM}]\!] \stackrel{\texttt{def}}{=} \bigcup_{ev \in Events} [\![T_{i,ev}]\!] \tag{4.4}$$

(3) $[\![T_{\vec{p}}]\!]$ represents the transition relation of a system program turn when robots execute their events in the order $\vec{p}$, where $\vec{p}$ is a permutation of $\texttt{ID}$ given as $(i_1, i_2, \ldots, i_N)$. Formally, $[\![T_{\vec{p}}]\!]$ is given as Formula 4.5:

$$\begin{aligned}[\![T_{\vec{p}}]\!] \stackrel{\texttt{def}}{=} \{((gm_0, \{\boldsymbol{L}_i.lm\}), \{\boldsymbol{L}_i.sp\}, (gm_N, \{\boldsymbol{L}'_i.lm\}), \{\boldsymbol{L}'_i.ap\}) \mid \\ ((gm_0, \boldsymbol{L}_{i_1}.lm), \boldsymbol{L}_{i_1}.sp, (gm_1, \boldsymbol{L}'_{i_1}.lm), \boldsymbol{L}'_{i_1}.ap) \in [\![T_{i_1,RM}]\!] \wedge \\ ((gm_1, \boldsymbol{L}_{i_2}.lm), \boldsymbol{L}_{i_2}.sp, (gm_2, \boldsymbol{L}'_{i_2}.lm), \boldsymbol{L}'_{i_2}.ap) \in [\![T_{i_2,RM}]\!] \wedge \\ \vdots \\ ((gm_{N-1}, \boldsymbol{L}_{i_N}.lm), \boldsymbol{L}_{i_N}.sp, (gm_N, \boldsymbol{L}'_{i_N}.lm), \boldsymbol{L}'_{i_N}.ap) \in [\![T_{i_N,RM}]\!]\}\end{aligned} \tag{4.5}$$

(4) $[\![T_{RM}]\!]$ is the union of $[\![T_{\vec{p}}]\!]$ over all permutations of orders $\vec{p}$. Formally, $[\![T_{\vec{p}}]\!]$ is given as Formula 4.6:

$$[\![T_{RM}]\!] \stackrel{\texttt{def}}{=} \bigcup_{\vec{p} \in perms(\texttt{ID})} [\![T_{\vec{p}}]\!] \tag{4.6}$$

where $perms(\texttt{ID})$ refers to the set of permutations of $\texttt{ID}$.

Notice that, consistent with Table 4.1, the transition relation $[\![T_{RM}]\!]$ transits from the old

state variables S to new state variables S' which consists of the global variable $GVar$ and local variables of all agents $\boldsymbol{L}_i.MVar$.

Next, we define the transition relation $[\![T_{Env}]\!]$ of environment transitions for KOORD. Environment transitions in [33] capture the evolution of the actuator ports over a time interval $[0, \Delta T]$—all other parts of the robot configuration remain unchanged. We first define the environment transition relation $[\![T_{i,Env}]\!]$ for each robot i, and then construct the transition relation $[\![T_{Env}]\!]$ for the entire distributed robotic system. The KOORD semantics defines the environment transitions of each robot i by specifying a (possibly black-box) function for a dynamical system.[8] The type of this function ξ_i is defined by sensor and actuator ports, namely, $\xi_i : \mathcal{Q}_{Sens} \times \mathcal{Q}_{Acts} \times \mathbb{R}_{\geq 0} \mapsto \mathcal{Q}_{Sens'}$. Given old sensor values $\boldsymbol{L}_i.sp$, actuator values $\boldsymbol{L}_i.ap$ and a duration ΔT, ξ_i should return the new values for all sensor ports $\boldsymbol{L}'_i.sp$ for simulating the dynamics of each robot i for the duration ΔT. Given such a function ξ_i for each robot i and two time points $\tau, \tau' \in \mathbb{R}_{\geq 0}$ with $\tau \leq \tau' \leq \tau + \Delta T$, we define the transition relation $[\![T_{i,Env}]\!]$ to represent the evolution of each robot i over a $[\tau, \tau']$ time interval. $[\![T_{i,Env}]\!]$ is constructed by simply updating the sensor ports sp of robot i with ξ_i. Formally, $[\![T_{i,Env}]\!]$ is defined as Formula 4.7:

$$[\![T_{i,Env}]\!] \stackrel{\texttt{def}}{=} \{((\tau, \boldsymbol{L}_i.sp), \boldsymbol{L}_i.ap, (\tau', \boldsymbol{L}'_i.sp)) \mid \boldsymbol{L}'_i.sp = \xi_i(\boldsymbol{L}_i.sp, \boldsymbol{L}_i.ap, \tau' - \tau)\} \tag{4.7}$$

We construct the transition relation $[\![T_{\texttt{ID},Env}]\!]$ of the system such that all robots evolve according to the global time. Formally, $[\![T_{\texttt{ID},Env}]\!]$ is defined as Formula 4.8:

$$[\![T_{\texttt{ID},Env}]\!] \stackrel{\texttt{def}}{=} \left\{ ((\tau, \{\boldsymbol{L}_i.sp\}), \{\boldsymbol{L}_i.ap\}, (\tau', \{\boldsymbol{L}'_i.sp\})) \,\middle|\, \bigwedge_{i \in \texttt{ID}} ((\tau, \boldsymbol{L}_i.sp), \boldsymbol{L}_i.ap, (\tau', \boldsymbol{L}'_i.sp)) \in [\![T_{i,Env}]\!] \right\} \tag{4.8}$$

Notice that $[\![T_{\texttt{ID},Env}]\!]$ represents relations between transient configurations of any two time points τ and τ'. Now to conform to the KOORD semantics in [33], we carefully define the exact transition relation $[\![T_{Env}]\!]$ between the end of each *round* without transient configurations by restricting $\tau' = \tau + \Delta T$. Formally, $[\![T_{Env}]\!]$ is given as Formula 4.9:

$$[\![T_{Env}]\!] \stackrel{\texttt{def}}{=} \{((\tau, \{\boldsymbol{L}_i.sp\}), \{\boldsymbol{L}_i.ap\}, (\tau + \Delta T, \{\boldsymbol{L}'_i.sp\})) \in [\![T_{\texttt{ID},Env}]\!]\} \tag{4.9}$$

Recall in Table 4.1, the set of latent variables L is an empty set. the set of percept variables P includes the global clock τ and the sensor ports $\boldsymbol{L}_i.Sens$, and the set of feedback variables FB includes the actuator ports $\boldsymbol{L}_i.Acts$. Therefore, we have established that $[\![T_{Env}]\!]$ takes

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

feedback variables FB as input and updates the percept variables P.

Example 4.2. Following the LineForm example in Figure 4.1, we construct the transition relations as predicates for program turns in this example. We skip the construction for the environment turn because it is almost the same procedure. For program turn transitions, we first construct $T_{i,\texttt{Avg}}$ represents the transition relation of the robot i executing the `Avg` event. We can derive the transition relation of the event `Avg` in Figure 4.1 as the following Boolean expression in Formula 4.10:

$$\begin{aligned} T_{i,\texttt{Avg}} \stackrel{\texttt{def}}{=} & \neg(i = N-1 \vee i = 0) && \cdots \text{Precondition at Line 9} \\ & \wedge \boldsymbol{L}_i'.\texttt{M.tgt} = (\texttt{x}[i-1] + \texttt{x}[i+1])/2 && \cdots \text{Assignment at Line 11} \\ & \wedge \texttt{x}'[i] = \boldsymbol{L}_i.\texttt{M.pos} && \cdots \text{Assignment at Line 12} \\ & \wedge \mathit{unchanged\_vars} \end{aligned} \tag{4.10}$$

where *unchanged_vars* are additional constraints denoting that all other fields of $\boldsymbol{L}_i$ and $\boldsymbol{L}_i'$ stay the same, and similarly other array elements of `x` and `x'` stay the same. For example, $\boldsymbol{L}_i.\texttt{Motion.pos} = \boldsymbol{L}_i'.\texttt{Motion.pos}$ and $\bigwedge_{j \neq i} \texttt{x}[j] = \texttt{x}'[j]$. Similarly, We can derive the transition relation of the event `Skip` as Formula 4.11:

$$\begin{aligned} T_{i,\texttt{Skip}} \stackrel{\texttt{def}}{=} & (i = N-1 \vee i = 0) && \cdots \text{Precondition at Line 13} \\ & \wedge \boldsymbol{L}_i'.\texttt{M.tgt} = \boldsymbol{L}_i.\texttt{M.tgt} && \cdots \text{Actuators unchanged} \\ & \wedge \texttt{x}' = \texttt{x} && \cdots \text{Shared variables unchanged} \end{aligned} \tag{4.11}$$

Then, we get Formula 4.12:

$$T_{i,RM} = T_{i,\texttt{Avg}} \vee T_{i,\texttt{Skip}} \tag{4.12}$$

Now we have to consider all possible permutations of robots executing events. In the case when there are $N = 3$ robots, we first consider the transition relation for a particular execution order (i_1, i_2, i_3). For $k \in \{0, 1, \ldots, N\}$, we denote $GVar_k$ a copy of the set of global variables $GVar$ but each variable name is modified with the suffix k. The transition relation for the execution order (i_1, i_2, i_3) is then expressed as the predicate in Formula 4.13:

$$\begin{aligned} T_{(i_1,i_2,i_3)} \stackrel{\texttt{def}}{=} & \; T_{i_1,RM}[GVar_0/GVar, GVar_1/GVar'] \\ & \wedge T_{i_2,RM}[GVar_1/GVar, GVar_2/GVar'] \\ & \wedge T_{i_3,RM}[GVar_2/GVar, GVar_3/GVar'] \end{aligned} \tag{4.13}$$

The transition relation for all possible permutations for $N = 3$ robots is given as For-

mula 4.14:

$$T_{RM} \stackrel{\texttt{def}}{=} T_{(0,1,2)} \vee T_{(0,2,1)} \vee T_{(1,0,2)} \vee T_{(1,2,0)} \vee T_{(2,0,1)} \vee T_{(2,1,0)} \tag{4.14}$$

To summarize, we have shown that a KOORD application is an instance of the reactive module interacting with the environment. We show the conversion by grouping the variables from KOORD system configurations to latent, percept, state, and feedback variables as well as constructing the transition relations $[\![T_{RM}]\!]$ and $[\![T_{Env}]\!]$. From Example 4.2, we also observe that the growth of the size of the predicate T_{RM} is factorial with respect to the number of robots N. In general, for a KOORD program with n events running on N robots, the size of T_{RM} is proportional to $N! \times N \times n$.

4.2 DECOMPOSING INVARIANCE VERIFICATION

In this section, we show how to apply the inductive proof in Theorem 2.1 for showing the safety of KOORD applications. An *invariant* of a KOORD program is a predicate that holds in all reachable configurations. Invariant requirements can express safety, for instance, that no two robots are ever too close (Collision avoidance), or that robots always stay within a designated area (Geofencing). Following the discussion in Section 2.3, we now apply Theorem 2.1 to decompose the verification into checking the invariant for the program turn and the environment turn separately. We first assume that the safety is exactly the same as the invariant because there is no latent variables. Theorem 2.1 is instantiated for KOORD as below.

Proposition 4.1. *Given a predicate for the initial configurations of the system $Init_{Sys}$, a predicate Inv is an* inductive invariant *of the system if, for any system configuration $(\{\boldsymbol{L}_i\}_{i\in ID}, gm, \tau, turn)$, the following three proof obligations (POs) are met:*

$$(\{\boldsymbol{L}_i\}, gm, \tau, turn) \in [\![Init_{Sys}]\!] \implies (\{\boldsymbol{L}_i\}, gm, \tau, turn) \in [\![Inv]\!] \tag{Init}$$

$$(\{\boldsymbol{L}_i\}, gm, \tau, \texttt{prog}) \in [\![Inv]\!] \wedge ((gm, \{\boldsymbol{L}_i.lm\}), \{\boldsymbol{L}_i.sp\}, (gm', \{\boldsymbol{L}'_i.lm\}), \{\boldsymbol{L}'_i.ap\}) \in [\![T_{RM}]\!] \implies (\{\boldsymbol{L}'_i\}, gm', \tau, \texttt{env}) \in [\![Inv]\!] \tag{IndProg}$$

$$(\{\boldsymbol{L}_i\}, gm, \tau, \texttt{env}) \in [\![Inv]\!] \wedge ((\tau, \{\boldsymbol{L}_i.sp\}), \{\boldsymbol{L}_i.ap\}, (\tau + \Delta T, \{\boldsymbol{L}'_i.sp\})) \in [\![T_{Env}]\!] \implies (\{\boldsymbol{L}'_i\}, gm, \tau + \Delta T, \texttt{prog}) \in [\![Inv]\!] \tag{IndEnv}$$

That is, *Inv* holds in the initial configuration(s) by PO (*Init*), and *Inv* is preserved in both platform-independent discrete program transitions by PO (*IndProg*) and the platform-dependent environment transitions by PO (*IndEnv*). Proving PO (*Init*) is usually trivial. Therefore, we focus on PO (*IndProg*) and (*IndEnv*).

A major bottleneck in proving PO (*IndProg*) is the required enumeration of all permutations $\vec{p} \in perms(\texttt{ID})$ for all robots with reads/writes to global memory. Recall from Example 4.2 that the size of the predicate T_{RM} is proportional to $N! * N * n$ with N robots and n events. We therefore seek for an sufficient condition to avoid enumerating all permutations described below.

Lemma 4.2. *If the invariant Inv is preserved by the transition relation of every event $[\![T_{i,ev}]\!]$, then Inv is preserved by the transition relation of system program turn $[\![T_{RM}]\!]$. Formally, if for any robot $i \in \texttt{ID}$ and any event $ev \in Events$, the following proof obligation is met:*

$$\left(\begin{array}{c} (\{\boldsymbol{L}_i\}, gm, \tau, \texttt{prog}) \in [\![Inv]\!] \wedge \\ ((gm, \boldsymbol{L}_i.lm), \boldsymbol{L}_i.sp, (gm', \boldsymbol{L}'_i.lm), \boldsymbol{L}'_i.ap) \in [\![T_{i,ev}]\!] \end{array} \right) \implies (\{\boldsymbol{L}'_i\}, gm', \tau, \texttt{env}) \in [\![Inv]\!] \tag{IndEvent}$$

then PO (IndProg) holds.

Proof. The proof follows from expanding the definition of $[\![T_{RM}]\!]$ and inducting on each event sequence. As *Inv* is preserved in the transition relation $[\![T_{i,ev}]\!]$ of every event by every robot, the order of robot events do not violate *Inv*. QED.

With Lemma 4.2, we can prove PO (*IndEvent*) instead of PO (*IndProg*), and it is no longer required to enumerate all permutations of the order of robots. This reduces the size of predicates to $N * n$ in the proof obligations and helps address the scalability issue greatly.

Now we discuss our approach to discharge PO (*IndEnv*). We expand PO (*IndEnv*) as for any robot $i \in \texttt{ID}$ and for any time point $t \in [0, \Delta T]$

$$\left(\begin{array}{c} (\{\boldsymbol{L}_i\}, gm, \tau, \texttt{env}) \in [\![Inv]\!] \wedge \\ \bigwedge\limits_{i \in \texttt{ID}} \boldsymbol{L}'_i.sp = \xi_i(\boldsymbol{L}_i.sp, \boldsymbol{L}_i.ap, t) \end{array} \right) \implies (\{\boldsymbol{L}'_i\}, gm, \tau + \Delta T, \texttt{prog}) \in [\![Inv]\!] \tag{IndTraj}$$

PO (*IndTraj*) requires reasoning about the dynamic behavior of ξ_i during environment transitions, and it is a challenging research problem by itself. We revisit *controller assumptions* defined in [33, Definition 5.4] to abstract away the continuous dynamic behavior.

Definition 4.1. Given the ξ_i function and a pair of predicates G_i and RS_i, where $[\![G_i]\!] \subseteq \mathcal{Q}_{Sens} \times \mathcal{Q}_{Acts}$ and $[\![RS_i]\!] \subseteq \mathcal{Q}_{Sens'}$, the pair $\langle G_i, RS_i \rangle$ is a *controller port assumption* if for any robot configurations $\boldsymbol{L}_i$ and $\boldsymbol{L}'_i$, and for any time point $t \in [0, \Delta T]$,

$$(\boldsymbol{L}_i.sp, \boldsymbol{L}_i.ap) \in [\![G_i]\!] \wedge \boldsymbol{L}'_i.sp = \xi_i(\boldsymbol{L}_i.sp, \boldsymbol{L}_i.ap, t) \implies \boldsymbol{L}'_i.sp \in [\![RS_i]\!] \tag{PortAsm}$$

A controller assumption $\langle G_i, RS_i \rangle$ is similar to preconditions and postconditions for ξ_i with an additional guarantee that RS_i must hold during the whole time horizon $[0, \Delta T]$.

Lemma 4.3. *Given controllers assumptions $\langle G_i, RS_i\rangle$ of the function ξ_i for all robots $i \in \texttt{ID}$, if $\langle G_i, RS_i\rangle$ preserves the invariant Inv, that is,*

$$\left(\begin{array}{c}(\{\boldsymbol{L}_i\}, gm, \tau, \texttt{env}) \in [\![Inv]\!] \wedge \\ \bigwedge\limits_{i\in \texttt{ID}} (\boldsymbol{L}_i.sp, \boldsymbol{L}_i.ap) \in [\![G_i]\!] \Rightarrow \boldsymbol{L}'_i.sp \in [\![RS_i]\!]\end{array}\right) \implies (\{\boldsymbol{L}'_i\}, gm, \tau + \Delta T, \texttt{prog}) \in [\![Inv]\!] \tag{IndPort}$$

then PO (IndTraj) holds.

Proof. Lemma 4.3 holds because a controller assumption in Definition 4.1 is an instance of the guard-reachset assumption in Definition 3.4. Hence, following Proposition 3.4, controller assumptions over-approximate all transient behaviors of the (possibly black-box) function ξ_i at any time point $t \in [0, \Delta T]$. QED.

In addition, PO (*PortAsm*) can be validated by applying reachability analyses using the input and output of ξ_i as a black-box function as discussed in Section 3.2. We will demonstrate in Section 4.4 to validate PO (*PortAsm*) with our KOORD simulator and other specialized tools for continuous dynamics.

4.3 PROVING INDUCTIVE INVARIANT WITH SMT SOLVERS

The KOORD language benefits from the $\mathbb{K}$ semantics framework and comes with a symbolic execution engine (See Chapter 5 in [33]). We therefore can reuse this symbolic execution engine to construct the transition relation for program transitions as constraints. The symbolic execution engine generates such constraints for every robot $i \in \texttt{ID}$ by the $\mathbb{K}$-Z3 interface, and then checks their *validity* using the SMT solver Z3 [86], that is, it returns 'valid' if *the negation of a generated constraint* is unsatisfiable. Otherwise, it returns 'invalid' along with a satisfiable model as the counterexample. We reuse this capability to discharge the two proof obligations, PO (*IndEvent*) and PO (*IndPort*), shown in Lemma 4.2. Since the technical details are nicely summarized by Ghosh in [33], we encourage the readers to read [33] for the complete implementation.

We would like to verify that LineForm preserves a geofence. The geofence invariant is given as the following predicate in Formula 4.15:

$$Inv \stackrel{\texttt{def}}{=} \bigwedge_{i\in \texttt{ID}} \left(\begin{array}{l}\texttt{p}_{\min} \leq \boldsymbol{L}_i.\texttt{M.pos} \leq \texttt{p}_{\max} \wedge \\ \texttt{p}_{\min} \leq \boldsymbol{L}_i.\texttt{M.tgt} \leq \texttt{p}_{\max} \wedge \\ \texttt{p}_{\min} \leq \texttt{x}[i] \leq \texttt{p}_{\max}\end{array}\right) \tag{4.15}$$

This invariant asserts that the position and target of each robot i are always within the rectangle defined by $\mathtt{p}_{\min}$ and $\mathtt{p}_{\max}$, and that each agent updates its shared variable value to one within $\mathtt{p}_{\min}$ and $\mathtt{p}_{\max}$ as well. For simplicity, we assume the operator '$\leq$' is extended to 3D points where $\mathtt{p}_1 \leq \mathtt{p}_2$ means $\mathtt{p}_1[k] \leq \mathtt{p}_2[k]$ for every dimension k.

Discharge Proof Obligations for Events We first demonstrate how to discharge the proof obligation PO (*IndEvent*) for events. Recall from Example 4.2, we have the transition relation of the event `Avg` as the predicate $T_{i,\mathtt{Avg}}$ in Formula 4.10. To check PO (*IndEvent*), we can construct the following expression $\varphi_{\mathtt{Avg}}$ in Formula 4.16 for the satisfiability query:

$$\begin{aligned}
\varphi_{\mathtt{Avg}} &\stackrel{\mathtt{def}}{=} \mathit{Inv} \wedge T_{i,\mathtt{Avg}} \wedge \neg \mathit{Inv}[\mathit{GVar}'/\mathit{GVar}, \boldsymbol{L}_i'.\mathit{MVar}/\boldsymbol{L}_i.\mathit{MVar}, \boldsymbol{L}_i'.\mathit{Acts}/\boldsymbol{L}_i.\mathit{Acts}] \\
&\stackrel{\mathtt{def}}{=} \bigwedge_{i \in \mathtt{ID}} \begin{pmatrix} \mathtt{p}_{\min} \leq \boldsymbol{L}_i.\mathtt{M.pos} \leq \mathtt{p}_{\max} \wedge \\ \mathtt{p}_{\min} \leq \boldsymbol{L}_i.\mathtt{M.tgt} \leq \mathtt{p}_{\max} \wedge \\ \mathtt{p}_{\min} \leq \mathtt{x}[i] \leq \mathtt{p}_{\max} \end{pmatrix} \wedge \begin{pmatrix} \neg(i = N-1 \vee i = 0) \\ \wedge\ \boldsymbol{L}_i'.\mathtt{M.tgt} = (\mathtt{x}[i-1] + \mathtt{x}[i+1])/2 \\ \wedge\ \mathtt{x}'[i] = \boldsymbol{L}_i.\mathtt{M.pos} \end{pmatrix} \\
&\wedge \neg \bigwedge_{i \in \mathtt{ID}} \begin{pmatrix} \mathtt{p}_{\min} \leq \boldsymbol{L}_i.\mathtt{M.pos} \leq \mathtt{p}_{\max} \wedge \\ \mathtt{p}_{\min} \leq \boldsymbol{L}_i'.\mathtt{M.tgt} \leq \mathtt{p}_{\max} \wedge \\ \mathtt{p}_{\min} \leq \mathtt{x}'[i] \leq \mathtt{p}_{\max} \end{pmatrix}
\end{aligned} \tag{4.16}$$

Notice that only state variables $S = \mathit{GVar} \cup \boldsymbol{L}_i.\mathit{MVar}$ and feedback variables $\mathit{FB} = \boldsymbol{L}_i.\mathit{Acts}$ are replaced. Z3 is able to answer that $\varphi_{\mathtt{Avg}}$ is unsatisfiable, and thus the proof obligation is proven valid. If we manually inspect the expressions, we can easily see the unsatisfiability by checking the new values of $\boldsymbol{L}_i'.\mathtt{M.tgt}$ and $\mathtt{x}'[i]$. First, $\boldsymbol{L}_i'.\mathtt{M.tgt}$ will stay within the geofence, i.e., within $\mathtt{p}_{\min}$ and $\mathtt{p}_{\max}$, because the old values of $\mathtt{x}[i-1]$ and $\mathtt{x}[i+1]$ is within the geofence. Second, $\mathtt{x}'[i]$ will stay within the geofence because the old value of $\boldsymbol{L}_i.\mathtt{M.pos}$ is within the geofence as constrained in *Inv*. Hence, the negation leads to unsatisfiability, and the invariant is preserved by the event `Avg`.

Discharge Proof Obligations for Controller Ports Now we describe how to discharge the proof obligation PO (*IndPort*) for controller ports. We consider the following controller assumption $\langle G_i, RS_i \rangle$ for the over-approximation of the dynamics ξ_i:

$$\begin{aligned}
G_i &\stackrel{\mathtt{def}}{=} \mathtt{p}_{\min} \leq \boldsymbol{L}_i.\mathtt{M.pos} \leq \mathtt{p}_{\max} \wedge \mathtt{p}_{\min} \leq \boldsymbol{L}_i.\mathtt{M.tgt} \leq \mathtt{p}_{\max} \\
RS_i &\stackrel{\mathtt{def}}{=} \mathtt{p}_{\min} \leq \boldsymbol{L}_i'.\mathtt{M.pos} \leq \mathtt{p}_{\max}
\end{aligned} \tag{4.17}$$

This port assumption basically states that, if both the old position $\boldsymbol{L}_i.\mathtt{M.pos}$ and the target position $\boldsymbol{L}_i.\mathtt{M.tgt}$ stay within the geofence, then the new position $\boldsymbol{L}_i'.\mathtt{M.pos}$ should also stay within the geofence.

The SMT formula φ_{Env} for proving PO (*IndPort*) is then constructed as:

$$\varphi_{Env} \stackrel{\texttt{def}}{=} Inv \wedge \bigwedge_{i \in \texttt{ID}} (G_i \Rightarrow RS_i) \wedge \neg Inv[\boldsymbol{L}_i'.Sens/\boldsymbol{L}_i.Sens] \tag{4.18}$$

Z3 is also able to answer that φ_{Env} is unsatisfiable, and thus the proof obligation is proven valid. When we manually inspect the expressions, it is straightforward to see that the invariant *Inv* is preserved. Because *Inv* implies G_i, we can ensure that RS_i is valid by the implication. Further, the actuator values, global and local memories are unchanged during the environment transition, we only need to show that the updated sensor value $\boldsymbol{L}_i'.\texttt{M.pos}$ does not violate the invariant, and this is a direct result from RS_i. Hence, it is proven that the invariant is preserved by the environment transition.

4.4 VALIDATING PORT ASSUMPTIONS: REACHABILITY ANALYSIS

To validate port assumptions, we instantiate the proof obligation PO (*PortAsm*) for Port Assumption (4.17). The proof obligation is derived as for any time point $t \in [0, \Delta T]$,

$$\left(\begin{array}{c} \mathtt{p}_{\min} \leq \boldsymbol{L}_i.\texttt{M.pos} \leq \mathtt{p}_{\max} \wedge \\ \mathtt{p}_{\min} \leq \boldsymbol{L}_i.\texttt{M.tgt} \leq \mathtt{p}_{\max} \wedge \\ \boldsymbol{L}_i'.\texttt{M.pos} = \xi_i(\boldsymbol{L}_i.\texttt{M.pos}, \boldsymbol{L}_i.\texttt{M.tgt}, t) \end{array}\right) \implies \mathtt{p}_{\min} \leq \boldsymbol{L}_i'.\texttt{M.pos} \leq \mathtt{p}_{\max} \tag{4.19}$$

It states that if the starting position and the target of the robot are within the geofence, then it remains within the geofence for the next ΔT interval.

To discharge the above proof obligation, one has to use the dynamics ξ_i of the specific robot and the specifics of the waypoint-tracking controller driving the vehicles. For the verification of LineForm, we use the reachability analysis approach, and in the remainder of this section, we give an overview of this analysis.

In this LineForm example, we also use the DryVR [23] reachability analysis tool as we mentioned in Section 3.2. Here we provide DryVR with traces from the Koord simulator of the Hector Quadrotor model [42]. Figure 4.2 shows the outputs of the reachability analysis performed on the quadrotor model. First, the large green rectangle in each plot represents the geofence defined by $\mathtt{p}_{\min}$ and $\mathtt{p}_{\max}$. Second, we generate simulated traces in the following way. In each plot, the small blue rectangle in the lower left corner representing the region to sample a starting point, and the blue rectangle in the top right corner representing the region to select a target point. We then run the simulation for a ΔT duration to obtain a simulation trace for each selected pairs of starting and target points. Note that the starting and target points match the port variables $\boldsymbol{L}_i.\texttt{M.pos}$ and $\boldsymbol{L}_i.\texttt{M.tgt}$, and the positions in

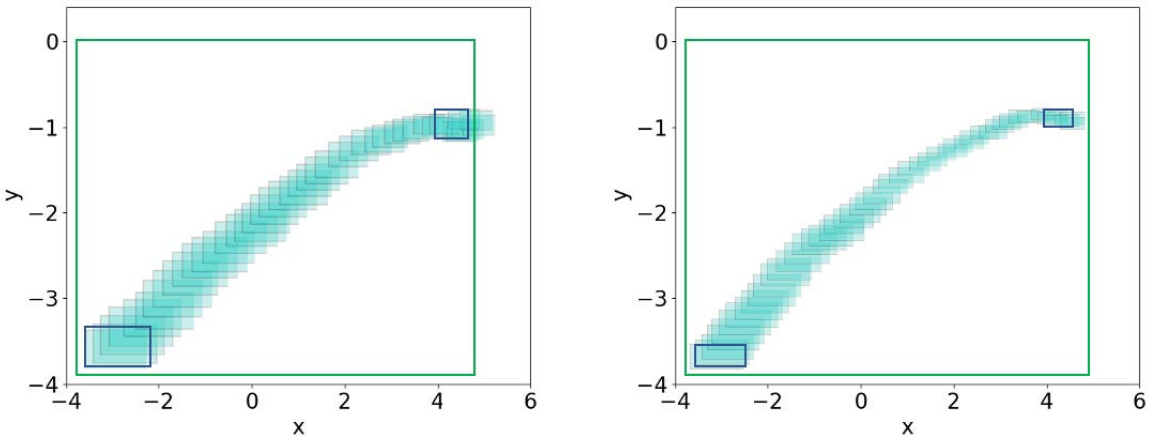


Figure 4.2: Reachset computations for Hector Quadrotors to validate port assumptions for LineForm. *Left*: The quadrotor overshoots, and the reachset is not contained within the port assumptions. *Right*: The quadrotor precisely tracks the target point, and the reachset is contained within the port assumptions.

a simulation trace matches the possible intermediate values of the new position $\boldsymbol{L}_i'.\texttt{M.pos}$ during the time horizon $[0, \Delta T]$. Third, the sequence of light green rectangles represents the generated reachsets by DryVR. These rectangles over-approximate all generated simulation traces and therefore over-approximate all possible values of the new position. We then check if these values stay within the geofence, which is equivalent to check if PO (4.19) is violated.

Figure 4.2 (*Left*) shows that the reachset of the quadcopter using a simple PID controller overshoots its target, and violates PO (4.19). Figure 4.2 (*Right*) shows that, for a quadrotor with the same controller after tuning control gains to achieve a lower settling time, the controller can track the target position more precisely with less overshoot; the controller assumption is therefore satisfied.

4.5 SUMMARY

We showed how Koord programs can be reformulated in our CPReact model as the interaction between reactive modules *RM* and the environment *Env*, so we can verify the safety of a Koord application by checking the three top level proof obligations in Theorem 2.1: (1) PO (*Init*) for initial configurations, (2) PO (*IndProg*) for program transitions, and (3) PO (*IndEnv*) for environment transitions. Further, we provided sufficient conditions to address scalability and black-box component issues. We showed in Lemma 4.2 how PO (*IndEvent*) strengthens PO (*IndProg*), and we introduced *controller assumptions*

and apply Lemma 4.3 to decompose PO (*IndEnv*) into PO (*IndPort*) and PO (*PortAsm*). In short, our formulation further simplifies and decomposes the verification for inductive invariants into four proof obligations,

(1) PO (*Init*) for checking initial configurations,

(2) PO (*IndEvent*) for checking each event in program transitions to avoid enumerating all permutations in program transitions,

(3) PO (*IndPort*) for checking controller port assumptions that abstract black-box dynamics away, and

(4) PO (*PortAsm*) for validating port assumptions.

We further demonstrated the decomposed verification approach on a formation control application, LineForm. We showed how to reuse the symbolic execution engine for KOORD from [33] to automatically generate proof obligations for the proposed inductive invariant. We then showed how to use controller assumptions to aid in verifying the inductive invariant. Finally, we showed that controller port assumptions are guard-reachset assumptions in Section 3.2, so we validated them through DRYVR.

Overall, we show how KOORD, a synchronous shared memory-based language model, is an instance of our CPREACT framework. In the next chapter, we apply our CPREACT modeling framework on Unmanned Aircraft Traffic Management systems which are distributed robotic systems with asynchronous communication.

Chapter 5: Abstractions for Unmanned Aircraft Traffic Management

In this chapter, we look into the safety analysis for the Unmanned Aircraft Traffic Management system, a multi-agent system with message passing communication. This is based on our conference paper [41].[9] The key concept for safe and efficient traffic management for Unmanned Aircraft Systems (UAS) is the notion of *operation volumes* (OVs). An OV is a 4-dimensional block of airspace and time, which can express the *intent* of an aircraft, and can be used for planning, de-confliction, and traffic management. While there are several high-level simulators for UAS Traffic Management (UTM), we are lacking a framework for creating, manipulating, and reasoning about OVs for heterogeneous air vehicles. In this chapter, we address this and present SkyTrakx—a software toolkit for simulation and verification of UTM scenarios based on OVs. Following the core idea of our CPReact model in Chapter 2, we divide a UTM system into an air traffic coordination protocol as the reactive module and the rest, such as individual air vehicle dynamics and maps of the airspace, as the environment. Subsequently, we analyze the reactive module and the environment separately with the help of an abstraction of the environment. First, we illustrate a use case of SkyTrakx by presenting a specific air traffic coordination protocol. This protocol communicates OVs between participating aircraft and an airspace manager for traffic routing. We show how existing formal verification tools, Dafny and Dione, can assist in automatically checking key invariant properties of the protocol. Especially, we show that the notion of OV abstracts away the complex air vehicle dynamics, and we use it to prove the safe separation provided by the protocol. Second, we show how to compute OVs for heterogeneous air vehicles like quadrotors and fixed-wing aircraft using another verification technique, namely *reachability analysis*. Finally, we show how to use SkyTrakx to simulate complex scenarios involving heterogeneous vehicles, for testing and performance evaluation in terms of workload and response delay analysis. Our experiments delineate the trade-off between performance and workload across different strategies for generating OVs.

5.1 OVERVIEW OF UNMANNED AIRCRAFT TRAFFIC MANAGEMENT

Unmanned Aircraft Traffic Management (UTM) is an ecosystem of technologies that aim to enable unmanned, autonomous and human-operated, air vehicles to be used for transportation, delivery, and surveillance. By 2024, 1.48 million recreational and 828 thousand

[9] This is a joint work with Dr. Hussein Sibai, Hebron Taylor, and Yifeng Ni and presented in the 25$^{\text{th}}$ IEEE International Conference on Intelligent Transportation Systems (ITSC) in 2021.

commercial unmanned aircraft are expected to be flying in the US national airspace [101]. Unlike the commercial airspace, this emerging area will have to accommodate heterogeneous and innovative vehicles relying on real-time distributed coordination, federated enforcement of regulations, and lightweight training for safety. NASA, FAA, and a number of corporations are vigorously developing various UTM concepts, use cases, information architectures, and protocols towards the envisioned future where a large number of autonomous air vehicles can safely operate beyond visual line-of-sight.

FAA's UTM ConOps [12] defines the basic principles for safe coordination in UTM and the roles and responsibilities for the different parties involved such as the vehicle operator, manufacturer, the airspace service provider, and the FAA. The building-block concept in UTM is the notion of *operation volumes (OVs)* which are used to share *intent information* that, in turn, enables interactive planning and strategic de-confliction for multiple UAS [12]. Roughly, OVs are 4D blocks of airspace with time intervals. They are used to specify the space that UAS is allowed to occupy over an interval of time (see Figures 5.1 and 5.2). While there have been small-scale field tests for UTM protocols using OVs [102], there remains a strong need for a general-purpose framework for simulating and verifying UTM protocols based on OVs. Such a framework will need to

(1) manipulate and communicate OVs for traffic management protocols,

(2) reason about dynamic OVs for establishing safety of the protocols,

(3) compute OVs for heterogeneous air vehicles performing different maneuvers, and

(4) evaluate UTM protocols in different simulation environments.

In this section, we address this need and present *SkyTrakx—an open source toolkit for simulation and verification of UTM scenarios*. The toolkit offers a framework that

(1) provides automata theory-based APIs for designing UTM protocols that formalize the communication of OVs,

(2) integrates existing tools, Dafny and Dione, to assist in verifying the safety and liveness of the protocols,

(3) uses the reachability analysis tool DryVR to compute OVs for heterogeneous air vehicles, and

(4) expands the ROS and Gazebo-based CyPhyHouse framework [55] to simulate and evaluate configurable UTM scenarios.

Benefit from our CyPhyHouse framework [55], protocols can be ported from simulations to hardware implementations. The detailed contributions of SkyTrakx are as follows:

Provably Safe De-confliction Using OVs For the first time, we show how the intention expressed as OVs can ensure provably safe distributed de-confliction in Sections 5.3 and 5.4. Following our CPReact model in Chapter 2, a UTM system is composed of an air traffic coordination protocol as the reactive module and the individual air vehicle dynamics with maps of the airspace as the environment. We can analyze the reactive module and the environment separately using OVs to abstract away the environment. As an example, we develop an automata-based de-confliction protocol using SkyTrakx APIs. This protocol specifies how the participating agents, the air vehicles, should interact with the Airspace Manager (AM). We then formally verify the safety and liveness of this protocol with OVs. In general, verification of distributed algorithms is challenging, but our safety analysis shows that the use of OVs helps decompose the global de-confliction of the UAS into local invariant on the Airspace Manager AM and local real-time requirements on each agent. We further show that Dione [103], a proof assistant for Input/Output Automata (IOA) built with the Dafny program analyzer [104], can prove the local invariant on the AM automatically.[10] We prove that the safety of the protocol is achieved when individual agents follow their declared OVs. The liveness analysis further shows that every agent can eventually find a non-conflicting OV, under a stricter set of assumptions.

Reachability Analysis for OV Conformance The guarantees of our protocol rely on proving the OVs can abstract away individual air vehicle dynamics, in other words, showing that the agents do not violate their declared OVs. We stick to the same approach presented in Section 3.2 and apply reachability analysis. In Section 5.5, we again show how to use the data-driven reachability analysis tool, DryVR [23], to create OVs for heterogeneous air vehicles with low violation probability. We apply such analysis on a quadrotor model, Hector Quadrotor [42], and a fixed-wing aircraft model, ROSplane [44], and incorporate them in SkyTrakx. We show both air vehicles in Figure 5.1 and visualize their OVs for a landing scenario in Figure 5.2.

Performance Evaluation In Section 5.6, we first discuss the implementation of SkyTrakx. Then, we perform a detailed empirical analysis of our protocol in a number of representative scenarios using SkyTrakx. We compare two strategies for the generation

[10] Dafny is based on the Z3 solver, so in essence we still prove the invariant with SMT solvers according to Corollary 2.2.

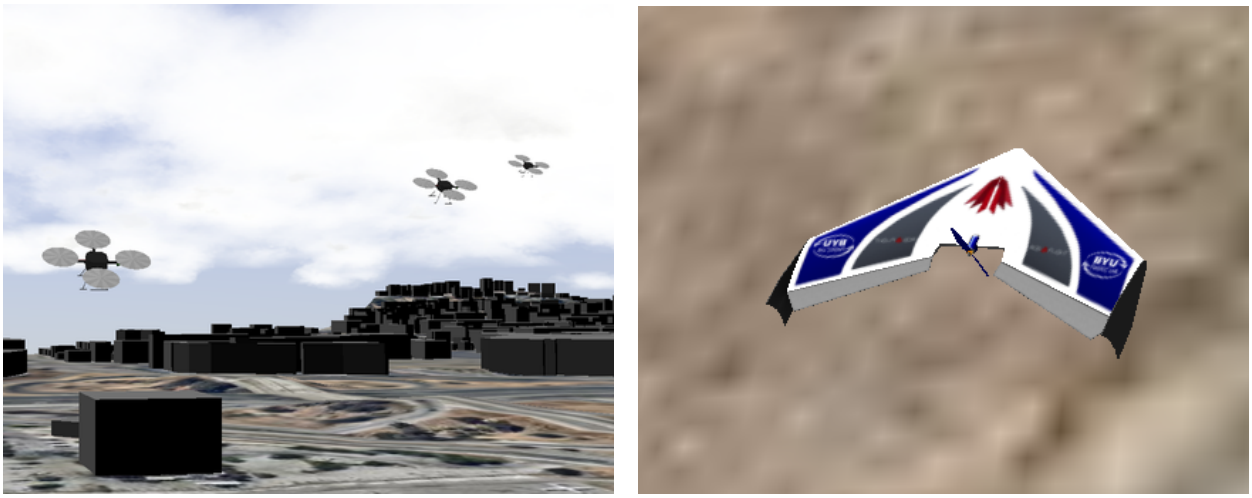


Figure 5.1: Hector Quadrotor [42] (*Left*) and ROSplane [44] (*Right*) models in Gazebo simulator.

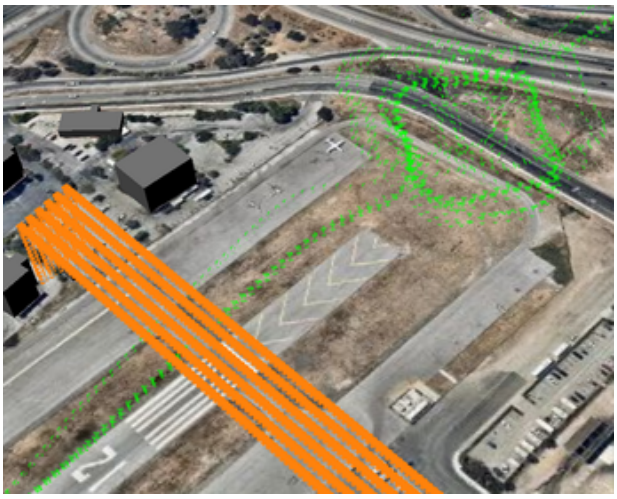
of OVs with different aggressiveness, namely Conservative and Aggressive. Our experiments quantify the performance and workload on the AM, and we measure these metrics with respect to the number of participating agents and different strategies for generating OVs. Our results suggest that the workload on the AM scales linearly with the number of agents, and Aggressive provides 1.5-3X speedup but leads to 2-5X increased workload on the airspace manager AM.

5.2 RELATED WORKS

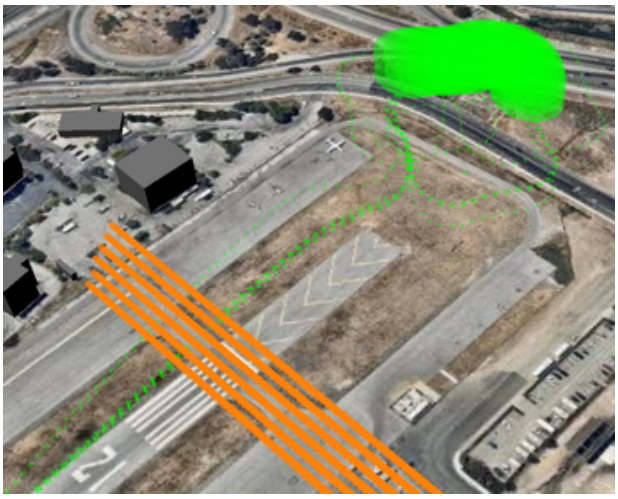
Collision Avoidance Protocols Prior to the development of the UTM ecosystem, traffic management protocols for manned aircraft include the family of Traffic Alert and Collision Avoidance Systems (TCAS) [105, 106, 107, 108, 109, 110].

UTM and TCAS are complementary—the former is for long range strategic safety against loss of separation with other aircraft and static obstacles, weather events, and anomalous behaviors, while the latter is for shorter-range tactical safety. Accordingly, the protocol we discuss (in Section 5.4) coordinates over longer range and not *only* for potential collision avoidance. SkyTrakx could be augmented with existing collision avoidance protocols in the future. For instance, if an aircraft violates its operation volume in our protocol, then a TCAS-like protocol can be used to avoid collision.

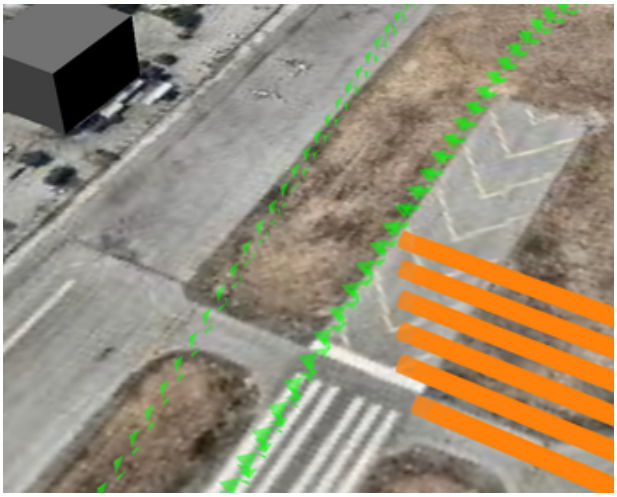
Formal Approaches to UTM and Collision Avoidance The formal methods' research community has engaged with the problem of air-traffic management in a number of different ways. There have been several works on formal analysis of TCAS [111, 112, 113, 114], ACAS



(a) ROSplane reserved OVs for loitering and descending.



(b) ROSplane loiters and waits for Quadrotors.



(c) Quadrotors passed the runway before ROSplane descends.

(d) ROSplane descends.

Figure 5.2: Visualization of a landing scenario with heterogeneous air vehicles in an airport. The OVs for Hector Quadrotors are annotated with orange and OVs for the ROSplane are shown in green. Reserved OVs are outlined with dots, and OVs in use are represented with solid tubes.

X [75, 115, 116], and other protocols [117, 118, 119, 120, 121, 122].[11] These verification efforts rely on various simplifying assumptions such as precise state estimates, straight-line trajectories, constant velocity of the intruder and ownership. Algorithms to synthesize safe-by-construction plans for multiple drones flying in a shared airspace have been developed in [53, 55, 123, 124]. These approaches rely on predicting and communicating future behavior of participating aircraft under different sources of uncertainty [53, 119, 123].

[11] `https://ti.arc.nasa.gov/news/acasx-verification-software/`

For example, [119] uses reachability analysis to check for collisions under sensing and synchronization errors. The work in [123] uses Signal Temporal Logic (STL) to synthesize trajectories that are far from predicted trajectories shared by other drones. It accounts for discretization errors of continuous trajectories while restricting them to be simple (straight lines, or with free end velocities but zero acceleration). In [53], the authors propose a framework for the synthesis of safe drone trajectories given those of the other drones while accounting for synchronization errors.

In [125], the authors present an approach for decentralized policy synthesis for route planning of individual vehicles modeled as Markov decision processes. Our approach decouples the low-level dynamically feasible planning from the distributed coordination, and solves the latter problem using a centralized coordinator (Airspace Manager) via distributed mutual exclusion over regions of the airspace (Section 5.4). In [126], the authors present a framework for decentralized controller synthesis for different managers of neighboring airspaces. They use finite game and assume-guarantee approaches to generate decision-making mechanisms that satisfy linear temporal logic specifications. An application of their approach is to design policies for airspace managers that enforce a maximum number of vehicles in the airspace or maximum loitering time. Their framework assumes the operating regions for actions such as takeoff or loitering are predefined. Our framework is complementary to this work as we show how a vehicle can generate an OV based on its vehicle dynamics from infinite choices of regions and time.

In [119], the authors present a protocol for online decentralized safety checking for a swarm of drones accomplishing individual goals while avoiding static and cross-agent collision. Different drones share their bounded time reachsets, projected to the time and 3D space, periodically. Each drone checks the intersection of its projected reachset with other drones' projected reachsets while accounting for time delays. A projected reachset is an example of an OV, and their method is an example of an OV abstraction-based protocol. Their total verification time per period scales linearly with the number of agents. They achieve real-time performance by using the face-lifting method to compute reachsets and representing reachsets as a single hyper-rectangle.

5.3 A FORMAL MODEL OF OPERATION VOLUMES

In this section, we formalize the notion of OVs described in [12] which is the fundamental building block for UTM protocols. This formalization is also implemented in SkyTrakx for creating, manipulating, and reasoning about OVs. We refer to a UAS participating in the UTM system as an *agent*, or equivalently, an *air vehicle*. Every agent in the system

has a unique identifier. The set of all possible identifiers is ID. We assume that each agent has access to a common global clock which takes non-negative real numbers. The *airspace* is modeled as a compact subset $\mathcal{W} \subseteq \mathbb{R}^3$. Large airspaces may have to be divided into several smaller airspaces, and one has to deal with hand-off across airspaces. In this chapter, we do not handle this problem of air vehicles entering and leaving $\mathcal{W}$. Other works have synthesized safe protocols for this problem (e.g., [126]). The airspace is different from the state space of individual air vehicles which may have many other state components like velocity, acceleration, pitch and yaw angles, etc. Informally, an OV is a schedule for an air vehicle for occupying airspace.

Definition 5.1. An *operating volume (OV)* is a sequence of pairs $C = (\mathcal{R}_1, \tau_1), (\mathcal{R}_2, \tau_2), \ldots, (\mathcal{R}_k, \tau_k)$ where each $\mathcal{R}_i \subseteq \mathcal{W}$ is a compact subset of the airspace, and τ_i's is a monotonically increasing sequence of time points.

The total *time duration* $\tau_k - \tau_1$ of the OV C is denoted by $C.dur$, and the length k of C is denoted by $C.len$. Further, we denote the last time point τ_k by $C.\tau_{last}$, the last region $\mathcal{R}_k$ by $C.\mathcal{R}_{last}$, and the union of all regions, $\bigcup_{i=1}^{k} \mathcal{R}_i$, by $C.\mathcal{R}_{all}$. We denote the set of all possible contracts as **OV**. An air vehicle meets an OV at real-time t if

(1) $t \in [\tau_i, \tau_{i+1})$ for any $i < k$ implies that the air vehicle is located within $\mathcal{R}_i$, and

(2) $t \geq \tau_k$ implies that the agent is located within $\mathcal{R}_k$ *ever after* τ_k.

Formally,

Definition 5.2. Any OV C represents a compact subset $[\![C]\!]$ of space-time:

$$[\![C]\!] \stackrel{\texttt{def}}{=} \bigcup_{i=1}^{k-1} \{(r,t) \mid r \in \mathcal{R}_i \wedge \tau_i \leq t < \tau_{i+1}\} \cup \{(r,t) \mid r \in \mathcal{R}_k \wedge \tau_k \leq t\} \tag{5.1}$$

Further, given the current position *pos* and clock reading *clk* of an air vehicle, we say that the air vehicle meets the contract C if and only if $(pos, clk) \in [\![C]\!]$.

5.3.1 OVs are Closed under Set Operations

We now show that OVs are closed under all set operations by first defining basic operations to align time points of any two OVs. Then we show that OVs are closed under intersection, union, and set difference.

Definition 5.3. Two OVs are *time-aligned* if they use the same sequence of time points. Given two time-aligned OVs, $C^a = (\mathcal{R}^a_1, \tau_1), \ldots, (\mathcal{R}^a_k, \tau_k)$ and $C^b = (\mathcal{R}^b_1, \tau_1), \ldots, (\mathcal{R}^b_k, \tau_k)$, and a set operation $\oplus \in \{\cap, \cup, \setminus\}$, we define

$$C^a \oplus C^b \stackrel{\texttt{def}}{=} (\mathcal{R}^a_1 \oplus \mathcal{R}^b_1, \tau_1), \ldots, (\mathcal{R}^a_k \oplus \mathcal{R}^b_k, \tau_k). \tag{5.2}$$

We now generalize the definition to OVs that are not *time-aligned*.

Definition 5.4. Given any OV $C = (\mathcal{R}_1, \tau_1), \ldots, (\mathcal{R}_k, \tau_k)$, $\mathit{prepend}(C, \tau_{pp})$ where $\tau_{pp} < \tau_1$, $\mathit{split}(C, \tau_{sp})$ where $\tau_i < \tau_{sp} < \tau_{i+1}$, and $\mathit{append}(C, \tau_{ap})$ where $\tau_k < \tau_{ap}$ are defined as,

$$\begin{aligned}
\mathit{prepend}(C, \tau_{pp}) &\stackrel{\texttt{def}}{=} \boldsymbol{(\emptyset, \tau_{pp})}, (\mathcal{R}_1, \tau_1), \ldots, (\mathcal{R}_k, \tau_k) \\
\mathit{split}(C, \tau_{sp}) &\stackrel{\texttt{def}}{=} (\mathcal{R}_1, \tau_1), \ldots, (\mathcal{R}_i, \tau_i), \boldsymbol{(\mathcal{R}_i, \tau_{sp})}, (\mathcal{R}_{i+1}, \tau_{i+1}), \ldots, (\mathcal{R}_k, \tau_k) \\
\mathit{append}(C, \tau_{ap}) &\stackrel{\texttt{def}}{=} (\mathcal{R}_1, \tau_1), \ldots, (\mathcal{R}_k, \tau_k), \boldsymbol{(\mathcal{R}_k, \tau_{ap})}
\end{aligned} \tag{5.3}$$

Finally, we define $\mathit{insert}(C, T)$ function over any T,

$$\mathit{insert}(C, T) \stackrel{\texttt{def}}{=} \begin{cases} \mathit{prepend}(C, T) \text{ if } T < \tau_1 \\ \mathit{split}(C, T), \text{ if } \tau_i < T < \tau_{i+1} \\ \mathit{append}(C, T), \text{ if } \tau_k < T \\ C, \text{otherwise.} \end{cases} \tag{5.4}$$

Lemma 5.1. *By definition, the OV produced by prepend, split, append, and insert functions represents the same set of space-time by C. That is, given any OV C and time point T,*

$$[\![\mathit{insert}(C, T)]\!] = [\![C]\!] \tag{5.5}$$

With the help of *insert*, we can always align two OVs. We can then implement intersection, union, and difference on OVs on top of the same operators for airspace.

Proposition 5.2. *Given any OV C^a and C^b, any set operation $\oplus \in \{\cap, \cup, \setminus\}$, we have the following equivalences:*

$$[\![C^a \oplus C^b]\!] = [\![C^a]\!] \oplus [\![C^b]\!]. \tag{5.6}$$

The proof is to expand the definition of $[\![\cdot]\!]$ and skipped here. Given Proposition 5.2, OVs are closed under all set operations; hence we drop the $[\![\cdot]\!]$ notation.

Several concepts are defined naturally as set operations on OVs. We abuse the notation

sometimes and use C as the set represented by the contract C, i.e. the set

$$C \stackrel{\text{def}}{=} \bigcup_{i=1}^{k-1} \{(r,t) \mid r \in \mathcal{R}_i \wedge \tau_i \leq t < \tau_{i+1}\} \cup \{(r,t) \mid r \in \mathcal{R}_k \wedge \tau_k \leq t\}. \tag{5.7}$$

For example, checking if C^a *refines* C^b is to simply check if C^a uses less space-time than C^b does, i.e., $C^a \subseteq C^b$, or equivalently $C^a \setminus C^b = \emptyset$.

We will use the defined operations in our protocol in Section 5.4 to update OVs of individual agents and check intersections. We will show how to create such OVs using reachability analysis in Section 5.5.

5.4 A SIMPLE COORDINATION PROTOCOL USING OVS

We present an example protocol for safe traffic management using OVs and its correctness argument. We further implement the protocol with SkyTrakx. The protocol involves a set of agents communicating OVs with an *airspace manager or controller (AM)*. The overall system is the composition of the airspace manager (AM) and all agents ($agent_i$):

$$Sys \stackrel{\text{def}}{=} AM||\{agent_i\}_{i \in \texttt{ID}}. \tag{5.8}$$

In Section 5.4.1 and 5.4.2, we describe the protocol by showing the interaction between participating agents and the AM through `request`, `reply`, and `release` messages. We then analyze the safety of the protocol under instant message delivery in Section 5.4.3, and its liveness in Section 5.4.5.

```
automaton AirspaceManager

variables:
    contr_arr: [ID → OV]
    reply_set: Set⟨ID⟩

output reply_i(contr: OV = contr_arr[i])
    pre: i ∈ reply_set
    eff: reply_set := reply_set \ {i}

input request_i(contr: OV)
    eff:
        reply_set := reply_set ∪ {i}
        if ⋀_{j≠i, j∈ID} (contr ∩ contr_arr[j] = ∅):
            contr_arr[i] := contr_arr[i]∪contr

input release_i(contr: OV)
    eff: contr_arr[i] := contr_arr[i] \ contr
```

Figure 5.3: Airspace Manager automaton.

5.4.1 Airspace Manager

We design the AM as an Input/Output Automaton (IOA) [19] defined in Figure 5.3. The AM keeps track of all contracts and checks for conflicts before approving new contracts. It uses a mapping `contr_arr` in which `contr_arr[i]` records the contract held by the agent i, and a set `reply_set` to store the IDs of the agents whose requests are being processed and pending reply.

Whenever the AM receives a $\texttt{request}_i(\texttt{contr})$ from agent i (Line 11), i is first added to `reply_set`. Then, `contr` is checked against all contracts of other agents by checking disjointness (Line 14). Only if the check succeeds, `contr` is included in `contr_arr[i]` via set union (Line 15).

When i is in `reply_set`, the $\texttt{reply}_i(\texttt{contr})$ action is triggered to reply to agent i with the recorded `contr`=`contr_arr[i]` (Line 7). Note that the AM replies with the *recorded contract* `contr_arr[i]` at Line 7 irrespective of whether the *requested contract* `contr` in Line 11 was included in `contr_arr[i]` or not. Finally, if the AM receives a $\texttt{release}_i(\texttt{contr})$, then it removes `contr` from `contr_arr[i]` via set difference (Line 18).

5.4.2 Agent's Protocol

The agent's coordination protocol sits in between a *planner/navigator* that proposes OVs and a *controller* which drives the air vehicle to its target. We will discuss approaches to estimate OVs for waypoint-based path planners and waypoint-following controllers in Section 5.5. Figure 5.4 shows the simplified state diagram of the agent protocol. At a high level, agent i's protocol starts in the idle state and initiates when a `plan` action with a given `contr` is triggered by the agent's planner. Then, the protocol requests this contract from the AM, and waits for the reply. If the requested contract is a subset of the one replied by the AM, the agent protocol enters the moving state. At this point, the agent's controller starts moving the air vehicle and ideally making it follow the contract strictly. Once the air vehicle reaches the last region of OV successfully, the protocol releases the unnecessary portion of the contract and goes back to idle state. In the case that the requested contract is not a subset of the one replied by the AM, the protocol directly releases and retries. If the agent violates the contract while moving, it notifies the AM that the contract is violated. We provide the formally specified automaton and detail explanations of the agent's protocol in Figure 5.5.

The detailed automaton is shown in Figure 5.5. The agent protocol has a `status` variable to keep track of the discrete states in Figure 5.4. In addition, it uses three contract-typed

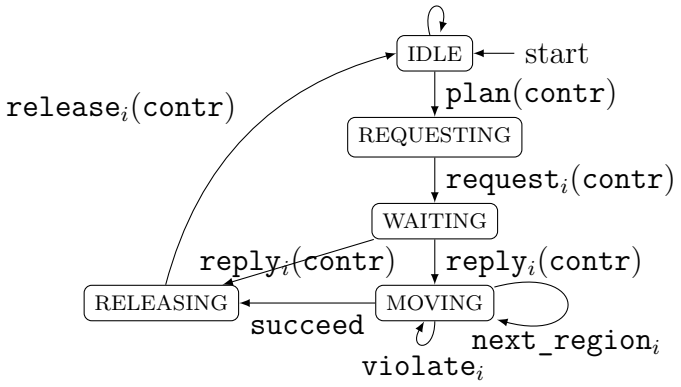


Figure 5.4: Simplified state diagram for Agent.

```
1  automaton Agent_i
2  variables: // Discrete variables
3     status: {IDLE,REQUESTING,WAITING,
4           MOVING,RELEASING} := IDLE
5     curr_contr: OV
6     plan_contr: OV
7     free_contr: OV
8
9  input plan_i(contr: OV)
10    eff:
11       if status = IDLE:
12          plan_contr := contr
13          status := REQUESTING
14
15 output request_i(contr: OV = plan_contr)
16    pre: status = REQUESTING
17    eff: status := WAITING
18
19 input reply_i(contr: OV)
20    eff:
21       if status = WAITING:
22          if curr_contr ⊈ contr:
23             warning("Contract too small")
24             curr_contr := contr
25          if plan_contr ⊆ contr:
26             curr_contr := contr
27             status := MOVING
28          else:
29             free_contr := contr\curr_contr
30             status := RELEASING
32 variables: // Continuous variables
33    clk: ℝ_≥0
34    pos: ℝ^3 // Position sensor
35
36
37 output next_region_i():
38    pre: status = MOVING ∧ len(plan_contr)≥2
39       ∧ clk ≥ plan_contr.T_2
40    eff: plan_contr.pop_front()
41
42 internal succeed():
43    pre: status = MOVING ∧ len(plan_contr)=1
44       ∧ (pos, clk) ∈ plan_contr
45    eff:
46       free_contr := curr_contr\plan_contr
47       status := RELEASING
48
49 output violate_i():
50    pre: status=MOVING ∧ (pos,clk) ∉ curr_contr
51    eff:
52       error("Current contract is violated")
53
54 output release_i(contr: OV = free_contr)
55    pre: status = RELEASING
56    eff:
57       curr_contr := curr_contr \ free_contr
58       status := IDLE
```

Figure 5.5: Agent automaton

variables for the following purposes:

(1) `curr_contr` is a local copy of the current contract maintained for i by the AM,

(2) `plan_contr` is a contract that i wants to propose to the AM to be able to visit the planned waypoints, and

(3) `free_contr` tracks the releasable portion of the current contract `curr_contr`.

In addition, the agent i can read its current position from the variable `pos` and the current global time from the variable `clk`. To provide a simple abstraction of arbitrary controllers for the agent, we create the variable `traj_ctrl` that stores a list of waypoints that the agent would follow when it is in the `MOVING` status. `traj_ctrl` has two abstract interfaces: `set_waypoints` to store the plan waypoints and calculate the necessary control signal (using PID, for example) and `start` to start moving the agent to follow waypoints.

Each agent i is initialized in `IDLE` status. When it receives a `plan` action with a given `contr` (Line 9), the agent stores `contr` as `plan_contr` (Line 12) and enters the `REQUESTING` status (Line 13). A number of strategies may be followed to create contracts from waypoints lists, for example using reachability analysis for a given waypoint-tracking controller for the aircraft, or creating fixed-sized 3D rectangles centered at the segments connecting the waypoints. We will discuss this further in Section 8.5. Agent i then makes a request $\texttt{request}_i(\texttt{contr})$ with `contr`=`plan_contr` to denote the planned contract is sent as output, and enters `WAITING` status to wait for a reply from the AM (Line 15).

When agent i receives a $\texttt{reply}_i(\texttt{contr})$ from the AM, the contract `contr` represents the contract of agent i recorded by the AM (Line 19). It is the union of all contracts agent i have acquired and not yet released. Agent i first checks whether the contract `curr_contr` is a subset of `contr` or not. If not, it means the local copy is less restrictive, so the AM may grant contracts to other agents conflicting with agent i. This may lead to a safety violation, and hence agent i raises a warning (Line 19). Otherwise, the agent checks if the contract `contr` approved by the AM contains `plan_contr`, i.e. $\texttt{plan\_contr} \subseteq \texttt{contr}$ (Line 25). If yes, then it updates its `curr_contr` to be equal to the new approved `contr`. The agent then calls `traj_ctrl.start` to start following the waypoints, and transitions to the `MOVING` status. If no, i.e. there is a part of `plan_contr` that is not approved `contr` and not approved by the AM, then agent i does not change `curr_contr`. It only checks the part of the contract saved by the AM that is no longer a part of `curr_contr` of the agent. It then stores this portion of the contract in `free_contr` (Line 29), and directly goes to the `RELEASING` status to release and re-plan (Line 30).

When the agent is in the `MOVING` status, the `next_region` action will be triggered whenever the global time passes the time bound of a region in the contract (Line 37). That action will remove that pair of region and time point from `plan_contr` (Line 40). Once there is only a single pair left in the planned contract `plan_contr` and the contract is not violated, the `succeed` action is triggered to indicate the plan is executed successfully (Line 42). Agent i then calculates the releasable contract `free_contr` to be its contract `curr_contr` excluding the last pair of `plan_contr` (Line 46). Finally, it enters `RELEASING` status. It sends $\texttt{release}_i(\texttt{contr})$ to notify the AM the contract that agent i can release, and goes back to

IDLE status (Line 58).

If at any point in time the current contract is violated, the violate action would be triggered (Line 49). Remember that the contract is violated if the current pair of position and time of the agent is outside the space-time specified by the contract. This can happen in case the agent moves outside a region in a time interval of the contract, or the agent could not reach a region before its specified time point in the contract. It then declares a violation to the AM. For simplicity, we skip the protocol after entering STOPPING status. In practice, recovery actions or emergency landing can be implemented.

5.4.3 Protocol Correctness: Safety under Instantaneous Delivery

We now discuss the safety property ensured by our protocol. Here, $agent_i$.curr_contr denotes the contract that the i^{th} agent is following. Assuming that none of the agents triggered their violate action, then an agent always follows its local contract curr_contr. In that case, collision avoidance is defined naturally as the disjointness between the curr_contrs of all agents. Our goal therefore is to show that the following proposition is an invariant of the system:

Proposition 5.3 (Safety)**.** *If none of the agents triggered their* violate *action, the current contracts followed by all agents are pairwise disjoint, i.e., we show the Safe predicate below is valid:*

$$Safe \stackrel{\text{def}}{=} \bigwedge_{i \in ID} \bigwedge_{j \neq i, j \in ID} agent_i.\texttt{curr\_contr} \cap agent_j.\texttt{curr\_contr} = \emptyset. \tag{5.9}$$

Our proof strategy is to show that first the global record of contracts maintained by the AM are pairwise disjoint by Lemma 5.4. Then, we ensure the local copy by each agent is as restrictive as the global record and hence preserves disjointness by Lemma 5.5. With Lemma 5.4 and Lemma 5.5, Proposition 5.3 is derived following basic set theory. We start from Lemma 5.4 for the AM.

Lemma 5.4. *If none of the agents triggered their* violate *action, all contracts recorded by the AM are pairwise disjoint, i.e., we show the Inv predicate below is valid:*

$$Inv \stackrel{\text{def}}{=} \bigwedge_{i \in ID} \bigwedge_{j \neq i, j \in ID} AM.\texttt{contr\_arr}[i] \cap AM.\texttt{contr\_arr}[j] = \emptyset. \tag{5.10}$$

Proof. This is a direct result from examining all actions of the AM automaton. The $\texttt{request}_i$ action ensures that a contr is only included into contr_arr[i] if it is disjoint from all other contracts contr_arr[j]. The $\texttt{reply}_i$ action does not modify contr_arr at all, and $\texttt{release}_i$ action only shrinks the contracts. QED.

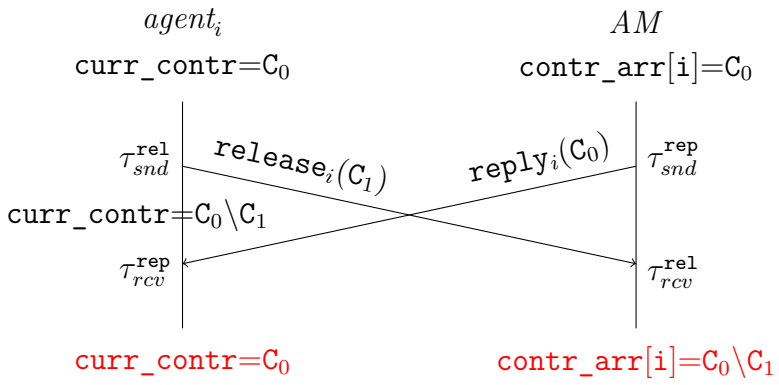


Figure 5.6: Sequence diagram for an impossible unsafe OV release.

Lemma 5.5. *If none of the agents triggered their* `violate` *action, the* `curr_contr` *of agent* i *is always as restrictive as* `contr_arr[i]`*, i.e., the following auxiliary predicate Aux is valid:*

$$Aux \stackrel{\text{def}}{=} \bigwedge_{i \in ID} agent_i.\texttt{curr\_contr} \subseteq AM.\texttt{contr\_arr}[i]. \tag{5.11}$$

Proof. This is proven by examining all actions of the agent automaton regardless of the order of execution. Due to the space limit, we only consider when actions are delivered instantaneously. The `curr_contr` is only modified in `reply` and `release` actions. In `reply` action, `curr_contr` is to copy `contr` sent by the AM and thus Lemma 5.5 holds. In `release` action, `curr_contr` removes `contr` first; then `release` is delivered to the AM to remove `contr`. As a result, Lemma 5.5 still holds. QED.

5.4.4 Protocol Correctness: Safety under Bounded Delay

Now we consider the case where actions are delivered with bounded delay under the reliable communication assumption. Our proof is to show the impossibility of unsafe action sequences under the reliable communication. Because the current contract of each agent is only updated after receiving $\texttt{reply}_i$ from the AM and shrunk when sending $\texttt{release}_i$, the potential counterexample shown in Figure 5.6 can only happen if $\texttt{reply}_i$ is delivered to agent i to update its local copy while $\texttt{release}_i$ is delivered to the AM to shrink the global copy concurrently, i.e., $\tau^{\texttt{rel}}_{snd} < \tau^{\texttt{rep}}_{rcv}$ and $\tau^{\texttt{rep}}_{snd} < \tau^{\texttt{rel}}_{rcv}$. Recall from Figure 5.4 that our protocol ensures $\texttt{request}_i$, $\texttt{reply}_i$, and $\texttt{release}_i$ happen in such order by design. We can prove this order of actions by induction on the formally defined automaton but skip the proof here for simplicity. Therefore, we know that there must be a $\texttt{request}_i$ sent after $\texttt{release}_i$, and $\texttt{reply}_i$ is the response to this request. Now we provide a simplified reliable communication

assumption for this proof.

Assumption 5.1. The reliable communication guarantees the messages sent by the same agent are delivered in order. In particular, if $agent_i$ sends a $\texttt{release}_i$ first and $\texttt{request}_i$ second, we denote $\tau_{snd}^{\texttt{rel}}$ as the time $\texttt{release}_i$ is sent and $\tau_{rcv}^{\texttt{rel}}$ as the time received, similarly $\tau_{snd}^{\texttt{req}}$ and $\tau_{rcv}^{\texttt{req}}$ for $\texttt{request}_i$. Formally,

$$\tau_{snd}^{\texttt{rel}} \leq \tau_{snd}^{\texttt{req}} \Rightarrow \tau_{rcv}^{\texttt{rel}} \leq \tau_{rcv}^{\texttt{req}} \tag{5.12}$$

Also by definition, $T_{snd}^{*} \leq T_{rcv}^{*}$ because sending must happen before receiving. The order between actions can be formally specified as $\tau_{snd}^{\texttt{rel}} \leq \tau_{snd}^{\texttt{req}} \leq \tau_{rcv}^{\texttt{req}} \leq \tau_{snd}^{\texttt{rep}}$ because the request must have been delivered to the AM for it to trigger the $\texttt{reply}_i$. We can then derive $\tau_{snd}^{\texttt{rel}} \leq \tau_{snd}^{\texttt{req}} \leq \tau_{rcv}^{\texttt{req}} \leq \tau_{snd}^{\texttt{rep}} < \tau_{rcv}^{\texttt{rel}}$. This contradicts to our assumption of reliable communication because messages from $agent_i$ are delivered out of order. To be more precise, $\texttt{release}_i$ is sent before ($\tau_{snd}^{\texttt{rel}} \leq \tau_{snd}^{\texttt{req}}$) but delivered later ($\tau_{rcv}^{\texttt{req}} < \tau_{rcv}^{\texttt{rel}}$) than $\texttt{request}_i$. This contradicts to $\tau_{snd}^{\texttt{rel}} \leq \tau_{snd}^{\texttt{req}} \Rightarrow \tau_{rcv}^{\texttt{rel}} \leq \tau_{rcv}^{\texttt{req}}$. Hence, we prove by contradiction.

5.4.5 Protocol Correctness: Liveness

For liveness property, we would like to see every agent eventually reaches its target. In our protocol, this is formulated as every agent eventually reaches the last region of its OV that it proposed in `plan` action and triggers its `succeed` action. The overall proof is to show that an agent can always find an OV which the AM approves.

Since a newly proposed OV may be rejected, we denote it as `plan_contr` to distinguish from `curr_contr` which an agent always follows. It is worth noting that liveness depends on the OV for each agent. A simple scenario where liveness cannot be achieved is when the final destinations of two agents are too close; thus the last region where one agent stays at the end could block the other agent forever. Therefore, we first require the following assumption:

Assumption 5.2 (Disjointness of different agents' regions)**.** For any agent $i \in \texttt{ID}$, all regions that it plans to traverse are disjoint from the last regions of all other agents. Formally,

$$\bigwedge_{j \neq i} \texttt{plan\_contr}_i.\mathcal{R}_{all} \cap AM.\texttt{contr\_arr}[j].\mathcal{R}_{last} = \emptyset. \tag{5.13}$$

Assumption 5.2 can be achieved by querying the AM when planning since Lemma 5.5 ensures the AM's record of OVs includes the agents' OVs.

Definition 5.5. Given an OV $C = (\mathcal{R}_1, \tau_1), \ldots, (\mathcal{R}_k, \tau_k)$ and a time duration δ, we define *reschedule*(C, δ) as:

$$\textit{reschedule}(C, \delta) \stackrel{\texttt{def}}{=} (\mathcal{R}_1, \tau_1 + \delta), (\mathcal{R}_2, \tau_2 + \delta), \ldots, (\mathcal{R}_k, \tau_k + \delta) \tag{5.14}$$

Now we start our argument for liveness. By our protocol design, if agent i never violates its OV, it must reach the last region successfully. Therefore, we only have to prove that agent i's request to the AM must be accepted eventually. With Assumption 5.2, we prove the claim that an agent i can always *reschedule* a plan so that the AM approves its OV.

Proposition 5.6 (Liveness)**.** *If* $\texttt{plan\_contr}_i$ *satisfies Assumption 5.2, then there is a time duration* δ_0 *such that the AM approves reschedule*$(\texttt{plan\_contr}_i, \delta)$ *for all* $\delta \geq \delta_0$*. Formally,*

$$\bigwedge_{j \neq i, j \in ID} \textit{reschedule}(\texttt{plan\_contr}_i, \delta) \cap AM.\texttt{contr\_arr}[j] = \emptyset. \tag{5.15}$$

Proof. Following Assumption 5.2, we first derive the disjointness of regions of airspace. For any $j \neq i$ and any δ,

$$\textit{reschedule}(\texttt{plan\_contr}_i, \delta).\mathcal{R}_{all} \cap \ AM.\texttt{contr\_arr}[j].\mathcal{R}_{last} = \emptyset, \tag{5.16}$$

because *reschedule* does not modify the regions. Further, we derive that the following two OVs are disjoint for any $\delta_j \geq AM.\texttt{contr\_arr}[j].\tau_{last}$:

$$\textit{reschedule}(\texttt{plan\_contr}_i, \delta_j) \cap AM.\texttt{contr\_arr}[j] = \emptyset. \tag{5.17}$$

The proof is to expand the definition and is skipped here. Intuitively, this is because every agent j is expected to reach and stay in $AM.\texttt{contr\_arr}[j].\mathcal{R}_{last}$ ever after $\delta_j \geq AM.\texttt{contr\_arr}[j].\tau_{last}$. Therefore, the rescheduled OV for agent i does not overlap with OVs of any other agent j.

Finally, let $\delta_0 \stackrel{\texttt{def}}{=} \max_{j \neq i} \ AM.\texttt{contr\_arr}[j].\tau_{last}$ and it directly leads to the proof of Proposition 5.6. QED.

In addition to the manual proof presented, we have also explored using Dione [103] with Dafny proof assistant [104] to generate induction proofs for invariants of IOA. We chose this tool due to its support for IOA and automated SMT solving for set operations on OVs. We discovered that these tools can automatically prove the local invariant Lemma 5.4 for the AM. However, they lack support for continuous time to model agents and communication delay; hence we cannot use Dione to prove other lemmas and propositions directly.

5.5 REACHABILITY ANALYSIS AND OPERATION VOLUMES

In Section 5.4, we show that the protocol ensures safety and liveness. However, the proof assumes that the air vehicle does not violate its OV. In this section, we discuss how to use existing reachability analyses to over-approximate regions of space-time that an air vehicle may visit. This over-approximation can be used to

(1) generate OVs that are unlikely to be violated, or

(2) monitor air vehicles at runtime to predict and avoid possible violations.

Formally, given a dynamical system with state space $\mathcal{X}$, a set of initial states $\mathcal{X}_0 \subseteq \mathcal{X}$, and a time horizon $[\tau_0, \tau_1)$, reachability analysis tools can compute *reachset*, a set of states $\mathcal{X}_1$ reachable within $[\tau_0, \tau_1)$. We further require a function $\hat{\pi} : 2^{\mathcal{X}} \mapsto 2^{\mathcal{W}}$ to transform state space to air-space. Then, one can build an OV $C_{reach} = (-\infty, \hat{\pi}(\mathcal{X}_0)), (\tau_0, \hat{\pi}(\mathcal{X}_1)), (\tau_1, \mathcal{W})$. This means that when air vehicle stays within $\hat{\pi}(\mathcal{X}_0)$ before τ_0, it will then stay within $\hat{\pi}(\mathcal{X}_1)$ between τ_0 and τ_1, and it can be anywhere after τ_1. We then can merge C_{reach} for different time horizons to propose OVs.

In this chapter, we use DryVR [23] to compute reachtubes from simulation traces as we discussed in Chapter 2. DryVR uses collected traces to learn the sensitivity of the trajectories of the air vehicle, and generates reachtubes for a new simulation trace with probabilistic guarantees. We use DryVR to generate OVs for a quadrotor model, Hector Quadrotor [42], and a fixed-wing model, ROSplane [44], using the Gazebo simulator.

Hector Quadrotor The state variables for Hector Quadrotor already include x, y, and z for its position. They also include other variables for orientation and velocity. Hence, $\hat{\pi}$ for this model is to simply apply projections to the x, y, and z axes. We compute C_{reach} for a scenario where the air vehicle follows the waypoint (0, 0, 2.5). Figure 5.7 shows the projection of C_{reach} as hyper-rectangles to the xy-plane (*Left*) and to the z-axis against time (*Right*). We can generate OVs using a Conservative strategy that covers C_{reach} for the entire time horizon with a bounding rectangle, or an Aggressive strategy to use the gray rectangles as an OV with short time intervals. In general, we can generate a spectrum of OVs from C_{reach} between Conservative and Aggressive strategies, and all OVs in this spectrum can guarantee, using reachability analysis, a low probability of violations. We further explore the performance trade-off between the two strategies in Section 8.5.

ROSplane Similarly, the state variables for ROSplane include x, y, and z representing its position but in North-East-Down (NED) coordinates. Hence, $\hat{\pi}$ for this model is to

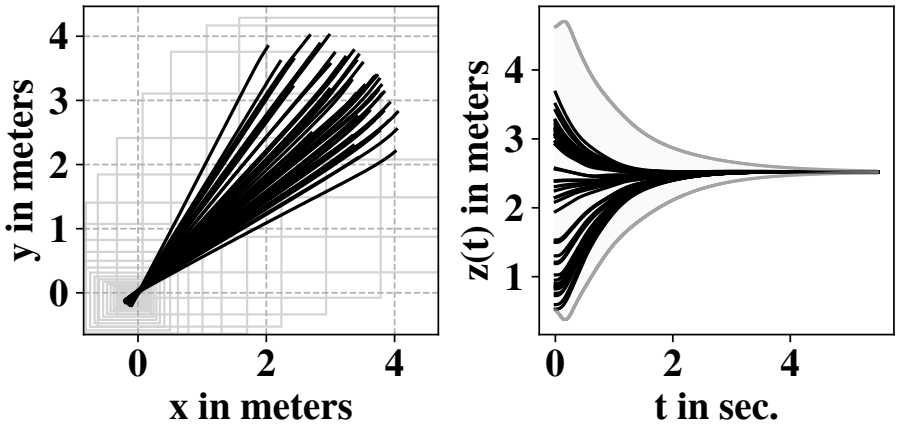


Figure 5.7: Simulation traces in *Black* and boundary of the reachtube computed by DryVR in *Gray* for Hector Quadrotor going to the waypoint at (0, 0, 2.5). The reachtube is projected to xy-plane (*Left*) and z-axis over time (*Right*).

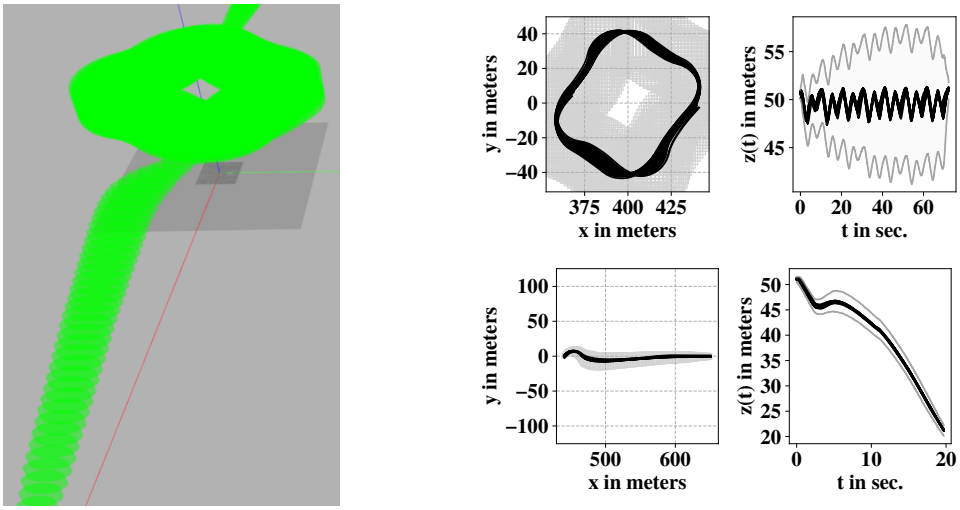


Figure 5.8: Reachtube by DryVR in 3D (*Left*) for ROSplane to loiter and then descend. The traces and reachtube for loiter (*Top Row*) and descent (*Bottom Row*) are projected to xy-plane (1^{st} *column*) and z-axis over time (2^{nd} *column*).

apply projections to x, y, and z axes and transform to the coordinates used by the Airspace Manager. We simulate some of its traces and then divide them into segments to analyze several path primitives denoted as modes for ROSplane [44]. In Figure 5.8, we show the reachtubes for two modes, namely loiter and descend. Unsurprisingly, the plane may not maintain the desired altitude (z-axis) precisely while loitering, and thus it is important to reserve enough range of altitude in OVs for ROSplane.

In summary, we are able to derive useful, i.e., not overly conservative, OVs using DryVR, even with noisy simulations, as shown in Figure 5.8. The main engineering difficulty we

faced using DryVR is to divide traces into proper segments that are from the same mode for ROSplane. This requires domain knowledge on each air vehicle model, for which we refer the readers to [42] and [44].

5.6 SKYTRAKX IMPLEMENTATION AND EVALUATION

Our experiments are conducted using SkyTrakx. SkyTrakx and all simulation scripts are available at our GitHub repository.[12] We present experiments with the Hector Quadrotor model [42] with its default waypoint-following controller. We first describe SkyTrakx, then the scenarios, then the experimental results followed by a brief discussion.

5.6.1 SkyTrakx: System Details

SkyTrakx consists of four major components:

(1) Dione verification discussed in Section 5.4,

(2) reachability analysis and reachtubes from DryVR described in Section 5.5,

(3) an executable reference UTM protocol implemented in Python of Section 5.4, and

(4) UTM protocol simulation and visualization with CyPhyHouse [55].

Here we focus on the executable UTM protocol and its simulation.

To faithfully follow the semantics of our example UTM protocol, we first provide a data structure to represent and easily manipulate rectangular OVs. We provide APIs for designing executable (timed) input/output automata that can interact with simulated vehicles in CyPhyHouse, and implement an execution engine to simulate the input/output automata alongside CyPhyHouse. To reuse reachtube from DryVR, we also design APIs to load pre-computed reachtubes for estimating OVs. Finally, we also provide several scripts to set up desired scenarios and environments in CyPhyHouse, and implement a plugin to better visualize OVs in the Gazebo simulation backend of CyPhyHouse.

5.6.2 Evaluation Scenarios

Following the protocol defined in Section 5.4, a *scenario* for evaluation is specified by

(1) the set of agents `ID` which we consider $N = |\texttt{ID}|$

[12] `https://github.com/cyphyhouse/CyPhyHouseExperiments`

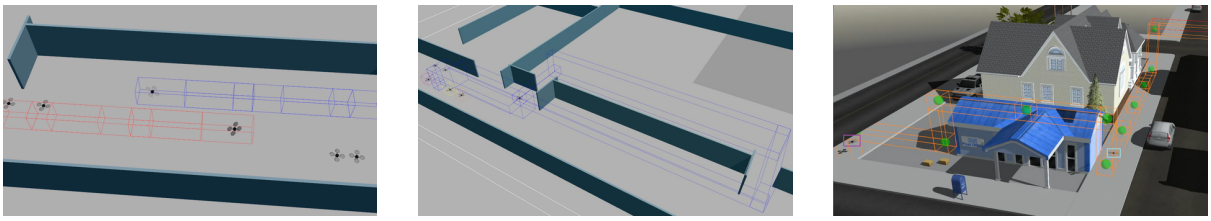


Figure 5.9: Maps: CORRIDOR (*Left*), LOOP (*Mid*), and CITYSIM (*Right*)

(2) the world map and the sequence of waypoints for each agent denoted as the *map*, and

(3) the strategy that the agents use to generate OVs from their waypoints.

For example, the *Left* figure in Figure 5.9 shows a scenario with $N = 6$ drones in the CORRIDOR map. It uses the AGGRESSIVE strategy to generate OVs, which are visualized in the red and blue frames.

We evaluate our protocol in the following maps shown in Figure 5.9:

(1) CORRIDOR simulates two sets of drones on the opposite sides of a tight air corridor trying to pass through. This may happen in a garage-like space where a fleet of air vehicles enter or leave.

(2) LOOP simulates each drone following the vertices of the same closed polygonal chain. This models common segments in the routes of air vehicles such as pickup packages or return to bases' routes.

(3) CITYSIM is a more realistic scenario which simulates drones flying in a city block.

(4) RANDOM(k) are scenarios where each drone follows a sequence of k random waypoints inside a $25m \times 25m$ arena. This is to validate our protocol via random testing.

In addition, a designated landing spot for each drone is specified as the last waypoint in all maps to ensure the liveness property. This avoids the situation where a landed drone blocks other air vehicles.

Conservative and AggressiveOVs We implemented two strategies, namely CONSERVATIVE and AGGRESSIVE, to generate OVs from given waypoints and positions. Both strategies are deterministic and use only *hyper-rectangles* for specifying regions in OVs. As discussed in Section 5.5, CONSERVATIVE reserves large rectangles covering consecutive waypoints with longer durations between time points. Thus, it acquires unnecessarily large volumes and may obstruct other agents. In contrast, AGGRESSIVE heuristically selects smaller rectangles and

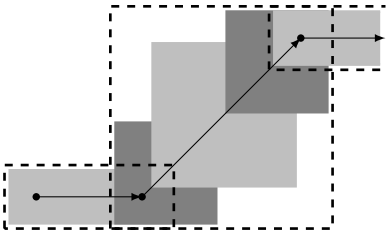


Figure 5.10: Regions for waypoints generated by AGGRESSIVE (*Solid rectangles*) and CONSERVATIVE (*Dashed rectangles*) strategies.

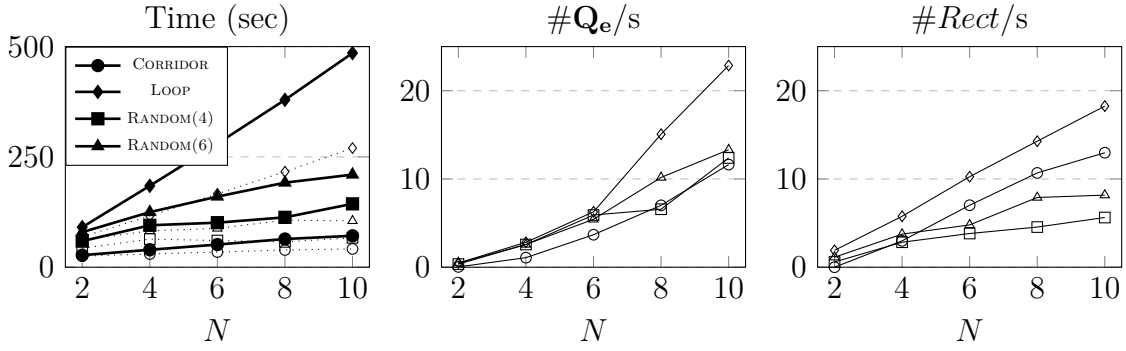


Figure 5.11: Response time per agent (*Left*), #emptiness queries per second (*Mid*), and #rectangles checked by the *AM* per second (*Right*) for each map using CONSERVATIVE strategy. Max is in *Solid marks and lines* and Avg. is in *Hollow marks and dotted lines*

shorter durations. Therefore, AGGRESSIVE is less likely to block other agents but increases the workload of the *AM* because the OVs (numbers of rectangles) are more complex.

5.6.3 Experimental Results

Setup Our simulation experiments were conducted on a machine with 4 CPUs at 3.40GHz, 8 GB memory, and an Nvidia GeForce GTX 1060 3 GB video card. The software platform is Ubuntu 16.04 LTS with ROS Kinetic and Gazebo 9. For the time usage, we report the simulation time from Gazebo (time elapsed in the simulated world) instead of the wall clock time. This is to reduce the time variations due to irrelevant workload on our machine. To address the nondeterminism arising from concurrency in simulating multiple agents, we simulate each scenario three times, and report the average value of each metric.

Response Time and Workload Figure 5.11 shows the response time for each drone starting from sending the first request to finish traversing all waypoints using the CONSERVATIVE strategy in the CORRIDOR, LOOP, and RANDOM(k) maps. As expected, the maximum response time per agent grows linearly against the number of participating agents

Table 5.1: Comparison of simulation time between CONSERVATIVE and AGGRESSIVE. N is the number of agents, Time(s) is the total time according to *the simulated clock* in seconds, $\#Rect/s$ is the number of rectangles per second in the disjointness query of OVs by the AM.

		CONSERVATIVE		AGGRESSIVE			Increased
Map	N	Time(s)	$\#Rect/s$	Time(s)	$\#Rect/s$	Speedup	$\#Rect/s$
CORRIDOR	2	27.52	0.00	21.30	0.00	1.29X	N/A
	4	39.78	2.99	27.24	6.16	1.46X	2.71X
	6	51.63	7.02	34.14	14.10	1.51X	2.06X
	8	64.18	10.68	37.91	22.13	1.69X	2.01X
	10	95.47	12.97	41.94	35.14	2.28X	2.07X
LOOP	2	91.05	1.91	37.63	6.85	2.42X	3.59X
	4	184.88	5.77	70.89	23.33	2.61X	4.04X
	6	280.51	10.26	103.28	40.52	2.72X	3.95X
	8	379.53	14.28	134.62	63.71	2.82X	4.46X
	10	485.58	18.26	169.25	90.94	2.87X	4.98X
CITYSIM	2	77.42	1.77	49.92	4.48	1.55X	2.53X

because, in the worst case, all agents are accessing the shared narrow air-corridor, and the last agent has to wait until all other agents finish. The average response time shows that it is possible to finish faster if agents can execute concurrently in disjoint airspaces. For example, the average time for 10 agents is smaller than the time for 8 agents in RANDOM(6).

In Figure 5.11, we consider the number of emptiness/disjointness queries (denoted as $\#\mathbf{Q_e}$) and of hyper-rectangles to check (denoted as $\#Rect$) per second for the AM. $\#Rect$ provides a finer estimation of computation resources needed by the AM than $\#\mathbf{Q_e}$. The growth of $\#\mathbf{Q_e}$ as expected is roughly quadratic against N in the worst scenario due to checking pairwise disjointness. However, the growth of $\#Rect$ is not as fast and is seemingly linear to N in the worst scenario. Therefore, it is very likely that the workload increases only linearly instead of quadratically when we use hyper-rectangles for OVs.

Conservative vs. Aggressive We compare the time between the CONSERVATIVE and AGGRESSIVE strategies in the CORRIDOR, LOOP, and CITYSIM maps. Due to the heavier demand for computational resources required, we only simulated two drones in CITYSIM. Table 5.1 shows that the AGGRESSIVE strategy can reduce the overall response time and provides a 1.3-2.8X speedup with larger number of participating agents. This experiment shows that our framework is suitable for comparing and quantifying the trade-offs between performance, safety, and workload under different strategies for generating OVs.

5.7 DISCUSSIONS AND FUTURE DIRECTIONS

There is a strong need for a toolkit for formal safety analysis and larger scale empirical evaluations of different UTM protocols. We present SKYTRAKX, a toolkit with an executable

formal model of UTM operations and study its safety, scalability, and performance.

Our toolkit SkyTrakx offers an open and flexible reference implementation of a UTM coordination protocol using ROS and Gazebo. Our formal analyses in SkyTrakx illustrate how formal reasoning can be applied to the family of UTM protocols. We discovered the capability but also the lack of features of Dione [103] and Dafny [104] for providing automated proofs, and to our knowledge, there is no other proof assistant for IOA that also supports the modeling of OVs. We further studied the connection between OVs and reachability analysis, and we showcased how to use DryVR to over-approximate the reachable regions of airspace using simulation traces. The simulator also makes it possible to study different strategies for reserving OVs.

Some simplifying assumptions made can be removed with careful engineering, while others require brand-new ideas. Handling timing and positioning inaccuracies and heterogeneous vehicles fall in the former category. We have partly addressed this category using existing reachability analyses in Section 5.5. In the latter category, a major concern is when there are unavoidable violations of OVs due to, for example, hardware failures. Possible solutions include integration with existing predictive failure detection or failure mitigation strategies and collision avoidance protocols, incorporation of human operators, or generation of notifications to other participating agents for collision avoidance. Finally, an important extension is the design of a coordination protocol for multiple airspace managers having the same guarantees.

Chapter 6: Abstractions for Smart Manufacturing Systems

In this chapter, we consider distributed systems with nonidentical agents commonly seen in smart manufacturing systems. It is based on our conference paper published in the IEEE 18th International Conference on Automation Science and Engineering (CASE 2022) [127]. A smart manufacturing system is a complex cyber-physical system consisting of a collection of component machines and a floorplan layout defining the spatial relationship between components. Each component may be of different physical behavior with different control software. Simulation and testing on smart manufacturing systems require a software infrastructure that can orchestrate the execution of heterogeneous, cyber-physical components besides modeling physical machines in respect to floorplan layouts. Automated simulation as a result is challenging and error-prone. Recent strides in formal modeling of cyber-physical systems and programming languages offer some new techniques for addressing this challenge. In this chapter, we present a compositional automata-based modeling formalism and programming abstractions to design coordination logic between heterogeneous robots in different layouts. Our formalism allows us to automatically simulate and compare performance metrics for different floorplan layouts. We implement our proof-of-concept prototype with the challenging simulation environment for 2021 Agile Robotics for Industrial Automation Competition [45]. Our experiment results demonstrate how our simulation can be used to evaluate and compare performance under different layouts and applicable for reconfiguration and virtual commissioning.

6.1 INTRODUCTION

Modeling and simulation are essential for rapid testing and debugging of smart manufacturing systems. For example, in *virtual commissioning*, many machine configurations, layouts, and variations of associated control programs have to be evaluated, before the actual system is commissioned. The state of the art and the outstanding challenges surveyed in [128, 129, 130] suggest that the high level of expertise and effort required for creating such simulation models make virtual commissioning prohibitively expensive, especially for small and medium-sized enterprises. The challenges arise from two distinct sources. First, developing digital models of physical machines and mechanisms is challenging and requires domain expertise. And second, even with available component models, the software infrastructure that can orchestrate the execution of heterogeneous, cyber-physical components in a manageable simulation is challenging and error-prone. Recent advances in formal mod-

eling of cyber-physical systems [20, 83, 84] and programming languages, offer some new techniques for addressing this second challenge. These techniques not only provide a sound mathematical basis for developing complex models, but they also offer software for creating executable simulation programs [32, 55]. In this chapter, we propose an approach that builds-up on these recent advances to show how the burden of developing simulation models can be reduced with abstractions and compositional modeling.

Broadly, a simulation model for a manufacturing system consists of a collection of component machines (we call them generically as *robots* in this chapter) and a *layout* that defines the spatial relationship between the robots. Robots may be of different types, they may have different control software, and they interact physically (e.g., through transfer of physical materials) and logically (e.g., through transfer of data as needed by a coordination software layer on top of control software). A simulation framework for a manufacturing system therefore requires a good abstraction for

(1) modeling physical interactions such as reading sensor and writing actuator ports on the robot,

(2) programming coordination software layer such as accessing shared variables between control software

(3) composing all robots, control software, and coordination software layer while considering the layout.

In recent years, thanks to the effort from government and the open source community, the Agile Robotics for Industrial Automation Competition (ARIAC) [45], hosted by the National Institute of Standards and Technology (NIST), provides freely available simulated industrial robots such as the robot arms from Universal Robots and rail-guided vehicles in Figure 6.1. These high-fidelity simulation frameworks enable exploring different combinations of available robot models to improve smart manufacturing systems. For example, using fewer robots can help reduce cost and lower redundancy in general. With more robots, there will be higher cost, but the performance is not guaranteed to improve and depends on good coordination of control software. Testing and evaluating the simulation can thus help to answer if a new layout with more robots can achieve higher throughput or only creates redundancy. However, running the simulation and evaluating the performance remains error-prone because the current simulation frameworks have not addressed the complexity of coordination between the control software and the composition of heterogeneous robots under different layouts.

Figure 6.1: GEAR simulation environment for the ARIAC 2021 competition.

In this chapter, we demonstrate the feasibility of automated simulation of different layouts of robot placements to evaluate performance, throughput, and cost-effectiveness. Given a valid layout, we showcase how to automatically generate and execute the parameterized controller to obtain analytic from simulations. Our framework therefore allows users to explore the improvement and trade-offs in the design of smart manufacturing systems.

In summary, the contributions in this chapter are as follows:

(1) We present a compositional automata-based modeling formalism to program coordination logic between heterogeneous robots in different layouts.

(2) We propose an analysis and simulation framework for our modeling formalism to automatically measure and compare performance metrics for different layouts.

(3) We implement and demonstrate our proof-of-concept prototype with the high-fidelity simulation environment in ARIAC 2021.

Looking ahead, we believe that the type of compositional modeling illustrated in this chapter can not only lower the barrier to creating simulation models for smart manufacturing systems, but it can also help with developing and tuning controller programs, performance evaluation [131], and anomaly detection.

Related works Research on software defined control (SDC) framework [132, 133, 134] has proposed approaches on the management and monitoring for smart manufacturing systems and provides a global view of the system for decision-making. All the works benefit

from integrating fixed simulation models or digital twins for predictive decision-making and anomaly detection. Our simulation approach with parameterized layouts can further extend the existing simulation for more advanced application such as virtual commissioning and reconfiguration.

A number of ideas from the extensive body of research on modeling for cyber-physical systems inform our modeling formalism (see [16, 20, 83, 84] and the citations therein). A previous mathematical formalism (without realistic simulations) for smart manufacturing appeared in [135]. The formalism presented in this chapter integrates the notion of shared memory and controller ports from our KOORD [32] programming language.

6.2 AN EXAMPLE: AGILE ROBOTICS FOR INDUSTRIAL AUTOMATION

We illustrate our motivating example following the Agile Robotics for Industrial Automation Competition (ARIAC) [45] hosted by the National Institute of Standards and Technology (NIST). In particular, we study ARIAC 2021: a smart manufacturing scenario revolving around the theme of the COVID-19 pandemic along with the Gazebo Environment for Agile Robotics (GEAR).

GEAR for ARIAC 2021 In ARIAC 2021, the goal is to design the software for a smart manufacturing system that will assemble on-demand, ventilator briefcases consisting of four items: a battery, a sensor, a regulator, and a pump. The competing control software is required to transport the correct items from the conveyor belt to one of the assembly stations (`as1` to `as4`) and assemble the briefcase as shown in Figure 6.1. GEAR consists of four types of robot platforms: a conveyor belt, a kitting robot, Automated Guided Vehicles (AGVs), and a gantry robot. The conveyor belt serves as an entry point for new items and moves items from left to right. The kitting robot consists of a UR10 robot arm, a vacuum gripper to grab items, and a linear rail to move parallel to the conveyor. The AGVs carry items along a defined path with fixed destinations and deliver items to assembly stations. Lastly, the gantry robot consists of another UR10 robot arm that is attached to a rotatable torso, which can move along a two-dimensional plane on two linear rails.

Simulations for Layout Optimization A crucial application of the simulation environment is to evaluate over different layouts of robots to compare various metrics such as throughput, safety, robustness, etc. This allows finding the optimized layout. However, automated simulation for different layouts has to address two concerns:

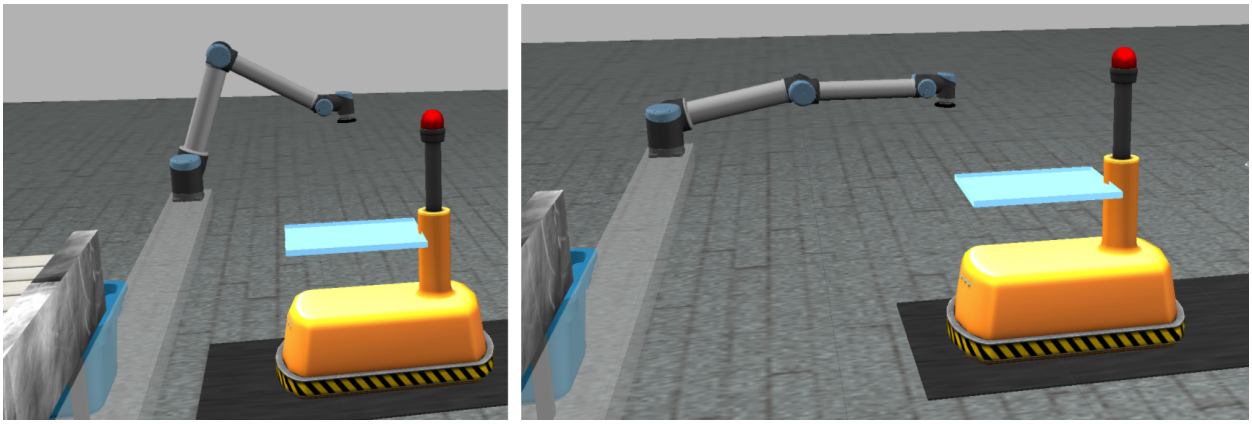


Figure 6.2: Feasible vs infeasible layouts constrained by the operating range of the robot arms. The robot arm is able to reach the AGV in a feasible layout (Left). The robot arm is too far from the AGV in an infeasible layout (Right).

(1) the controllers for robots will require adjustments under different layouts, and

(2) the physical parameters, such as the operating range of the robot arms, limit the relative placements of the AGVs.

To address the first concern, our approach of parameterized automata with shared variables generalizes the controllers to work under different layouts. More specifically, we are able to support all four main robot types given in ARIAC 2021 the conveyor belt, AGVs, the kitting robot, and the gantry robot. An automaton is constructed for each of the existing robots so that it can accordingly carry out its own tasks, and the composition of all automata is used to simulate the end to end goal of delivering items between the robots and assembling the ventilator.

To solve the second concern, we consider the information provided in the layout. For the different layouts shown in Figure 6.2, the problematic layout on the right does not consider the operating range of the robot arm, and hence the robot arm cannot transfer items to the AGV. We consider this as an infeasible layout with a *connectivity* issue. Our approach takes in different layouts as inputs and generates a *connectivity graph* to ensure the connectivity of robots, such as the one shown in Figure 6.3. If there exists a connected path from the source to the destination in our connectivity graph, items can be transferred from the conveyor belt to the assembly stations. In other words, we can detect the connectivity issues by checking if a path exists from the source to the destination, which then can be used to exclude infeasible layouts, since the robots will not be able to work together to transfer the items. We will formally define the connectivity graph and check connectivity using standard graph search algorithms in Section 6.3.1.

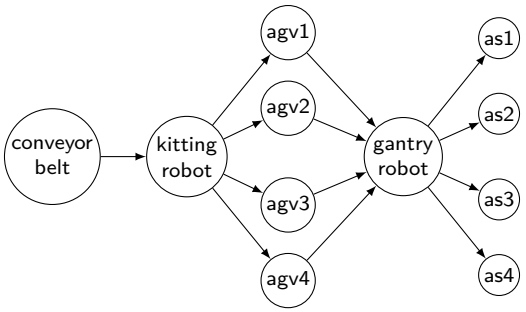


Figure 6.3: The connectivity graph generated for the default world in Figure 6.1.

6.3 A COMPONENT-BASED MODEL OF SMART MANUFACTURING SYSTEMS

In this section, we will discuss our component-based modeling framework for smart manufacturing systems. We will show how each instance of a robotic platform is modeled as a special type of *parameterized automata*. Each type of automata has different sets of *shared variables* for communication with other automata and *ports* to interact with the physical environment. We then discuss the definition of the complete system as a composition of various automata instances defined by a *layout*. This compositional definition enables us to automatically check the compatibility of a layout and generate executable simulation code.

Robot Components The overall system is built by instantiating a number of robots from a library of possible robot types. Let the set of robot types be $\{R_1, R_2, \ldots, R_N\}$. For example, $R_1 = \mathsf{AGV}$ and $R_2 = \mathsf{Conveyor}$. A system consists of robot platform instances $\{r_1, ..., r_k\}$ of different robot types. If a robot platform instance r_i is of the robot type R_j, we denote it as $\mathit{type}(r_i) = R_j$. The robot instance $\mathsf{agv1}$ in Figure 6.1 is of the type AGV. We can write it as $\mathit{type}(\mathsf{agv1}) = \mathsf{AGV}$.

Each type of the robots can be customized or instantiated by fixing a number of *robot parameters*, such as location, orientation, and certain controller parameters. For each robot type R_i, we write the corresponding parameter space as P_{R_i}. For example, $\mathsf{agv1}$ and $\mathsf{agv2}$ in Figure 6.1 are both instances of AGV with different station positions inside the warehouse. Assuming there are three required parameters, the position of the start station, the interval between the stations, and the number of items to transfer in each trip, the parameter space for AGV is $P_{\mathsf{AGV}} = \mathbb{R}^2 \times \mathbb{R} \times \mathbb{N}$. The particular parameter values of $\mathsf{agv1}$ is denoted as $\mathit{param}(\mathsf{agv1}) = ((-2.3, 4.6), 0.73, 4) \in P_{\mathsf{AGV}}$ to represent that the start station is placed at (-2.3, 4.6), the interval between stations is 0.73 meters. Or equivalently, it defines the station positions at (-2.3, 4.6), (-3.03, 4.6), and (-3.76, 4.6), and $\mathsf{agv1}$ transfers 4 items for each trip.

System Composition Our formalization is inspired by the transition system model for software defined controllers in [135] and the shared memory model for distributed robotics system in [32]. We model the entire system as the composition of all robot instances:

$$Sys \stackrel{\text{def}}{=} \mathcal{A}(r_1) \parallel \mathcal{A}(r_2) \parallel ... \parallel \mathcal{A}(r_k) \tag{6.1}$$

where each robot instance is a discrete automaton $\mathcal{A}(r) = (X, Q, \Theta, \Sigma_r, \delta)$ where

(1) $X = X^L \dot{\cup} X^G \dot{\cup} X^P$ is a finite set of *state variable* names. X^L, X^G, and X^P denotes the sets of *local* variables, *shared* variables, and *controller port* names respectively. We assume the function $type(x)$ to return the set of possible values for x. A *state* q is a mapping from $x \in X$ to a value $q(x) \in type(x)$ and $val(X)$ denotes the set of all possible states of X.

(2) $Q \subseteq val(X)$ is the set of states.

(3) $\Theta \subseteq Q$ is the set of initial states.

(4) Σ_r is the set of all actions. The actions are parametrized by the robot parameters $param(r)$.

(5) $\delta \subseteq Q \times \Sigma_r \times Q$ is the transition relation.

Figure 6.4 demonstrates how the local variable (variable s to represent the current status), shared variables (all_loaded and all_dropped), controller ports (pos), and robot parameters are used to define the (partial) automaton for agv1. The system starts from LOADING status and waits for the product items being loaded on the tray. It waits for the shared variable all_loaded to become true before entering MOVING status. This shared variable may be controlled by other robot instances, such as the kitting arm robot, to allow loading and delivering multiple items at a time. Notice that the "goto" action is parametrized by the position of the station at (-3.03,4.6). We can generalize the action to support the other station at (-3.76, 4.6). Once entering the MOVING status, it may take multiple cycles to reach to the particular station. In this case, the value of controller port pos is decided and changed according to the physical environment instead; therefore, it monitors the controller port variable pos until its position reaches the station at (-3.03,4.6) and transits to the DROPPING status. Similarly, all_dropped shared variable is used to decide whether all items have been dropped to the assembly station so that it can enter RETURNING status to go back to the starting station and reset to LOADING status.

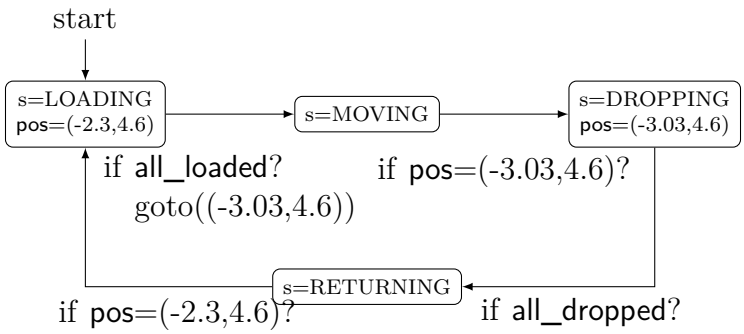


Figure 6.4: Partial automaton for the robot instance agv1 working between stations at (-2.3,4.6) and (-3.03,4.6).

6.3.1 Connectivity Check for Feasible Layouts

```
"robots": [
    {   "name": "kitting",
        "type": "kitting",
        "pose": [-1.3, 0, 1.127],
        "rail_dim": "y",
        "rail_range": [-4.8, 4.8]},
    {   "name": "conveyor_belt",
        "type": "conveyor",
        "pose": [-0.573, 0, 0],
        "orient": "y",
        "orient_range": [-5,5]}
]
```

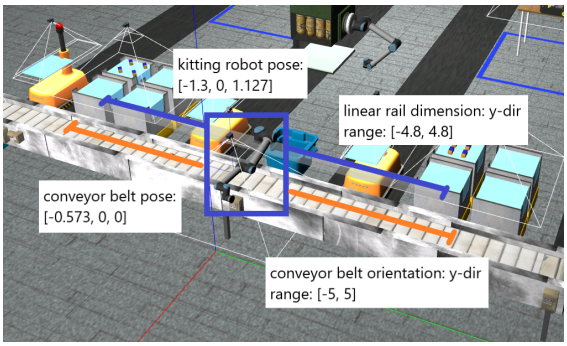


Figure 6.5: A programmable layout in JSON format (*Left*) and the corresponding Gazebo simulation environment (*Right*).

For simplicity, we consider the robot instances as places in the two-dimensional space $\mathbb{R}^2$. We begin by computing the operating ranges of all the robots specified in a layout. Figure 6.5 demonstrates how the position, orientation, and range parameters in the environment are specified in JSON format, and the annotated operating ranges in the Gazebo simulation environment. The type of robot determines the operating range of motion. For instance, the kitting robot's range of motion is a cylinder with its axis along the x or y-direction. An AGV's operating range can be represented by a set of 2D vertical lines, since other robots will be dropping items and grabbing them off of the AGV. Since computing the geometric space of the operating range is standard, we skip the detail of the computation and use $range(r_i)$ to denote the operating range derived from a given layout.

Formally, a connectivity graph is a directed graph $G = \langle V, E, s, T \rangle$ where V is the set of all robot instances $\{r_1, r_2, ..., r_k\}$, an edge $(r_i, r_j) \in E$ represents that an item can be directly

transferred from r_i to r_j, $s \in V$ is the source node, and $T \subset V$ is the set of destination nodes. The process of generating a connectivity graph is as follows. Starting from a graph with all robot instances $\{r_1, r_2, ..., r_k\}$, but without any edges, we mark the robot instance where the items enter the system as the source node s and the robot instances where the final product item leaves the system as the destination nodes $t \in T$. For the connectivity graph in Figure 6.3 as an example, the conveyor belt robot is the source node $s = \mathsf{conveyor}$, and the assembly stations are the destination nodes $T = \{\mathsf{as1}, \mathsf{as2}, \mathsf{as3}, \mathsf{as4}\}$. We add an edge from node r_i to node r_j if and only if

(1) the operating ranges of robot instances intersect, i.e., $range(r_i) \cap range(r_j) \neq \emptyset$ and

(2) robot r_i would transfer an item over to robot r_j according to the robot types $type(r_i)$ and $type(r_j)$.

Finally, we can check if every node is reachable starting from the source node s, and every node can reach any destination node $t \in T$. A standard algorithm is to use depth-first search (DFS) starting from the source node s. If there is no path from the source node s to a node r_i, then there is no path for an item to be delivered to r_i, hence the robot instance r_i is not usable. Similarly, if there is no path from r_i to any destination node t, the item will never be used in the final product. We can therefore detect and reject these invalid layouts by constructing the connectivity graph.

6.4 IMPLEMENTATION

In this section, we present our method of implementation for building our system. The three major components of this include (1) the input layout in the form of a JSON file and the generation of a Gazebo world as an SDF file, (2) controller automata, and (3) a connectivity check. All of our implementation is in an open source repository at [136].

Programmable Layouts to Gazebo Environments We parse the input JSON file and retrieve all dynamic models (robots) and static models (assembly stations, bins, other obstacles). We then write this information in world files that are applied to SDF files when the Gazebo world is launched. The resulting world should show the layout specified in the JSON file as a Gazebo simulation.

Building Robot Controllers As discussed in Section 6.3, each robot follows an automaton depending on the robot type with tunable robot parameters. To implement the

automaton, the basis of all robotic movement in our simulation is done through ROS, specifically through messages passed over ROS topics and ROS service calls. For simpler robots, such as the conveyor belt and the AGVs, a simple command to control the speed is enough. For more complex robot arms such as the kitting and gantry robot, we use the MoveIt framework for sending ROS messages to control the arm joints. On top of MoveIt, we use an analytical-based inverse kinematics approach for motion planning to generate a sequence of desired arm joint values and brings the vacuum gripper to a desired position.

The MoveIt framework provide a well established motion planning tool that allowed us to abstract over the lower level details of sending ROS messages. This helped us build an analytical approach-based inverse kinematics solver for the two robot arms without going into an extreme level of detail. There are also plenty of alternatives including a sampling-based motion planner, such as a Probabilistic Roadmap (PRM) or a Rapidly-Exploring Random Tree (RRT).

Connectivity Check We use a short Python function to implement our connectivity check. In short, we create a node for each robot specified in the JSON file and then create a directed edge between nodes where robots can transfer items to one another. We implement a standard depth-first search algorithm and run it on the starting node to determine whether connectivity from start to finish is satisfied or not.

6.5 EXPERIMENT RESULTS

We use the competition interface provided by the ARIAC for running our experiment. Items will be spawned on the conveyor belt and shipped to briefcases located at assembly stations. We run the simulations on three different layouts, namely the default competition layout in Figure 6.1 (Default), a more spaced layout (Spaced), and a layout with less AGVs (LessAGV):

- Default layout has one conveyor belt, one kitting robot, four AGVs, and one gantry robot. The default space between the conveyor belt and the kitting robot is 0.73 meter. The starting poses of the four AGVs (in order from 1 to 4) are (-2.3, 4.6), (-2.3, 1.3), (-2.3, -1.3), and (-2.3, -4.6).

- Spaced layout sees increased space between the conveyor belt and kitting robot to 0.93, as well as increased space between the four AGVs. The starting poses of the AGVs are now (-2.5, 4.2), (-2.5, 0.9), (-2.5, -0.9), (-2.5, -4.2). It uses the same number of robots as the Default layout.

- LessAGV world has the same layout with the same spacing as Default world, with the removals of two robots: agv2 and agv4.

We also consider varying the robot parameters for the robot automata. The tunable robot parameter in our experiments is the capacity of an AGV, the maximum number of items loaded for each trip from the conveyor belt to assembly stations, denoted as *Cap*.

Recall from Section 6.2 that an order is to assemble a ventilator briefcase composed of four different kinds of items. For each combination of a layout and a parameter value, we conduct a simulation of filling 8 orders and record the total time usage of a simulation run. We then divide the total time by 8 to calculate the average amount of time for finishing one order as shown in Table 6.1.

All of our experiments are conducted on a Ubuntu workstation (version 18.04.6 LTS) with CPU model Intel Xeon Silver 4110, GPU model NVIDIA Corporation GP104GL, ROS Melodic (Version 1.14.12), and Gazebo 9.16.0.

Table 6.1: Average time to finish one order in seconds under different combinations of simulation of shipping 8 orders.

	$Cap = 2$	$Cap = 4$	$Cap = 8$
Default	196.650	173.037	133.327
Spaced	198.654	197.384	156.379
LessAGV	TIMEOUT	180.993	222.375

Table 6.1 shows that the time usage is generally worse with increased space, but only marginally for certain values of *Cap*. We see significant improvements with loading more items on less AGVs for the default world and the spaced world, suggesting that operations with the gantry robot are the bottleneck and that optimizations regarding the gantry movement and its parameter tuning should be the focus in future runs. In LessAGV layout with less AGVs, we see significantly higher times to ship items but a different parameter value that produces optimal throughput. The results here show the possible improvements by adjusting just one robot parameter.

In addition, we track the trajectories of items in transitions for runtime monitoring. Figure 6.6 shows the position trajectories of two battery items starting from the right end of the convey belt, i.e., lower right of the plot. Both batteries move to the left by the conveyor belt. The curves denote the two items are transferred to agv3 at (-2.3, -1.3) by the kitting_robot, put down slightly separated, and carried to the assembly station as3 in the same trip. Finally, the plot shows the gantry_robot picked up and transfer battery_1 to the assembly table. This showcases that the prototype is able to do fine-grained runtime

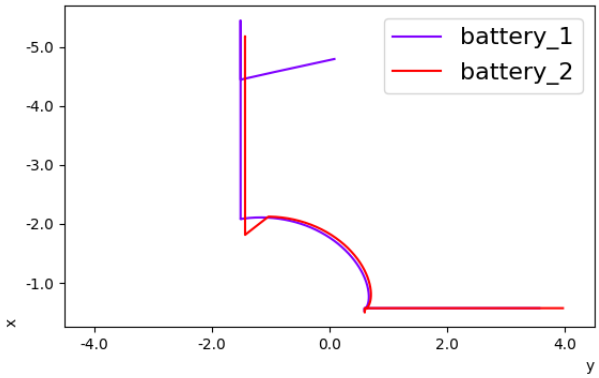


Figure 6.6: 2D position trajectories of two battery items in Default layout. XY-axes are rotated 270° to match Figure 6.1. The trajectories start from the lower right corner. Both batteries are moved to the left by the conveyor belt. put on agv3 separately by the kitting_robot, and carried to the assembly station as3 in one trip. Finally, battery_1 is transferred to the assembly table by the gantry_robot.

tracking of individual components. In short, our experiment shows that our framework can provide system-level metrics such as throughput, as well as component-level details such as the trajectories of items, and we can repeat the simulation and analysis for different layouts and parameters to identify critical components and subsequently optimize the entire smart manufacturing system.

6.6 DISCUSSIONS AND FUTURE DIRECTIONS

We presented our approach to a compositional modeling and simulation of smart manufacturing systems. Given an input JSON file representing the layout of a manufacturing floorplan, and the parameterized controllers for the robot types, our system can generate a detailed simulator for the entire plant. The solution uses the notion of controller ports and shared variables for physical and logical interaction among robot components. We showed how the generated simulator can be used for measuring and comparing throughput of the ventilator assembly orders in ARIAC 2021 with four distinct robot types. Our current implementation relies on the Gazebo simulator and the ARIAC 2021 scenario; however, the concepts can be implemented on other simulators and scenarios.

Beyond the preliminary experimental results we have discussed, the work suggests several directions for future research. First, the parameterized controller programs used for simulation can be compiled to executable code and deployed on the actual hardware. This can significantly reduce the development and testing cost. A variant of this idea for the KOORD language has been demonstrated in the context of mobile robotic applications [55]. Gener-

ating programmable logic controller (PLC) code directly from communicating state machine models would be an interesting direction to explore. Second, the compositional models, with the well-defined interfaces (ports and shared variables) should be amenable to automatic generation of runtime monitors [137, 138] for software defined control [132, 133, 134]. Finally, rapid generation of simulators and their evaluations open-up the possibility of optimizing designs and auto-tuning parameters for manufacturing systems.

Chapter 7: Approximate Abstraction for Vision-Based Perception

In this chapter, we study the safety of autonomous systems with vision-based perception components. This is based on our conference paper [30] in the International Conference on Embedded Software (EMSOFT) in 2022.[13] We aim to provide safety guarantees on the complex vision-based perception components integrated in safety-critical control systems. Unfortunately, fully formal verification of these complex components is likely to remain challenging in the foreseeable future. Take the lane tracking control system (LTC) for autonomous vehicles in Figure 7.1 as an example, the perception component at least includes a camera sensor, *cam*, to take images and a vision or DNN algorithm, *nn*, to identify roads and lanes in the images. This perception component reveals two key challenges on formally verifying the autonomous system with vision:

- *No Specification*: It is difficult if not impossible to mathematically specify the set of camera images containing a lane, and
- *Scalability*: Existing NN verification tools for image processing DNNs currently only verifies local robustness against perturbations on a single camera image [81, 82]. Global robustness for the entire image domain is still beyond the capabilities of existing NN verification techniques.

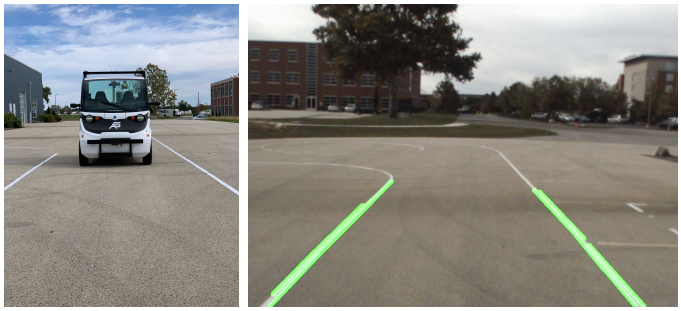


Figure 7.1: Vision-based lane tracking control system (LTC) on an autonomous vehicle.

We therefore develop our approach following a pragmatic path: We approximate the camera and DNN components together as the perception component $nn \circ cam$, and we systematically search for an *Approximate Abstract Perception* (AAP), a safe approximate model M of $nn \circ cam$ which still enjoys verifiability, but the abstraction relationship between M and $nn \circ cam$ can have an error, and we can quantify and estimate this error arbitrarily precisely with high probability via sampling input and output of the component $nn \circ cam$.

[13] This is a joint work with Yangge Li, Dawei Sun, Keyur Joshi, and Prof. Sasa Misailovic.

This approach helps us bypass the two challenges of the lack of specification and the scalability issue. Firstly, the combined component $nn \circ cam$ is to perform a state estimation task, for instance, the perception task in the LTC system is to estimate the relative position of the vehicle to the lane. We can naturally specify the state estimation output with respect to the ground truth position, and avoid specifying the semantics of a lane over image pixels, Secondly, the input and output domains have much fewer dimensions compared with the high dimensional image domain.

Another benefit of our approach is that by choosing an appropriate structure of M, we can make it *intelligible*. That is, M not only proves safety but helps explicate *why* the overall system is safe and where it deviates from the actual perception $nn \circ cam$. The importance of such explanations have been argued in [139, 140]. We propose a general framework to allow a customized construction of M. Users can design the proper description and visualization of M, and therefore they can understand and investigate the error of M with respect to $nn \circ cam$. These three axes—safety, intelligibility, and precision—define a space for exploring different safety assurance methodologies for autonomous systems. Our main claim is that this is one of the first[14] approaches to provide safety assurance for realistic vision-based control systems with abstractions, approximate or otherwise.

In Section 7.1, we first give an overview of the technical contributions presented in this chapter. We in particular connect the idea of AAPs with the *abstractions of the environments* formalized in Chapter 2, and we provide an outline of the remain sections in this chapter.

7.1 OVERVIEW

We first discuss our CPReact system model for autonomous systems with vision-based perception components and our design choices to search for an abstraction of for the perception component. We consider the actual perception component $nn \circ cam$ as a part of the real environment $REnv$, and $nn \circ cam$ is to provide the values of the percept variables P to the reactive module RM. Following our verification approach in Section 3.2, we aim to synthesize a *safe abstraction* of the real environment including $nn \circ cam$. To solve this synthesis problem, we draw inspirations from robust control literature and use a *nominal model*, an assumed model with the ideal behavior. The reactive component RM such as the software controller would be designed with respect to the nominal model, and RM interacting with the nominal model should ensure the safety requirement $[\![Safe]\!]$ through preserving some desired invariant property $[\![Inv]\!]$. The nominal model however is not necessarily an abstraction

[14]The only other closely related works are [28, 29, 59].

of the real environment *REnv* because it may not simulate *REnv*. Our main insight is to find an abstraction built on top of the nominal model combined with *uncertainties*, additional parameters capturing differences between the real environment and the nominal model.

In the context of vision-based perception, we assume there exists a perfect perception function h as the nominal model that computes the *ground-truth* percept values. Suppose the ground truth percept value is $h(x)$ at a given state x. The actual vision pipeline $nn \circ cam$ estimates $h(x)$ using images, which depends on the state x and also environmental parameters *env* such as lighting, weather conditions. These environmental parameters *env* can add bias and variance in the estimation. As a result, the real environment *REnv* should have the variable x and additionally environmental parameters *env* as the latent variables, and we search for an abstraction *Env* that has the latent state variable x, and an AAP M used in *Env* should account for the variations in the estimation caused by *env*.

We use a piecewise affine template to search for an AAP as a *set-valued function* M, where the center (mean or bias) of the set is a piecewise affine function $\mathrm{A}_i \times h(x) + \mathrm{b}_i$ of the ground truth $h(x)$. We can infer the linear model using regression on the estimate from running the vision pipeline $nn \circ cam$ on images and their ground truth labels. While the center (mean) of the set $M(x)$ is defined by training data, the size and shape of the set (variance) is inferred from the invariant property. Assume that the control system with perfect perception is safe with respect to a given invariant set $[\![Inv]\!]$ to ensure the safety $[\![Safe]\!]$. Using program analysis tools like CBMC [141] and IKOS [142] on the code for the controller, we infer the set of *unsafe* perception outputs for any x. Then, the set-valued output from $M(x)$ is determined to be the *largest set*, centered at $\mathrm{A}_i \times h(x) + \mathrm{b}_i$, that keeps the system safe. The computation of this largest set is an optimization problem.

The constructed AAP M is a piecewise affine set-valued function of the actual variable that the perception system $nn \circ cam$ is trying to estimate. By construction, M is verified safe relative to the given invariant $[\![Inv]\!]$ and therefore the safety $[\![Safe]\!]$. We also double-check this using CBMC by plugging-in M into the downstream modules of the control system. We apply the method to two realistic end-to-end autonomous systems using Gazebo for rendering images and detailed vehicle control models: a vision-based lane tracking controller for an electric vehicle and a vision-based corn row scouting robot.

We empirically evaluate the *precision* of the constructed AAPs, i.e., the error between M and $nn \circ cam$, across large variations in the environment such as roads with varying numbers of lanes, lighting conditions, different types of crops and fields. On the positive side, for parts of state space, with probability at least 0.9, we observe the precision of M approximating $nn \circ cam$ is over 90%. This type of analysis tells us that the end-to-end system is safe for these parts of state space, because the verifiable-by-construction M is likely to be an exact

abstraction of $nn \circ cam$ in these parts. Somewhat counter-intuitively, we observe that error between M and $nn \circ cam$ can be high in some very safe states (e.g., vehicle at the center of the lane and aligned with the lane). While these states happen to be the ones where safety assurance is less important in practice we discuss why this makes sense and how it can be addressed with better verification techniques or multi-resolution approximations.

In summary, our contributions are: (1) A formalization of verifiable approximate models for vision-based perception used in autonomous systems. (2) An approach to compute piecewise affine set-valued approximations. (3) A demonstration of how to compose constructed approximations with the downstream modules for the end-to-end verification using existing techniques. (4) Careful empirical evaluations on the precision of the verified approximations for two significant case studies. Safety, intelligibility, and precision appear to be a useful dimension for thinking about AAPs. The constructed approximate abstractions are useful for verification, identifying where perception fails, which can in turn help design better perception, and help define system-level *operational design domains (ODDs)* [143].

7.2 SYSTEM DESCRIPTION

The problem of assuring safety of an autonomous system can be stated as follows: given an autonomous system *Sys*, we would like to check that it satisfies a safety requirement $[\![Safe]\!]$. For example, for a lane tracking control system (LTC) for a vehicle in Figure 7.2, the safety requirement is that the vehicle always remains within the lanes. This textbook statement of the problem is complicated by two factors in an actual autonomous system. Firstly, *Sys* uses vision for perception–converting pixels to *percepts* such as deviation from lane-center, and such perception systems are not amenable to formal specification and verification. Secondly, the output of the perception pipeline depends on environmental factors $\mathcal{E}$ such as lighting, texture, and pavement moisture. These dependencies are neither well-understood nor controllable.

We model the complete control system as a discrete time transition system[15] *Sys* with four components transforming different types of data (Figure 7.2). The *dynamics* defines the evolution of the system state x as a function of the previous state and the output from the *control*. We model the *dynamics* as a function $f : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$. In our example, the state x of the vehicle includes its position, orientation, velocity, etc., and the dynamics function defines how the state changes with a given control action $u \in \mathcal{U}$. In this chapter, we consider

[15]We use the discrete time transition system instead of the CPReact model to significantly simplify the notations. We will discuss equivalent CPReact models in Section 7.3.

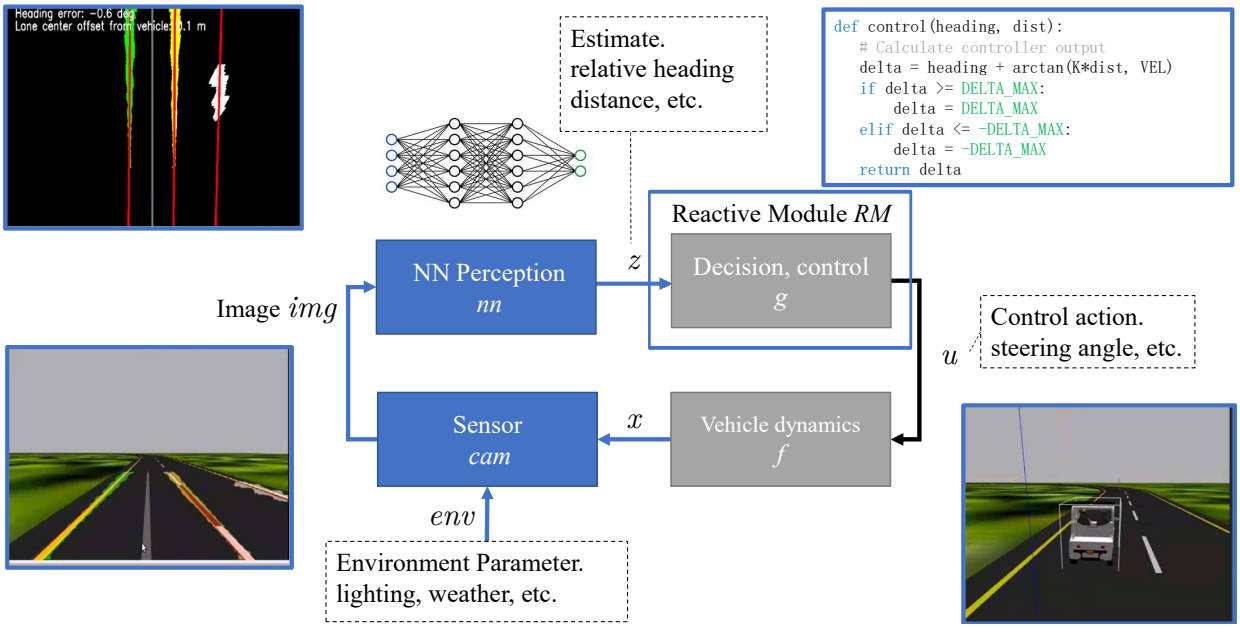


Figure 7.2: Closed-loop model of LTC *Sys* with camera and DNN-based perception.

discrete time models, and write the state at time $t+1$ as

$$x_{t+1} = f(x_t, u_t), \tag{7.1}$$

where x_t and u_t are the state and the control actions at time t. This state transition function could be generalized to a relation to accommodate uncertainty, without significantly affecting our framework or the results.

Then the *control* module takes a percept z as an input and produces a control action $u = g(z)$ as the output. In LTC, the control action u can be a vector of throttle, steering, and brake signals. The implementation of the controller *control* may involve a number of modules including navigation, planning, and optimization. Abstractly, *control* is a function $g : \mathcal{Z} \to \mathcal{U}$ mapping the space of precepts to the space of control actions. During the design of the controller module, the developers often use a nominal perception function $h : \mathcal{X} \to \mathcal{Z}$ to produce a *ground-truth* percept vector $z = h(x)$ for simple verification and sanity checks.

On the other hand, the actual vision-based perception pipeline takes an image (or a high-dimensional vector) *img* as an input and produces a low-dimensional percept *estimate* vector $\hat{z} = nn(img)$ as the output. In a lane tracking control system (LTC), $\hat{z}$ is the position of the camera relative to the lanes seen in the image. That is, we model the perception pipeline as a function $nn : \mathcal{P} \to \mathcal{Z}$ mapping the space of images $\mathcal{P}$ to the space of percepts $\mathcal{Z}$.

The final component closing the loop is the *sensor* which defines the image *img* as a

function of the current state x and a set of non-time varying, *environmental parameters* *env*. For LTC, these parameters include, for example, lighting conditions, nature of the road surface, types of markings defining lanes, etc. We model the *sensor* as a function $cam : \mathcal{X} \times \mathcal{E} \to \mathcal{P}$, where $\mathcal{E}$ is the space of environmental parameter values. In a real system, we may not know all the environmental parameters, they may not be time-invariant, and their precise functional influence on the image will also be unknown. Therefore, it does not make sense to prove anything mathematically about *cam*. For the purpose of generating AAPs of $nn \circ cam$, we reasonably assume that we can sample inputs of *cam* according to some distribution over $\mathcal{E}$ and $\mathcal{X}$. Based on the samples, an empirical precision of the AAP can be computed. In Proposition 7.4, we also give a lower bound for the actual precision of the AAP using the empirical precision. In our experiments, we generate synthetic data using a simulator, and the same could also be done with the actual vehicle platform at a higher cost.

7.3 SYSTEM-LEVEL SAFETY ASSURANCE

The behaviors of the overall system are modeled as sequences of states called *executions*. Given an initial state $x_0 \in \mathcal{X}$ and an environmental parameter value $env \in \mathcal{E}$, an execution of the overall system $\alpha(x_0, env)$ is a sequence of states $x_0, x_1, x_2, \ldots$ such that for each index t in the sequence:

$$x_{t+1} = f(x_t, g(nn(cam(x_t, env)))). \tag{7.2}$$

We would like to have methods that can assure that, given a range of environmental parameter values $\mathcal{E}$, a safe set $[\![Safe]\!] \subseteq \mathcal{X}$, and a set of initial conditions $\Theta \subseteq \mathcal{X}$, none of the resulting executions of the system from Θ can leave $[\![Safe]\!]$ under any choice of $\mathcal{E}$. Such a method will be a useful tool in checking safety of autonomous systems. Secondly, it can help search for $\mathcal{E}$ for which the system can (and cannot) be assured to be safe, and therefore, can be used as a scientific basis for specifying the *operating design domain (ODD)* [11] for the control system (and direct expensive field tests, respectively). Since, the functions *cam* and *nn* are partially unknown with unknown dependence on *env* and x, it is unreasonable to look for the above type of methods. Instead, in this chapter, we develop a method for the following weaker problem:

Problem Given a safe set $[\![Safe]\!] \subseteq \mathcal{X}$ and a range for the parameters $\mathcal{E}$, find an approximation $M : \mathcal{X} \to \mathcal{Z}$ of the perception $nn \circ cam : \mathcal{X} \times \mathcal{E} \to \mathcal{Z}$, such that it is:

(a) *Safe*, i.e., M used in the closed-loop system substituting $nn \circ cam$ makes the resulting system provably safe with respect to $[\![Safe]\!]$.

(b) *Intelligible*, i.e., human designers can design and understand the behavior of M.

(c) *Precise*, that is, M and $nn \circ cam$ are close.

M may and indeed will depend on the safe set $[\![Safe]\!]$. For the substitution in (a) to make sense, we make M a set valued function to accommodate variations in $nn \circ cam$ from different environments. Since the actual perception system $nn \circ cam$ and its dependence on the environment $\mathcal{E}$ is incompletely understood, assertions about the precision (c) have to be statistical. We will see later that indeed fine-grained measurement of closeness is possible.

Connections to Synthesis Problem of Safe Abstractions Here we draw connections from the problem of finding a safe approximation M to the synthesis problem of safe abstractions formalized in Section 3.2. We now provide an equivalent model of the system *Sys* in our CPREACT framework, and we shall see that, compared with the discrete transition system model, the CPREACT model is more general but complicates the notations significantly. For the CPREACT model of the system *Sys*, we choose the controller g as the reactive module RM. The dynamics f, camera sensor *cam*, and NN perception *nn* belong to the environment *REnv*. We then define the set of percept variables, $P = \{\mathtt{z}\}$, the set of feedback variable, $FB = \{\mathtt{u}\}$, the set of latent state variables of *REnv*, $LR = L \cup R = \{\mathtt{x}\} \cup \{\mathtt{env}\}$, and no state variables of RM, $S = \emptyset$. We then can define the transition relation $[\![T_{RM}]\!]_{S,P,S',FB}$ of the reactive module with the simple predicate below:

$$T_{RM} \stackrel{\mathtt{def}}{=} (\mathtt{u} = g(\mathtt{z})) \tag{7.3}$$

where we know the controller g can be implemented as an expression in a program. In contrast, it can be extremely difficult to express the transition relation of the real environment $\mathcal{T}_{REnv}$ as a predicate due to the complex sensing *cam* and vision-based perception *nn*. Formally, $\mathcal{T}_{REnv} \subseteq \mathcal{Q}_{L,R,P,FB,L',R',P'}$ is defined as:

$$\mathcal{T}_{REnv} \stackrel{\mathtt{def}}{=} \left\{ (l, r, p, fb, l', r', p') \;\middle|\; \begin{array}{c} p(\mathtt{z}) = nn \circ cam(l(\mathtt{x}), r(\mathtt{env})) \wedge \\ l'(\mathtt{x}) = f(l(\mathtt{x}), fb(\mathtt{u})) \wedge r'(\mathtt{env}) = r(\mathtt{env}) \\ p'(\mathtt{z}) = nn \circ cam(l'(\mathtt{x}), r'(\mathtt{env})) \end{array} \right\} \tag{7.4}$$

where $l, l' \in \mathcal{Q}_L, r, r' \in \mathcal{Q}_R, p, p' \in \mathcal{Q}_P$ and $fb \in \mathcal{Q}_{FB}$. In simple terms, the old percept value $p(\mathtt{z})$ is calculated by applying sensing *cam* and perception *nn* on the old state $l(\mathtt{x})$ and

environmental parameter $r(\texttt{env})$. The old state $l(\texttt{x})$ evolves to the new state $l'(\texttt{x})$ according to the dynamic f and the control actions $fb(\texttt{u})$ while the new environmental parameter value $r'(\texttt{env})$ is the same as the old value $r(\texttt{env})$ because it is time-invariant. Finally, the new percept value is derived from the new state $l'(\texttt{x})$ and environmental parameter $r'(\texttt{env})$.

Recall that the set of latent state variables of the real environment *REnv* contains both the state variable and environmental parameters, i.e., $LR = L \cup R = \{\texttt{x}\} \cup \{\texttt{env}\}$. If we can find a safe approximation M of the perception $nn \circ cam$, we equivalent can build a model of environment *Env* for *REnv* with the latent variables $L = \{\texttt{x}\}$, and the transition relation T_{Env} can be specified with the predicate:

$$T_{Env} \stackrel{\texttt{def}}{=} (\texttt{z} \in M(\texttt{x}) \wedge \texttt{x}' = f(\texttt{x}, \texttt{u}) \wedge \texttt{z}' \in M(\texttt{x}')) \tag{7.5}$$

where the approximation M and the dynamics f are given as expressions. Equivalently, this defines the transition relation:

$$[\![T_{Env}]\!]_{L,P,FB,L',P'} \stackrel{\texttt{def}}{=} \left\{ (l, p, fb, l', p') \;\middle|\; \begin{array}{c} p(\texttt{z}) \in M(l(\texttt{x})) \wedge \\ l'(\texttt{x}) = f(l(\texttt{x}), fb(\texttt{u})) \\ p'(\texttt{z}) \in M(l'(\texttt{x})) \end{array} \right\} \tag{7.6}$$

where $l, l' \in \mathcal{Q}_L, p, p' \in \mathcal{Q}_P$ and $fb \in \mathcal{Q}_{FB}$. Observe that the environmental parameter $\texttt{env}$ does not appear, and the transition relation $[\![T_{Env}]\!]$ is noticeably simpler than $\mathcal{T}_{REnv}$ for analysis. Further, if we can show M is an *abstraction* of $nn \circ cam$, which we will introduce later in Section 7.5, it follows from Proposition 3.2 that *Env* is an abstraction of *REnv*.

In short, finding safe approximations M for discrete transition systems helps synthesize safe abstractions of real environments in CPREACT models, so we will continue using the discrete transition system model for simpler notations in the rest of the chapter.

7.4 AN EXAMPLE: VISION-BASED LANE TRACKING

We now provide the details of the lane tracking control system in Figures 7.1 and 7.2.

Dynamics and Control The vehicle state $x \in \mathcal{X}$ consists of the 2D position (x, y) of the center of the front axle in a global coordinate system, and the heading angle θ with respect to the x-axis. The input $u \in \mathcal{U}$ is the steering angle δ. The discrete time model shown in Figure 7.3 is the well-known kinematic bicycle function [144] $f(x, u)$ with the center of the front axle as the body frame.

$$
\begin{aligned}
x_{t+1} &= x_t + v_f \cdot \cos(\theta_t + \delta) \cdot \Delta T \\
y_{t+1} &= y_t + v_f \cdot \sin(\theta_t + \delta) \cdot \Delta T \\
\theta_{t+1} &= \theta_t + v_f \cdot \frac{\sin(\delta)}{l_{WB}} \cdot \Delta T
\end{aligned}
$$

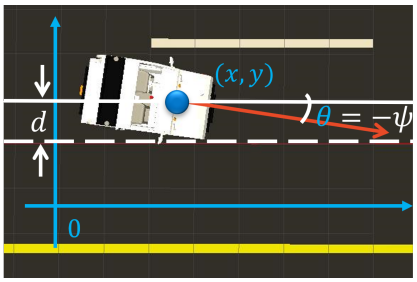


Figure 7.3: State (x, y, θ) and perception variables (d, ψ) for lane keeping. v_f, l_{WB}, and ΔT are constant values where v_f is the forward velocity, l_{WB} is the wheelbase, and ΔT is the time discretization parameter.

The input to f comes from the decision and control program. Here we use the standard Stanley controller [145] used for lateral control of vehicles. This controller uses the percept $z \in \mathcal{Z}$, which consists of the heading difference ψ and cross track distance d from the center of the lane to the ego-vehicle. In Figure 7.3, the heading θ coincides with the negation of the heading difference $-\psi$, but this happens only in the special case where the lane is aligned with the x-axis. The controller function $g(z)$ is defined as:

$$
\delta = g(d, \psi) = \begin{cases} \psi + \arctan\left(\frac{\kappa \cdot d}{v_f}\right), & \text{if } \left|\psi + \arctan\left(\frac{\kappa \cdot d}{v_f}\right)\right| < \delta_{max} \\ \delta_{max}, & \text{if } \psi + \arctan\left(\frac{\kappa \cdot d}{v_f}\right) \geq \delta_{max} \\ -\delta_{max}, & \text{if } \psi + \arctan\left(\frac{\kappa \cdot d}{v_f}\right) \leq -\delta_{max} \end{cases} \tag{7.7}
$$

where δ_{max} is the steering angle limit and κ is a controller gain parameter.

Perception The complicated perception pipeline estimates heading difference $\hat{\psi}$ and cross track distance $\hat{d}$ using several computer vision functions. First, the sensor function *cam* uses cameras to capture an image, and processes the image through cropping, undistortion, resizing, etc., to prepare the image *img* for the DNN. The particular DNN used here is LaneNet [146] which uses 512×256 RGB images to detect lane pixels. Internally, LaneNet contains two subnets for both the identification and instance segmentation of lane marking pixels. Then at post-process stage after LaneNet, curve fitting is applied on identified pixels to represent each detected lane as a polynomial function. Further, the perspective warping is applied to map the lanes to bird's eye view, which gives the percept $\hat{z} = (\hat{d}, \hat{\psi})$ as shown in Figure 7.2.

System Safety Requirement A common specification for lane keeping control is to avoid going out of the lane boundaries. We assume that the vehicle is driving on a straight road

with lane width W. For the purpose of simplifying exposition, we assume that the center line is aligned with the x-axis of the global coordinate system. Thus, the invariant set for safety can be specified as $[\![Safe]\!] = \{(x, y, \theta) \mid |y| \leq 0.5W\}$.

7.5 SAFE APPROXIMATE ABSTRACT PERCEPTION

In this section, we will discuss our method for constructing the approximation M for the perception system $nn \circ cam$. Section 7.5.1 sets the stage. It shows that plugging in any set-valued approximation M of $nn \circ cam$ naturally defines an approximation $\widehat{Sys}(M)$ of the original system Sys. Section 7.5.2 presents the main algorithm for constructing a particular type of M. It learns, from perception data, the center (mean) of the output set $M(x)$. Section 7.5.3 defines the next step in the construction of M. This step analyzes the control program and the vehicle dynamic $f \circ g$ to optimize the shape and the size of the output set around the mean, to assure the safety of $\widehat{Sys}(M)$ with respect to the invariant set $[\![Inv]\!]$. Section 7.5.4 establishes the safety of the constructed M, not only at the theoretical model level, but it also shows how to plug M in to the rest of the Sys code and verified using program analysis tools, namely CBMC [141] in our work. Finally, Section 7.5.5 discusses our methods for empirically evaluating the precision of M.

7.5.1 Approximate Abstract Perception in Closed-Loop

We will construct a set-valued perception function M that approximates the complex perception system $nn \circ cam$. For the safety requirement $[\![Safe]\!]$, our constructed function $M : \mathcal{X} \to 2^{\mathcal{Z}}$ should be such that when it is "substituted" in the closed-loop system of Equation (7.2), the resulting system is safe with respect to an invariant set $[\![Inv]\!]$ and, in turn, the safe set $[\![Safe]\!]$. Formally, substituting $nn \circ cam(x, env)$ with $M(x)$, the result is the non-deterministic system $\widehat{Sys}(M)$ given by:

$$x_{t+1} \in \{f(x_t, g(z)) \mid \exists z \in M(x_t)\}. \tag{7.8}$$

That is, when the actual system state is x_t (and the environmental parameters env), then the output from the abstract perception function M can be *anything* in the set $M(x_t)$. This set-valued approach is a standard way for modeling noisy sensors. Notice that we require M to be independent of environments.

Definition 7.1. A function $M : \mathcal{X} \to 2^{\mathcal{Z}}$ is an *abstraction* of $nn \circ cam$ if:

$$\forall env \in \mathcal{E}, \forall x \in \mathcal{X}. nn \circ cam(x, env) \in M(x). \tag{7.9}$$

This definition requires $M(x)$ covers all possible percepts from $nn \circ cam(x, env)$ for all states and environments, and that is why it is an abstraction.[16] If a function M is an abstraction of $nn \circ cam$, then it follows that $\widehat{Sys}(M)$ is an abstraction of Sys, that is, the set of executions of $\widehat{Sys}(M)$ contains the executions of Sys. Therefore, any state invariant $[\![Inv]\!] \subseteq \mathcal{X}$ for $\widehat{Sys}(M)$ carries over as an invariant of Sys.

Theorem 7.1. *If M is an abstraction of $nn \circ cam$, then $\widehat{Sys}(M)$ is an abstraction of Sys.*

Proof. Fixing an arbitrary initial state x_0 and an environment env, Theorem 7.1 follows immediately from Definition 7.1 by deriving:

$$f(x, g(nn \circ cam(x, env))) \in \{f(x, g(z)) \mid \exists z \in M(x)\}. \tag{7.10}$$

Subsequently, $\widehat{Sys}(M)$ can simulate every transition of Sys. QED.

Definition 7.1 is too general to be useful for constructing safe, intelligible, and precise abstractions. At one extreme, it allows the definition $M(x) \stackrel{\texttt{def}}{=} \{nn \circ cam(x, env) \mid \exists env \in \mathcal{E}\}$ which is a symbolic abstraction but does not help with intelligibility nor with safety. At the other end, we can make $M(x) \stackrel{\texttt{def}}{=} \mathcal{Z}$, which trivially covers all possible percept values but not useful for safety.

Our approach is to *utilize available information about safety of the control system without perception*. Informally, consider a version of the closed-loop control system that uses the ground truth values of ψ, d instead of relying on the vision pipeline to estimate these values. In order to prove safety of this *ideal system* with respect to $[\![Safe]\!]$, we can search for a standard invariant assertion $[\![Inv]\!]$ through various techniques [84, 88, 147, 148, 149]. We will construct M for Sys that can utilize the knowledge of such invariant $[\![Inv]\!]$.

Definition 7.2. Given a set $[\![Inv]\!] \subseteq \mathcal{X}$ and a function $M : \mathcal{X} \times \mathcal{E} \to 2^{\mathcal{Z}}$, M is *preserving* $[\![Inv]\!]$ if

$$\forall x \in [\![Inv]\!], \forall z \in M(x), f(x, g(z)) \in [\![Inv]\!]. \tag{7.11}$$

Finding an invariant preserving function satisfying Definition 7.2 will guide us towards creating more practical approximations of the perception system.

[16] We will see later that the M that we will construct cannot be guaranteed to satisfy this requirement, but this idea motivates it.

7.5.2 Learning Piecewise Approximations from Data

For an invariant preserving abstract perception function $M : \mathcal{X} \to 2^{\mathcal{Z}}$ to be intelligible, for any $x \in \mathcal{X}$, the output $M(x)$ should somehow be related to the ground truth value $z \in \mathcal{Z}$ that the perception system is supposed to estimate. For example, for a given state $x = (x, y, \theta)$ of the vehicle in the lane keeping system, the ground truth $z = (d, \psi)$—consisting of the relative position to lane center (d) and the angle with the lane orientation (ψ)—is uniquely determined by the geometry of the vehicle, the camera, and the lanes. The perception system $cam \circ nn$ is designed to capture this functional relationship between x and z (and it is affected by the environment env). For the sake of this discussion, let $h(x) = z$ be the idealized function that gives the ground truth percept z for any state x. A well-trained and well-designed perception system $nn \circ cam$ should minimize the error[17] $||h(x) - nn \circ cam(x, env)||$ over relevant states and environmental conditions. As M is an AAP of $nn \circ cam$, therefore, M should also minimize error with respect to $h(x)$ in order to achieve high precision.

In this chapter, we consider a piecewise affine structure of M. This is an expressive class of functions with conceptual and representational simplicity, and hence human-readable and comprehensible. First, given a partition $\{\mathcal{X}_i\}_{i=1\ldots N}$ of the target invariant domain, i.e., $[\![Inv]\!] = \bigcup_{i=1}^{N} \mathcal{X}_i$, we define M as:

$$M(x) = \begin{cases} \mathcal{R}_1(h(x)), & \text{iff } x \in \mathcal{X}_1 \\ \quad \vdots & \\ \mathcal{R}_N(h(x)), & \text{iff } x \in \mathcal{X}_N \end{cases} \tag{7.12}$$

where we search for $\mathcal{R}_i : \mathcal{Z} \to 2^{\mathcal{Z}}$ that returns a neighborhood around $h(x)$.

In what follows, we will show how $\mathcal{R}_i$'s can be derived as a linear function of $h(x)$ that is both safe with respect to the target invariant $[\![Inv]\!]$ and minimizes error with respect to training data available from the perception system. `ComputeAAP` gives our algorithm for computing this approximation for each partition $\mathcal{X}_i$.

To find a candidate $\mathcal{R}_i : \mathcal{Z} \mapsto 2^{\mathcal{Z}}$ for a given subset $\mathcal{X}_i \subseteq \mathcal{X}$, we consider that, when given z as input, $\mathcal{R}_i$ returns a parameterized ball defined as below:

$$\mathcal{R}_i(z) = \{z' \mid \|z' - (\mathrm{A}_i \times z + \mathrm{b}_i)\| \leq r_i\} \tag{7.13}$$

where the parameters A_i and b_i define an affine transformation from z to the ball's center,

[17] The precise choice of the error function is a design parameter, and we will discuss this in later sections.

Algorithm 7.1: Construction of the AAP M for partition $\mathcal{X}_i$. The output set is resented by a center defined by a transformation matrix A_i and a vector b_i, and a ball around the center defined by r_i.

Input: Subspace $\mathcal{X}_i$; Invariant $[\![Inv]\!]$; Dynamics f; Control g; Ideal Estimation h
Data: Training set of ground truth vs percepts $\{(z_1, \hat{z}_1), \ldots, (z_n, \hat{z}_n)\}$
Output: Linear Transform Matrix A_i; Translation Vector b_i; Safe Radius r_i

1 **Function** `ComputeAAP`
2 $\quad \mathrm{A}_i$, $\mathrm{b}_i \leftarrow$ `LinearRegression(`$\{(z_1, \hat{z}_1), \ldots, (z_n, \hat{z}_n)\}$`)`;
3 $\quad r_i \leftarrow$ `MinDist(`A_i,b_i,$\mathcal{X}_i$,$[\![Inv]\!]$,f,g,h`)`;
4 $\quad$ **return** A_i, b_i, r_i;

and r_i defines the radius. Here we are using a ball defined by the ℓ_2 norm on $\mathcal{Z}$. Our approach generalizes to other norms and linear coordinate transformations.

We start with the input to `ComputeAAP` in Algorithm 7.1. Besides the subset $\mathcal{X}_i \subseteq \mathcal{X}$, the invariant $[\![Inv]\!]$, aforementioned modules f, g, and h, `ComputeAAP` also requires a *training set* of pairs $(z, \hat{z})$ where $z = h(x)$ is the ground truth, and $\hat{z} = nn \circ cam(x, env)$ is the percepts obtained with the perception pipeline. These pairs can be obtained from existing labeled data for testing the vision pipeline or training CNNs. A labeled data point for nn is already an image $img = cam(x, env)$ sampled from $\mathcal{X}$ and $\mathcal{E}$ and its ground truth $z = h(x)$ as the label. In practice, the state $x = (x, y, \theta)$ can be obtained from other accurate sensors such as GPS to label the images. We use the state $x \in \mathcal{X}_i$ and obtain the ground truth $z = h(x)$. We then simply collect the perceived $\hat{z} = nn(img)$ by applying the vision pipeline on the image img.

`ComputeAAP` first uses the training set of pairs of $(z, \hat{z})$ to learn A_i and b_i using multivariate linear regression. The next section describes how it infers a safe radius r_i around the center $\mathrm{A}_i \times h(x) + \mathrm{b}_i$ by solving a constrained optimization problem.

7.5.3 Constructing Safe Approximations of Perception

At Line 2 of `ComputeAAP` in Algorithm 7.1, multivariate linear regression minimizes the distance from the center line $\mathrm{A}_i \times h(x) + \mathrm{b}_i$ to the training data in $\mathcal{X}_i$ and computes A_i and b_i. Next, we would like to infer a safe radius r_i around the center line $\mathrm{A}_i \times h(x) + \mathrm{b}_i$. There is a tension between safety and precision in the choice of r_i. On one hand, we want a larger radius r_i to cover more samples, making M a more conservative approximation of $nn \circ cam$. On the other hand, the neighborhood should not include any *unsafe perception value* that can cause a violation of $[\![Inv]\!]$.

Algorithm 7.2: Minimum distance to unsafe perception values.

Input: Linear Transform Matrix A_i; Translation Vector b_i; Subspace $\mathcal{X}_i$; Invariant $[\![Inv]\!]$; Dynamics f; Control g; Perfect Estimation h

Output: Safe radius $r_i \in \mathbb{R}_{\geq 0}$

Function `MinDist`

- solver.addVar(x, z, x')
- solver.addConstraints($x \in \mathcal{X}_i, x' = f(x, g(z)), x' \notin [\![Inv]\!]$)
- solver.setObjective($\|z - (\mathrm{A}_i \times h(x) + \mathrm{b}_i)\|$)
- $status, \hat{r}, b$ = solver.minimize()
- **if** *status is OPTIMAL or SUBOPTIMAL* **then**
  - $r_i \leftarrow \hat{r} - b$
- **else**
  - $r_i \leftarrow +\infty$ // status is *INFEASIBLE*
- **return** r_i

Formally, the set of unsafe percepts for a given state x is a function $unsafe : \mathcal{X} \rightarrow 2^{\mathcal{Z}}$

$$unsafe(x) \stackrel{\texttt{def}}{=} \{z \mid f(x, g(z)) \notin [\![Inv]\!]\} \tag{7.14}$$

and should be disjoint with the safe neighborhood. Figure 7.4 illustrates such a safe neighborhood for one particular state x. Note that $\mathcal{R}_i$ has to extend to all states $x \in \mathcal{X}_i$, and hence we need to find a safe radius r_i for any $x \in \mathcal{X}_i$. At the same time we would also like r_i as large as possible to cover more perceived values. Further, Figure 7.5 shows we have to infer for all $\mathcal{X}_i$ in the partition.

Our solution is to find an r_i just below the minimum distance r^* from the center $\mathrm{A}_i \times h(x) + \mathrm{b}_i$ to the set of unsafe percepts. This is formalized as the constrained optimization problem below:

$$r^* = \min_{x \in \mathcal{X}_i, z \in \mathcal{Z}, x' \in \mathcal{X}} \quad \|z - (\mathrm{A}_i \times h(x) + \mathrm{b}_i)\| \quad \text{s.t. } x' = f(x, g(z)), x' \notin [\![Inv]\!] \tag{7.15}$$

Observe that $x \in \mathcal{X}_i$ is a set of simple bounds on each state variable by designing the partition. $x' \notin [\![Inv]\!]$ is simply the invariant predicate over state variables. However, the third constraint $x' = f(x, g(z))$ encodes the controller g and dynamics f in optimization constraints. Encoding the dynamic model f as optimization constraints is a common technique in Model Predictive Control. Encoding the controller g can be achieved with a program analysis tool to convert each if-branch of control laws into equality constraints between z and controller output $u = g(z)$. An example template for Gurobi solver [150] is shown as `MinDist` in Algorithm 7.2.

We argue `ComputeAAP` computes a function $\mathcal{R}_i$ that returns a safe neighborhood for any ground truth percept $h(x)$.

Lemma 7.2. *For each $x \in \mathcal{X}_i$, $\mathcal{R}_i(h(x))$ computed by* `ComputeAAP` *is disjoint with the unsafe percepts, unsafe(x). That is,*

$$\forall x \in \mathcal{X}_i.(\mathcal{R}_i(h(x)) \cap \{z \mid f(x, g(z)) \notin [\![Inv]\!]\} = \emptyset) \tag{7.16}$$

Proof. Our proof is to analyze the possible outcome status from the optimization solver, and propagate each outcome through our functions. At Line 5, the solver may return the following statuses: (1) When *status=OPTIMAL* or *SUBOPTIMAL*, the solver returns an estimated minimum distance $\hat{r}$ and a bound b such that the true minimum r^* is within the bound, i.e., $\hat{r} \geq r^*$ and $\hat{r} - r^* < b$. Modern solvers all provide the bound to address numerical error or suboptimal solutions. Consequently, $r_i = \hat{r} - b$ at Line 7 ensures $r_i < r^*$; thus the ball defined by r_i is disjoint with the unsafe set. (2) When *status=INFEASIBLE*, the constraints are unsatisfiable, and the unsafe set $\{z \mid f(x, g(z)) \notin [\![Inv]\!]\}$ for every $x \in \mathcal{X}_i$ is proven to be $\emptyset$. We let $r_i = +\infty$ and thus $\mathcal{R}_i(x)$ is equivalent to the whole percept space $\mathcal{Z}$ for all $x \in \mathcal{X}_i$. QED.

7.5.4 Verifying with AAPs: Theory and Code

In this subsection, we summarize the claim that M computed by `ComputeAAP` indeed assures the safety of the approximated system $\widehat{Sys}(M)$ and show how it can be used for code-level verification. At a mathematical-level, the safety of M follows essentially from the construction in `ComputeAAP`. Using Proposition 7.2, we can show that M preserves the invariant $[\![Inv]\!]$.

Lemma 7.3. *If every function $\mathcal{R}_i : \mathcal{Z} \to 2^{\mathcal{Z}}$ returns the safe neighborhood of $\mathcal{X}_i$ for all i, then the AAP M preserves the invariant $[\![Inv]\!]$.*

Proof. Let us fix $x \in \mathcal{X}_i$ and the corresponding ground truth percept $h(x)$, and $\mathcal{R}_i(h(x))$ represents all percepts allowed by $\mathcal{R}_i$. Using the $\mathcal{R}_i$ computed by `ComputeAAP`, we have shown in Proposition 7.2 that $\mathcal{R}_i(h(x))$ does not intersect with any percept that can cause the next state $f(x, g(z))$ to leave $[\![Inv]\!]$. We then rewrite it as, for each $x \in \mathcal{X}_i$, any percept $z \in \mathcal{R}_i(h(x))$ preserves $[\![Inv]\!]$, i.e,

$$\forall x \in \mathcal{X}_i.\forall z \in \mathcal{R}_i(h(x)).f(x, g(z)) \in [\![Inv]\!]. \tag{7.17}$$

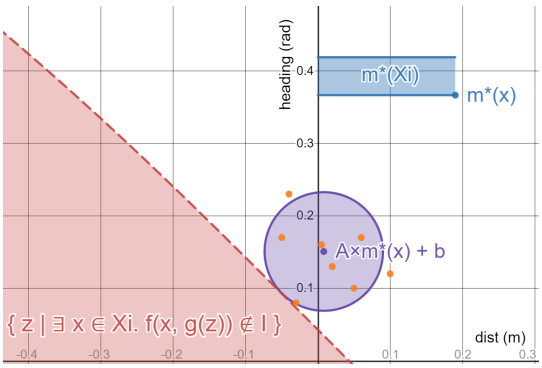


Figure 7.4: Example safe neighbor function $\mathcal{R}_i$ inferred from linear regression and constrained optimization.

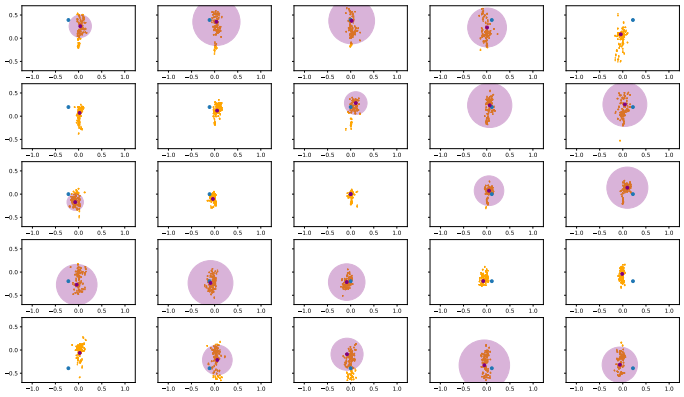


Figure 7.5: Ground truth (blue dot), perceived values (orange points), and inferred safe neighborhood (purple circle). Notice the biases in different subspace: the mean of the perceived values do not align with the ground truth.

$$
\begin{aligned}
&\mathcal{Z}\ \ M(\mathcal{X}\ \ x)\{\\
&\quad \_\_\text{CPROVER\_requires}\left(\textstyle\bigvee_{i=1}^{N} x \in \mathcal{X}_i\right);\\
&\quad \mathcal{Z}\ \ z\ =\ \text{nondet\_}z\,();\\
&\quad \_\_\text{CPROVER\_ensures}\left(\textstyle\bigwedge_{i=1}^{N} x \in \mathcal{X}_i \rightarrow z \in \mathcal{R}_i(h(x))\right);\\
&\quad \textbf{return}\ \ z;\\
&\}
\end{aligned}
$$

Figure 7.6: C code template for implementing the M function with CMBC's APIs.

Therefore, the invariant $[\![Inv]\!]$ is preserved for each subset $\mathcal{X}_i$, The proof of Proposition 7.3 is then to expand Definition 7.2 with the body of M and extend the guarantee from Equation (7.17) to all $x \in [\![Inv]\!]$ simply because $\{\mathcal{X}_i\}_{i=1\ldots N}$ covers $[\![Inv]\!]$. QED.

More importantly, the constructed AAP M can be plugged into the models of the system *Sys*, with different levels of detail, and verified using any number of powerful formal verification tools that have been developed over the past decades. For example, the abstract perception system could be plugged into the controller g and dynamics f functions represented by complex, explicit models, code, and differential equations, and we can verify the resulting system rigorously.

To illustrate this point, we showcase how to use M with C code implementations of g and f and verify the resulting system with CBMC [141] to gain a high-level of assurance for the control system. Recall our piecewise affine AAP defined in Section 7.5.2, it can be directly translated into program *contracts*, that is, preconditions and postconditions, supported by numerous existing program analysis tools [141, 142, 151, 152, 153]. In Figure 7.6, we imple-

ment M shown as C code in the following template with CBMC's APIs. We then are able to verify the whole system integrating the controller and the dynamics shown in the example code in Appendix A with CBMC.

7.5.5 Measuring the Precision of AAP

How close is the computed AAP M to the actual perception system $cam \circ nn$? As we discussed earlier, it is difficult if not impossible to rigorously answer this question because the perception system (and therefore the learning stage of M) depends on the *env* in complex and unknown ways. There are many options for measuring closeness that can factor in information about the environmental parameters.

We propose a simple and fine-grained empirical measure of precision. We fix a range of environmental parameter values $\mathcal{E}$. For each partition $\mathcal{X}_i$, we collect a *testing set* of pairs of $(z, \hat{z})$ by sampling across $\mathcal{X}_i \times \mathcal{E}$ using *some distribution* $\mathcal{D}$, where $\hat{z} = nn \circ cam(x, env)$ is the actual perception output and $z = h(x)$ is the ground truth. We denote a pair $(z, \hat{z})$ that satisfies $\hat{z} \in \mathcal{R}_i(z)$ as a positive pair. Then, the fraction of positive pairs gives us the empirical probability with respect to $\mathcal{D}$ that the actual perception system (with CNN) outputs percepts covered by $\mathcal{R}_i$. Formally, the empirical probability is defined as $\hat{p}_i = \frac{1}{N} \sum_{j=1}^{N} \mathbb{I}\left(\hat{z}_j \in \mathcal{R}_i(z_j)\right)$, where $(z_1, \hat{z}_1), (z_2, \hat{z}_2), \cdots, (z_N, \hat{z}_N)$ are i.i.d. samples from the distribution $\mathcal{D}$, and $\mathbb{I}$ is the indicator function. In contrast, the actual probability, which is more important, is defined as $p_i = \mathbb{E}_{\mathcal{D}}\left[\mathbb{I}\left(\hat{z} \in \mathcal{R}_i(z)\right)\right]$. Going forward we call p_i and $\hat{p}_i$ the *precision* and the *empirical precision* of M over the partition $\mathcal{X}_i$.

The following theorem immediately follows from Hoeffding's inequality, which bound the difference between the actual and the empirical probabilities.

Theorem 7.4. *For any $\beta \in (0, 1)$, with probability at least $1 - \beta$, we have that*

$$p_i \geq \hat{p}_i - \sqrt{-\frac{\ln \beta}{2N}}. \tag{7.18}$$

It may be tempting to interpret this probability as a probability of system-level safety, but without additional information how $\mathcal{D}$ is related to the actual distributions over $\mathcal{X}_i$ and $\mathcal{E}$, we cannot make such conclusions.

In the following sections, we use a uniform distribution $\mathcal{D}$ over simulated states and environments. Each heatmap shown in Figures 7.7 illustrates the empirical precision $\hat{p}_i$ of different $\mathcal{X}_i$ of M. A darker green $\mathcal{X}_i$ means that a higher fraction of outputs from the perception matches the provably safe AAP M. We ensure that at least $N = 300$ images are

collected for each $\mathcal{X}_i$. By Proposition 7.4, with probability at least 0.9 (i.e., $\beta = 0.1$), we have that $p_i \geq \hat{p}_i - 0.062$.

7.6 CASE STUDY 1: VISION-BASED LANE TRACKING WITH LANENET

We study the Polaris GEM e2 electric vehicle and its high-fidelity Gazebo simulation [46] (Motivating example of Section 7.4). The perception module uses LaneNet [146] for lane detection.[18] We discuss the construction of the approximation M in Section 7.6.1, the interpretation of the precision *heatmaps* in Section 7.6.2, finally, in Section 7.6.3 we study the behavior of the closed-loop system where LaneNet ($nn \circ cam$) is replaced by the approximation (M). We aim to study the impact of partitions $\{\mathcal{X}_i\}$ and the environment parameter distributions $\mathcal{D}$.

7.6.1 Implementation Details in Construction of AAPs

We recall the safe set is $[\![Safe]\!] = \{(x, y, \theta) \mid |y| \leq 2.0\}$, and the initial set of states is $\Theta = \left\{(x, y, \theta) \mid x = 0 \wedge |y| \leq 1.2 \wedge |\theta| \leq \frac{\pi}{12}\right\}$. We now briefly discuss that, in implementation, we need not specify an invariant candidate $[\![Inv]\!]$ explicitly by specifying an alternative constraint with a tracking error function.

A standard proof in control theory is to define a tracking error function (Lyapunov function) over the ground-truth percept values, and then prove that the error is non-increasing along the evolution of states for the entire state space (global Lyapunov stability) or for a neighbor around the equilibrium (local Lyapunov stability). This guarantees not only that the system eventually stabilizes to the equilibrium representing zero tracking error, but also that *any sublevel set of the error function* is an invariant set [154, Theorem 2]. Therefore, we can encode the constraint that the error is non-increasing along the evolution of states instead of specifying an invariant candidate.

Formally, given the equilibrium $\mathbf{0}_z \in \mathcal{Z}$, an error function $V : \mathcal{Z} \mapsto \mathbb{R}_{\geq 0}$ is a positive definite function, i.e., $V(\mathbf{0}_z) = 0$ and $V(z) > 0$ when $x \neq \mathbf{0}_z$. In this section, we use the vector norm, i.e., $V(d, \psi) = \|(d, \psi)\|$. Choices of different tracking error functions are discussed in Appendix A. The ground-truth percept value of a state (x, y, θ) is $(d, \psi) = h(x, y, \theta)$, and the tracking error is $V(d, \psi)$. Similarly, we denote the next state ass (x', y', θ'), the next ground-truth percept is obtained by $(d', \psi') = h(x', y', \theta')$, and the tracking error is $V(d', \psi')$. Recall that Lemma 7.2 reasons with the set of states leaving the invariant, i.e., $(x', y', \theta') \notin [\![Inv]\!]$.

[18] We use `https://github.com/MaybeShewill-CV/lanenet-lane-detection`, one of the most popular open source implementation of LaneNet on GitHub.

We similarly define the constraint of violating non-increasing error as:

$$V(h(x', y', \theta')) > V(h(x, y, \theta)) \tag{7.19}$$

which is used to replace $(x', y', \theta') \notin [\![Inv]\!]$. Detailed descriptions and values of parameters in f and g and the definition of h are in Appendix A.

To infer the AAP M, we consider the partitions $\{\mathcal{X}_i\}_{i \leq N}$ with y within $\pm 0.3W = \pm 1.2$ meters to ensure safety and heading angle θ within $\pm 15^\circ$, i.e.,

$$\bigcup_{i=1}^{N} \mathcal{X}_i = \left\{(x, y, \theta) \mid |y| \leq 1.2 \wedge |\theta| \leq \frac{\pi}{12}\right\} \tag{7.20}$$

Further, we consider three different partitions[19] $N \in \{8 \times 5, 8 \times 10, 8 \times 20\}$; larger numbers partition more finely and produce refinements of the coarser AAPs.

To prepare the training data for learning A_i and b_i to construct $\mathcal{R}_i$, we use the Gazebo model in [46] and generate camera images *img* labeled with their ground truth percepts z. Each image is sampled from an uniform distribution $\mathcal{D}$ over $\mathcal{X}_i \times \mathcal{E}$, where $\mathcal{E}$ is defined by:

(1) three types of roads with two, four, and six lanes,

(2) two lighting conditions, day and dawn.

The ground truth percept $z = h(x)$ is calculated using information from the simulator.

For each partition, given A_i and b_i learned from multivariate linear regression using the data. `MinDist`, implemented in Gurobi [150], solves the following nonlinear optimization problem to find r_i: Since each $\mathcal{X}_i$ covers an interval of 0.3 meter for y and 3° for θ. we discuss the optimization problem for a particular subset $\mathcal{X}_i$ that covers y from 0.9 to 1.2 meters and θ from 12° to 15° as an example, i.e, $\mathcal{X}_i = \left\{(x, y, \theta) \mid y \in [0.9, 1.2] \wedge \theta \in \left[\frac{\pi}{15}, \frac{\pi}{12}\right]\right\}$.

$$\begin{aligned} \min_{(x,y,\theta)\in\mathcal{X}_i,(d,\psi)\in\mathcal{Z},(x',y',\theta')\in\mathcal{X}} \quad & \|(d, \psi) - (\mathrm{A}_i \times h(x, y, \theta) + \mathrm{b}_i)\| \\ \text{subject to} \quad & x' = x + v_f \cos(\theta + g(d, \psi))\Delta T, \\ & y' = y + v_f \sin(\theta + g(d, \psi))\Delta T, \\ & \theta' = \theta + v_f \frac{\sin(g(d, \psi))}{l_{WB}} \Delta T, \\ & V(h(x', y', \theta')) > V(h(x, y, \theta)) \end{aligned} \tag{7.21}$$

[19]Here we do not partition along x because lanes are aligned with the x-axis, and partitioning x-axis does not produce interesting results.

All the computed AAPs were composed with the code for the controller g and the dynamics f and successfully verified for the corresponding invariant with CBMC. In addition to being an extra check, this CBMC verification closes the gap between the mathematical functions used in constructing the verified AAP, and the corresponding C functions in code (E.g., $\arctan\left(\frac{\kappa \cdot d}{v_f}\right)$ has to be implemented with `atan2` in C library to avoid division by zero).

7.6.2 Interpretation of the Precision of AAPs

Figure 7.7 shows the precision maps for three AAPs resulting from three increasingly finer partitions and two sets of testing environments. A darker green partition implies a higher empirical precision $\hat{p}_i$ (and therefore, a higher lower-bound of actual precision p_i by Proposition 7.4). That is, the safe AAP approximates the perception system with higher probability in those $\mathcal{X}_i$. First, we discuss the broad trends and then delve into the details.

At Equilibrium, AAP Breaks but It Does Not Matter All six heatmaps demonstrate a common trend where there is a lump of white (low score) cells around the origin. There are areas where either (1) the safe radius r_i of $\mathcal{R}_i$ is too small for M to include the outputs from $nn \circ cam$, *or* (2) the center of $\mathcal{R}_i$ is unsafe. This phenomenon can be understood as follows: First, the center (equilibrium) of the plot corresponds to near zero error in deviation d and heading ψ. Consider when a vehicle's state has nearly 0 tracking error; the percept must also approach the ground truth $h(x)$ so that the next state can maintain the 0 error. Recall that our AAP consists of the mean $\mathrm{A}_i \times h(x) + \mathrm{b}_i$ and the safe radius r_i. If $\mathrm{A}_i \times h(x) + \mathrm{b}_i$ already deviates from ground truth, it can lead to control actions to always increase tracking error in the next state. In this case, we cannot infer a safe region around the bias, and M returns an empty set. The precision is 0 by definition. In the other case, the mean is close to the ground truth. The safe region to maintain non-increasing error is still small, and hence r_i is almost 0. The precision will be very low because it is unlikely the percept from the vision pipeline to be extremely close to the ground truth. Alternatively, we can view the overall system $\widehat{Sys}(M)$ as a fixed-resolution quantized control system. It is well-known that such a system cannot achieve perfect asymptotic stability [155]. The feedback does not have enough resolution to drive the state to the equilibrium, and the error function V cannot be non-increasing around the origin. We note that not proving safety around the origin is less of a problem because the vehicle is safe—centered and aligned with the lane.

Finer Partitions Improve Precision We observe that finer partitions generate more precise approximations. With the finest partition, several cells achieve over 90 percent.

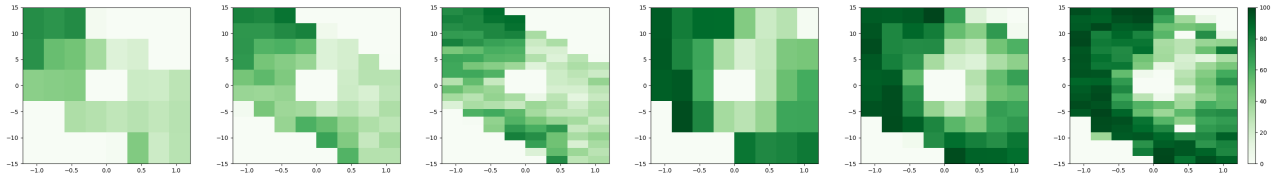


Figure 7.7: Precision heatmaps of AAPs inferred for LaneNet with Stanley controller. The partitions with $N = 8 \times 5$, 8×10, and 8×20. The environment parameter space with three road types (*Three from left*) and only two-lane road (*Three from right*).

This could be made higher with finer partitions. The reasons are twofold. (1) With a finer partition, linear regression can better fit a smaller interval of the original perception function. (2) The safe radius r_i is minimized for all $x \in \mathcal{X}_i$. If a subset $\mathcal{X}_j \subset \mathcal{X}_i$ excludes the worst state, the radius r_j for $\mathcal{X}_j$ can be larger than r_i.

Fewer Environmental Variations Improve Precision We generated two testing sets under different distributions over the environment space including (1) the same uniform distribution for the training set, and (2) an uniform distribution over the subspace with only the two-lane road. Observe the heatmaps in Figure 7.7, the colors become darker for the same cell locations. The variance in the perceived values by the vision pipeline reduces because of the fewer environmental variations. The same radius can cover more samples in the testing set.

7.6.3 Closed-Loop System with Approximate Perception Model

We test the performance of the worst AAP (with $N = 8 \times 5$ partitions) by simulating it in the closed-loop lane-keeping system in Gazebo (blue in Figure 7.8). At each time step, the AAP generates a set of possible percept values, and we randomly pick a point from this set and feed that into the controller g to close the loop. For comparison, we also run the original system with LaneNet (orange) and with perfect ground-truth (green) perception, starting from the exact same initial condition.

We run 50 simulation starting randomly from $0.6 \leq d \leq 0.9$ and $|\psi| \leq \frac{\pi}{60}$ each with time horizon 3s. For all these runs, we plot the mean and standard deviation of the perceived tracking error (left) and the actual tracking error (right) in Figure 7.8. First, we observe that, as expected, the distribution of tracking error using AAP and LaneNet are both biased compared to the ground truth. Second, the perceived tracking error from AAP is close to the tracking error of the actual system, and the real tracking error between the two (right) is even closer. These experiments provide empirical evidence that the closed-loop system

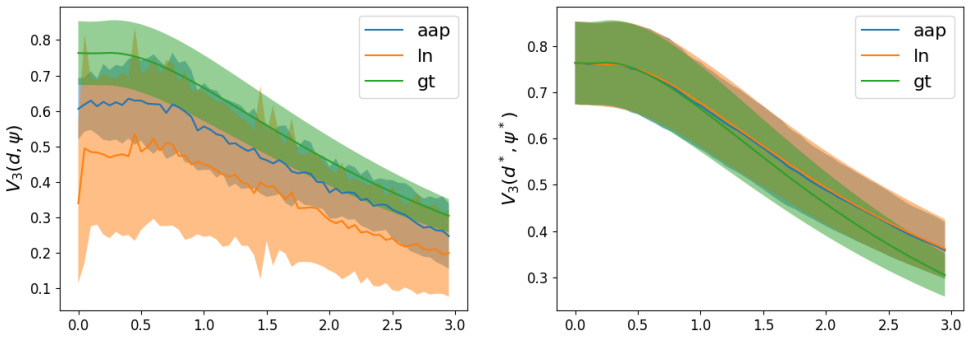


Figure 7.8: $V(d,\psi)$ (Left) computed using perception output, and $V(d,\psi)$ (Right) computed using ground truth.

with the AAP $\widehat{Sys}(M)$ closely approximates the actual system Sys on average.

7.6.4 Construction of AAPs Using Barrier Certificate Based Invariant

$\Theta = \left\{(x,y,\theta) \mid x = 0 \wedge |y| \leq 1.2 \wedge |\theta| \leq \frac{\pi}{12}\right\}$ is the initial set of states, and we recall the safe set is $[\![Safe]\!] = \{(x,y,\theta) \mid |y| \leq 2.0\}$. Given that the tracking error function $V(d,\psi) = \|(d,\psi)\|$ is a Lyapunov function, we can alternatively consider the sublevel set of V, $\mathcal{X}_\rho = \{x \mid V(h(x)) \leq \rho\}$, such that $\Theta \subseteq \mathcal{X}_\rho$ and $\mathcal{X}_\rho \subseteq [\![Safe]\!]$. The sublevel set $\mathcal{X}_\rho$ is guaranteed to be an invariant $[\![Inv]\!]$ by definition. This is equivalent to finding the barrier function $B(d,\psi) = V(d,\psi) - \rho$.

Given that h is linear for the LTC system and V is convex, it suffices to choose any ρ satisfying:

$$\sup_{x\in\Theta} V(h(x)) \leq \rho < \inf_{x\notin[\![Safe]\!]} V(h(x)) \tag{7.22}$$

to ensure $\Theta \subseteq \mathcal{X}_\rho$ and $\mathcal{X}_\rho \subseteq [\![Safe]\!]$. However, note that this is a sufficient condition, and ρ may not exist for any given tracking error function. In this case study, we can derive that $\sqrt{1.2^2 + (\frac{\pi}{12})^2} \leq \rho < 2.0$, and we choose $\rho = 1.27$.

For simplicity, we do not partition the invariant $\mathcal{X}_\rho$ and solve the optimization problem:

$$\begin{aligned}
\min_{(x,y,\theta)\in\mathcal{X},(d,\psi)\in\mathcal{Z},(x',y',\theta')\in\mathcal{X}} \quad & \|(d,\psi) - (\mathbf{A}\times h(x,y,\theta) + \mathbf{b})\| \\
\text{subject to} \quad & x' = x + v_f\cos(\theta + g(d,\psi))\Delta T, \\
& y' = y + v_f\sin(\theta + g(d,\psi))\Delta T, \\
& \theta' = \theta + v_f\frac{\sin(g(d,\psi))}{L}\Delta T, \\
& V(h(x,y,\theta)) \leq \rho, \quad V(h(x',y',\theta')) > \rho
\end{aligned} \tag{7.23}$$

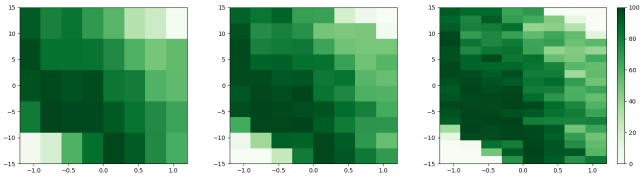


Figure 7.9: Precision heatmaps of AAPs inferred for LaneNet with respect to barrier function. The partitions with $N = 8 \times 5$, 8×10, and 8×20. The environment parameter space with only two-lane road.

The heatmaps in Figure 7.9 shows the same region of white cells on the top right and bottom left corners compared with Figure 7.7. These white cells indicate the system cannot satisfy the non-increasing error property, and they are also close to the boundary of the sublevel set $\mathcal{X}_\rho$. Hence, the states represented by these cells can leave the sublevel set and violate the invariant. On the other hand, this more relaxed invariant avoids the problem of requiring the perfect asymptotic stability around equilibrium. In return, it is required to first prove the given V is a Lyapunov function, and second find the right sublevel set $\mathcal{X}_\rho$ that separates the initial set from the unsafe set.

7.7 CASE STUDY 2: CORN ROW FOLLOWING AGBOT

Our second case study is the visual navigation system of the under-canopy agricultural robot (AgBot), CropFollow, developed in [156]. The system is responsible for the lateral control when the vehicle traverses the space between two rows of crops. Similar to our first case study, the system captures the image in front of the vehicle with a camera (Figure 7.10), applies a ResNet-18 CNN on the camera image to perceive the relative positions of the corn rows to the vehicle, and uses a modified Stanley controller to reduce the lateral deviation.

In CropFollow [156], the vehicle dynamics is approximated with a kinematic differential model of a skid-steering mobile robot. The state x consists of the 2D position x and y and the heading θ. The input u is the desired angular velocity ω. The modified Stanley controller takes a percept $z \in \mathcal{Z}$ composed of the heading difference ψ and cross track distance d to an imaginary center line of two corn rows, and outputs the angular velocity ω to steer the robot.

For the farm robots, we wish to avoid two undesirable outcomes:

(1) if $|y| > 0.5W = 0.38$ meters, the vehicle will hit the corn, or

(2) if $|\theta| > 30°$, the neural network output becomes highly inaccurate and recovery may be impossible.

Figure 7.10: Real and simulated camera images for corn row following for agricultural robots.

The safe set is then defined as $[\![Safe]\!] = \{(x, y, \theta) \mid |y| \leq 0.38 \wedge |\theta| \leq \frac{\pi}{6}\}$. We use the error function $V(d, \psi) = |\psi + \arctan(\frac{\kappa \cdot d}{v_f})|$ from [145] to specify the invariant $[\![Inv]\!]$. The definition of the dynamics and controller, the partitions of states, environments, constants for the dynamics, and the computed precision heatmaps are provided in Appendix B.

We observe almost identical broad trends including the white cells around equilibrium, the white spots in the upper right and lower left corners close to the violation of invariant, and higher precision score with finer partitions. This case study reaffirms the validity of our interpretation over the precision heatmap in Section 7.6. It also showcases that our analysis can be applied on different vision-based control systems.

7.8 DISCUSSION AND FUTURE DIRECTIONS

Safety assurance of autonomous systems that use machine learning models for perception is an important challenge. We presented an approach for creating approximate abstractions for perception (AAP) that are safe by construction. The approach learns piecewise affine set-valued AAPs of the perception system from data. Viewing AAPs along the triple axes of safety, intelligibility, and precision may give a productive perspective for tackling the problems of safety assurance of autonomous systems.

Within the space of intelligible AAPs, we have explored one corner with piecewise affine models. Our piecewise affine AAPs use uniform rectangular partitions, and the size of the partitions have significant impact on improving precision. The results suggest that non-uniform or adaptive partitioning (e.g., finer partitions nearer to the equilibrium) would yield more precise approximations. Exploration of other structures such as decision trees, polynomial models, and space partitions, would be fruitful from the point of achieving precision without making the partition size too big. Another direction is to derive templates preserving the structural properties of the control programs or the physical environment. For example, we may require the behavior of the vehicle to be symmetric that the reaction to facing the right lane boundary (both heading and distance are positive) is symmetric to the reaction to

facing the left lane boundary (both heading and distance are negative). If the controller and dynamics are invariant under translation and reflection, the learned approximation should also preserve the symmetry, and this may be enforced by the template.

As expected, the safety requirement and its verification method (e.g., invariants and Lyapunov functions) significantly impact the precision of the constructed approximation model. The precision maps shed light on parts of the state space and environment where the actual vision-based perception system is most fragile and is likely to violate requirements. Such quantitative insights can inform design decisions for the perception system, the control system, and the definition of the system-level *operating design domains (ODDs)*.

Finally, we chose to use discrete time models and used CBMC for verifying the closed-loop system with the AAP. We have shown that searching for AAPs for a discrete time model is in fact synthesizing approximate safe abstractions for an equivalent CPReact model, and extending the approach to find AAPs for continuous time and hybrid models can be achieved in the CPReact modeling framework. However, it will require solutions of ODEs of continuous systems in place of dynamic functions f of discrete systems for the encoding of the constrained optimization query, and solving the optimization query and verifying the closed-loop system with the AAP will require nontrivial extensions of existing optimization and verification tools.

Chapter 8: Approximate Abstraction for Vision-Based Drone Formation

Vision-based formation control systems recently have attracted attentions from both the research community and the industry for its applicability in GPS-denied environments. As discussed in Chapter 7, the safety assurance for such systems is challenging due to the lack of formal specifications for computer vision systems. The compound effect from interactions between robots further complicates the impact of imprecise vision-based perception. In this chapter, we propose a technique for safety assurance of vision-based formation control. Inspired by our approach in Chapter 7, our technique combines (1) the construction of a piecewise approximation of the worst-case error of perception and (2) a classical Lyapunov-based safety analysis of the consensus control algorithm. The analysis provides the ultimate bound on the relative distance between drones. This ultimate bound can then be used to guarantee safe separation of all drones. We implement an instance of the vision-based formation system on top of the photo-realistic AirSim simulator [47]. We construct the piecewise approximation for varying perception error under different environments and weather conditions, and we are able to validate safe separation provided by our analysis across different weather conditions with AirSim simulation. The result in this chapter is summarized in our preprint paper available on arXiv [157].[20] Our implementation of vision-based formation control and the code for simulation and analyses are all publicly available[21].

8.1 OVERVIEW

The literature on distributed consensus, flocking, and formation control is vast (see, for example [158, 159, 160, 161]). Flocking and swarm formation using computer vision [162, 163, 164] can leverage the advances in deep learning. They do not require localization systems, and thus, are attractive for GPS-denied environments. However, safety assurance of vision-based control systems poses challenges: (1) Formal specifications for computer visions systems are difficult, (2) deep learning-based perception functions can be fragile, and (3) safety analysis requires one to understand the impact of *imprecise* vision-based state (position) estimation on distributed control. These challenges have been recognized by the autonomy industry [165].

In this chapter, we present a technique for safety assurance of a vision-based swarm formation control system. The computer vision pipeline here uses feature detection, feature

[20] This is a joint work with Yangge Li and Yubin Koh.

[21] `https://gitlab.engr.illinois.edu/aap/airsim-vision-formation`

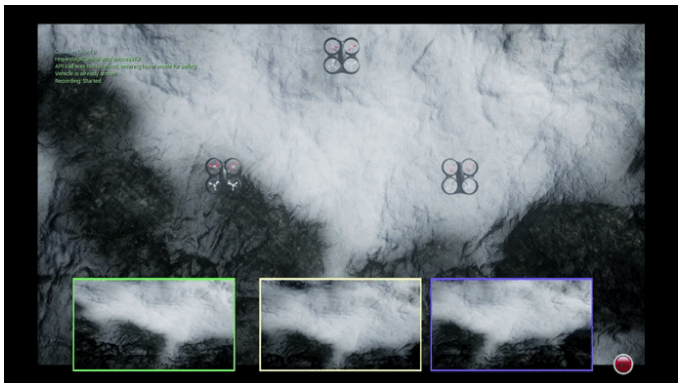


Figure 8.1: Vision-based drone formation using downward facing camera images in AirSim.

matching across a pair of images, and geometric models for 3D-vision to estimate the pairwise relative poses of the participating drones. The estimated relative poses are then used by a consensus-based formation control algorithm to achieve target formations. To our knowledge, this is the first investigation of end-to-end safety assurance of such systems. Our technique combines (1) Lyapunov analysis of the formation control algorithm with (2) an *approximate abstraction* for the vision-based perception component. The concept of the approximate abstraction was introduced in Chapter 7 on vision-based lane keeping control. The key idea was to approximate the worst case behavior of the complex perception system with a low-dimensional, and empirically precise, function of the ground truth lane-deviation values. Formation control for drones uses a completely different type of perception with pairs of images, feature matching, and camera geometry. Not surprisingly, a completely different method is needed for constructing the approximate abstraction. The two key challenges are:

(1) The perception error impacts the behavior of agents in a distributed system.

(2) The upper bound on the perception error for a pair of agents depends on their ground truth relative position.

In general, perception errors can get worse as the system approaches the equilibrium (desired formation), and thus, make stabilization difficult. Asymptotic stability may be unachievable, and we have to settle for the more practical notion of ultimate boundedness [166]. Our analysis gives an empirical method for testing out the environmental conditions (e.g., lighting, fog) under which a target formation can be safely achieved, despite perception errors. Thus, this study shows that on top of the safety assurance, the methodology with approximate abstractions may be useful for creating safe operating guidelines for vision-based swarms.

In summary, our contributions are as follows:

(1) An approach to construct a piecewise approximate abstraction to bound the perception error of the vision component.

(2) A Lyapunov-based analysis to derive the ultimate bound for the end-to-end drone formation system using the approximate abstraction.

(3) Detailed empirical evaluations with the photo-realistic AirSim simulator [47].

Connections to CPReact System Models We have presented in Chapter 2 how to model dynamical systems in our CPReact model, and we have shown that it can complicate the notations. We have discussed in Chapter 7 that finding approximate abstractions for perception components can help synthesize safe abstractions of the real environment. In this chapter, we again use dynamical system models instead of our CPReact model to simplify the notations, and we will only discuss finding abstractions for approximate perception components instead of the abstraction of the environment.

Related Works There is a line of work on the analysis of closed-loop systems with vision-based perception. Our approach in Chapter 7 provides the insight of approximate abstractions but focuses on the lane tracking system. VerifAI [27] uses techniques like fuzz testing and simulation to falsify the system specifications. Katz et al. [28] trains generative adversarial networks (GANs) to produce a network to simplify the image-based NN. NNLander-VeriF [29] verifies NN perception along with NN controllers for an autonomous landing system. In contrast, our approach is the first to provide safety analyses for a formation control system with vision-based perception, and we apply the notion of ultimate boundedness for safe separation and formation.

Organization In Section 8.2, we introduce the formation control system with the vision-based perception. We briefly review a well-studied controller. In Section 8.3, we describe the approximate abstraction for perception error bounds via sampling from vision-based pose estimation. In Section 8.4, we show safe separation under perception error using our main theory of ultimate boundedness. We then validate with AirSim simulation in Section 8.5 and discuss in Section 8.6.

8.2 VISION-BASED FORMATION CONTROL

We will study a distributed formation control system with N identical aerial vehicles or *agents* as shown in Figure 8.1. The target formation is specified in terms of relative positions

between agents. Each agent i has a downward facing camera, and it uses images from its own camera and its neighbor j's camera to periodically estimate the relative position of j with respect to i. Based on these estimated relative positions to all its neighbors, the agent i then updates it own position by setting a velocity to achieve the target formation.

Before describing the vision and control modules in more detail, we introduce some notations used throughout the paper. First, the neighborhood relation between agents is defined by an $N \times N$ symmetric adjacency matrix $W \in \mathbb{R}^{N \times N}$, i.e., j is a neighbor of i if and only if $\omega_{ij} = 1$, otherwise $\omega_{ij} = 0$. All diagonal entries of W are $\omega_{ii} = 0$.

Second, we only consider planar formations for simplicity though the agents are in 3-dimensional space. We use the technique to represent a vector in $\mathbb{R}^2$ with a complex number in $\mathbb{C}$. The position of the agent i in the world frame is represented by a complex number $q_i \in \mathbb{C}$ and its input velocity by another number $u_i \in \mathbb{C}$. The state of the overall system is $\mathrm{q} = [q_1\, q_2\, \cdots q_N]^T$. A desired formation is specified as the set of target equilibrium states defined by a desired state q^*:

$$\mathrm{E}_{\mathrm{q}^*} \stackrel{\texttt{def}}{=} \{\mathrm{q} \mid \bigwedge_{i=1}^{N-1} \bigwedge_{j=i+1}^{N} q_j - q_i = q_j^* - q_i^*\}. \tag{8.1}$$

That is, $\mathrm{E}_{\mathrm{q}^*}$ is the set of all states that form q^* up to position translations. We also specify a safe set demanding that the distance between any two agents are never too close, namely

$$[\![\mathit{Safe}]\!] \stackrel{\texttt{def}}{=} \{\mathrm{q} \mid \bigwedge_{i=1}^{N-1} \bigwedge_{j=i+1}^{N} \|q_j - q_i\| > 0\} \tag{8.2}$$

We now discuss the components of each agent i (Figure 8.2).

8.2.1 Vision-Based Relative Pose Estimation

Agent i's downward-facing camera *cam* periodically generates an image of the ground img_i, which depends on its state q_i and other environmental factors like background scenery, lighting, fog, etc. The neighboring agent j generates another image img_j of the ground and shares this with the agent i over the communication channel. We assume the whole system runs in synchronous mode, i.e., all the drones will capture the image at the same time and there's no communication delay between drones while sharing the images. The vision-based pose estimation algorithm *cv* takes a pair of images, img_i and img_j, as an input and produces the estimated relative position $\hat{q}_{ij}$ of agent j with respect to agent i following these steps:

(1) First, *cv* detects features from each image. Any of the various feature detection

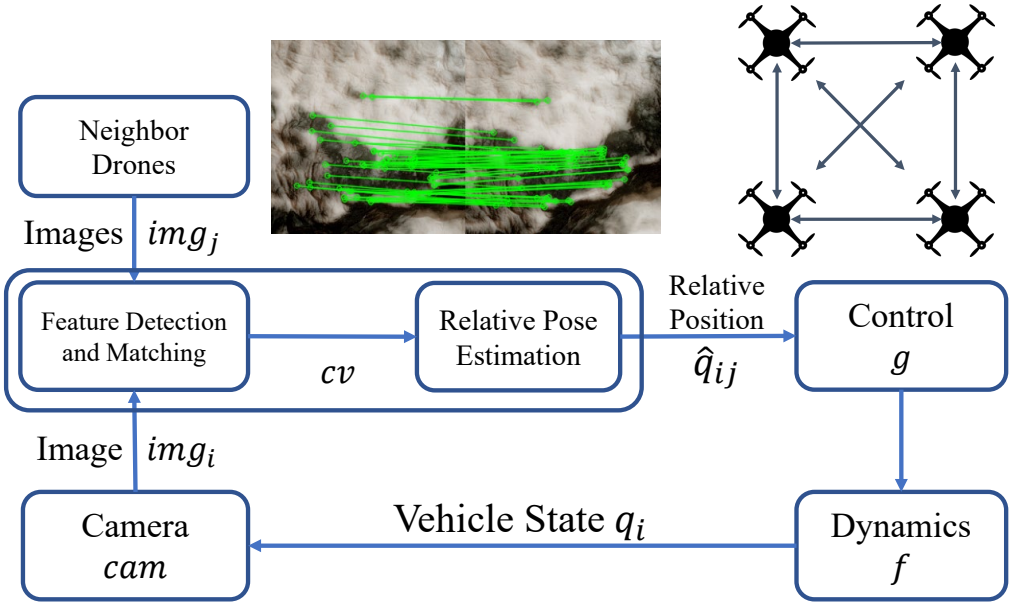


Figure 8.2: Architecture of an agent in the vision-based formation control systems.

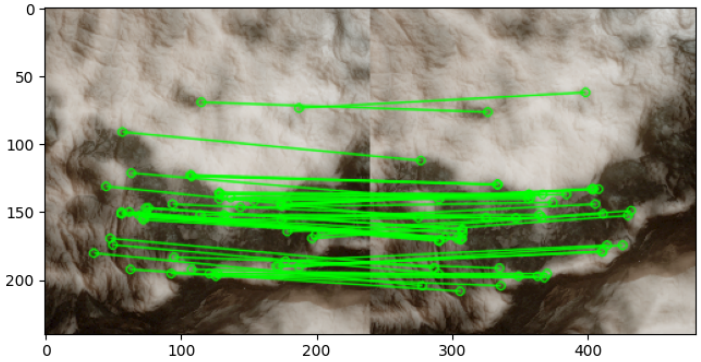


Figure 8.3: Feature matching on a pair of images collected in AirSim.

algorithms like SIFT [167], SURF [168], and ORB [169] can be used for this step. (2) Then, *cv* collects the detected features from the pair of images, and a feature matching algorithm (such as FLANN [170]) is used to match pairs of features in each image as shown in Figure 8.3. (3) For each feature point, we can set up a relationship between the pixel coordinate and the world coordinate of the feature point

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K[R \mid t] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{8.3}$$

where $[u, v, 1]^T$ is the pixel coordinate, $[x, y, z, 1]$ is the feature's world coordinate, K is the camera intrinsic matrix and $[R \mid t]$ is the extrinsic camera parameters. With a set of at least eight matched features, we can come up with eight pairs of equations between the pose of two cameras and by solving these equations, the relative rotation and the normalized translation vector can be calculated using the inverse geometry of image formation in cameras. Examples of this step appear in [171, 172, 173]. Further, the altitude and drone orientation can be used to estimate the distance to ground and recover the length of the translation vector.

The accuracy of the perception pipeline can be influenced by many factors. The change of environments such as background, lighting, and weather influence the quality of the image and the image features, which in turn influence the accuracy of relative pose estimation.

8.2.2 Formation Control

The relative pose estimates computed by agent i's perception modules are used to compute its velocity control inputs u_i, which in turn affects its position q_i. We consider the simple single-integrator dynamics relating q_i and u_i:

$$\dot{q}_i = u_i \tag{8.4}$$

The velocity control input is calculated using the well-known averaging rule [174]:

$$u_i = \sum_{j=1}^{N} \omega_{ij}(\hat{q}_{ij} - q^*_{ij}). \tag{8.5}$$

where we denote $q_{ij} = q_j - q_i$ and $q^*_{ij} = q^*_j - q^*_i$. In [174], it is proven that this controller stabilizes the system to a desired formation *if there is no perception error*, i.e., $\hat{q}_{ij} = q_{ij}$.

Finally, we study the ultimate boundedness [166] of the system. For the formation control, the ultimate bound is a constant b bounding the relative distance when the system is stabilized. Formally, a state q within the ultimate bound b is that, for each pair of $i, j \in \{1 \dots N\}$

$$\|q^*_{ij} - q_{ij}\| \leq b. \tag{8.6}$$

8.3 PIECEWISE APPROXIMATE ABSTRACTION

In this section, we first develop a model of the perception error which will be used later in our safety analysis. For vision-based perception, uniform worst case bounds on the perception error $\hat{q}_{ij} - q_{ij}$ can be overly conservative for system-level analysis. We have shown in

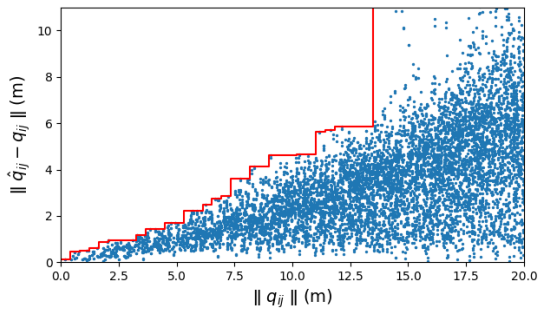


Figure 8.4: Perception error distances $\|\hat{q}_{ij} - q_{ij}\|$ (Blue dots) and empirical piecewise constant upper bound (Red line) with respect to true relative distances $\|q_{ij}\|$. We fix drone i as the origin and uniformly sample 10,000 positions of drone j within a circle of 20 meters in the LandscapeMountains environment from AirSim.

Chapter 7 that state-dependent error models can strike a balance between the conservatism of the safety analysis and the precision of characterizing vision-based perception systems.

Following the above pattern, we investigate the relationship between the ground truth and the perceived relative poses. We randomly sample pairs of camera images from two drones under different relative positions in AirSim. For each sample, we obtain a pair of true relative position q_{ij} from AirSim and perceived relative position $\hat{q}_{ij}$ via vision-based pose estimation pipeline (of Section 8.2.1). Figure 8.4 plots the norm of perception error $\|\hat{q}_{ij} - q_{ij}\|$ with respect to the true relative distance $\|q_{ij}\|$. We observe that the norm of the worst-case perception error indeed increases with respect to the true relative distance. Secondly, the error bound grows sharply when the relative distance crosses a certain threshold, e.g., about 14 meters in Figure 8.4. This is not too surprising: As the two drones become farther apart, the intersection of the two camera views is smaller, and there are fewer matched features than eight pairs, which leads to the failure of relative pose estimation.

For the analysis later in Section 8.4, our goal is to find an abstraction of the vision component with its worst perception error, and we empirically approximate the worst perception error from collected samples. Hence, we define the *approximate abstraction* as the piecewise constant function bounding the perception errors[22] as illustrated by the red line in Figure 8.4. Formally, given a sequence of $0 < d_1 < d_2 < \cdots < d_n$, we find a non-decreasing sequence $\gamma_1 \leq \gamma_2 \leq \cdots \leq \gamma_n$, such that γ_k serves as the empirical upper bound on the error for all

[22]Other possible error models could also be investigated. For example, we can consider the piecewise affine model in Chapter 7.

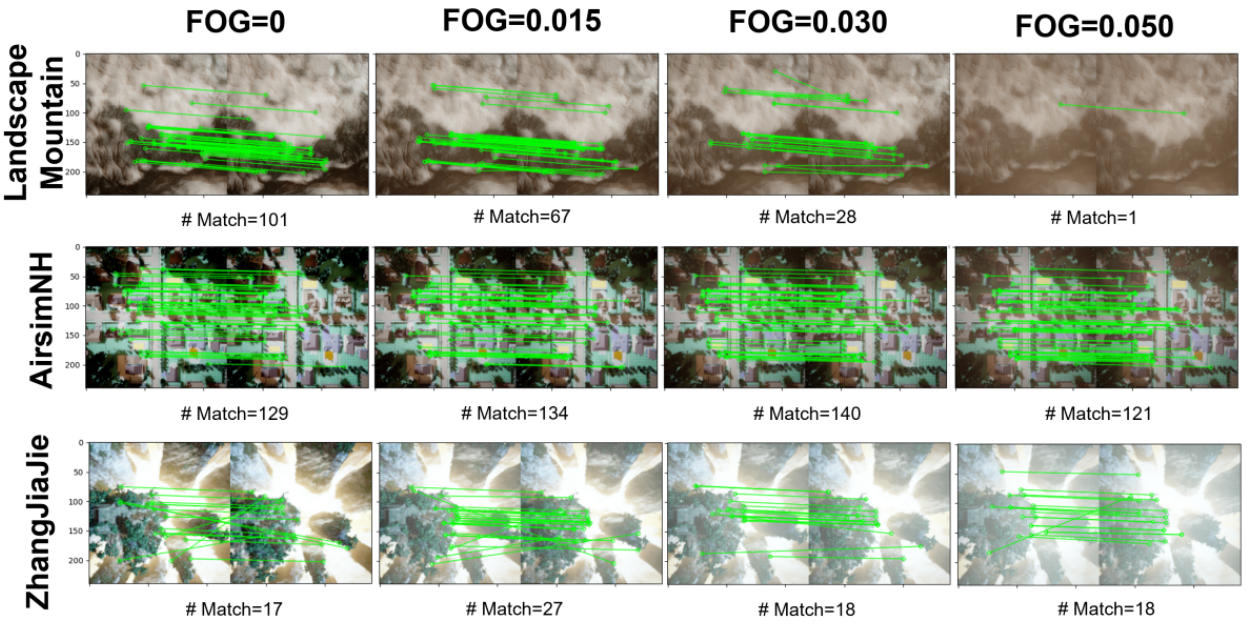


Figure 8.5: Matched feature points found by the feature detection and matching algorithm under three AirSim environments and four fog levels.

intervals $(d_{k-1}, d_k]$. We then construct the monotonic piecewise constant function $\gamma_{\max}$,

$$\gamma_{\max}(q_{ij}) = \begin{cases} \gamma_1, & \text{if } 0 \leq \|q_{ij}\| \leq d_1 \\ \vdots & \\ \gamma_n, & \text{if } d_{n-1} < \|q_{ij}\| \leq d_n \\ \infty, & \text{if } d_n < \|q_{ij}\| \end{cases} \tag{8.7}$$

The approximate abstraction will depend on environmental factors. To systematically study the impact of environmental variations on the approximate abstraction, we experimented with different environments and weather conditions in the photorealistic AirSim simulator. Figure 8.5 shows how the feature matching step degrades across three environments (namely LandscapeMountains, AirSimNH, and ZhangJiajie) and four fog levels. Note that at the fog level 0.050, only one pair of matching features is detected for LandscapeMountains for the same relative position.

Figure 8.6 shows the approximate abstractions for four fog levels under LandscapeMountains. The perception error bound increases much faster (against the relative distance) in a foggier weather. To better visualize this trend, we normalize the perception error, $\|\hat{q}_{ij} - q_{ij}\|$, to the *relative perception error*, $\frac{\|\hat{q}_{ij} - q_{ij}\|}{\|q_{ij}\|}$, and plot the 25%, 50%, and 75% percentiles of relative perception error for finer fog levels in Figure 8.7. We observe in Figure 8.7 that the

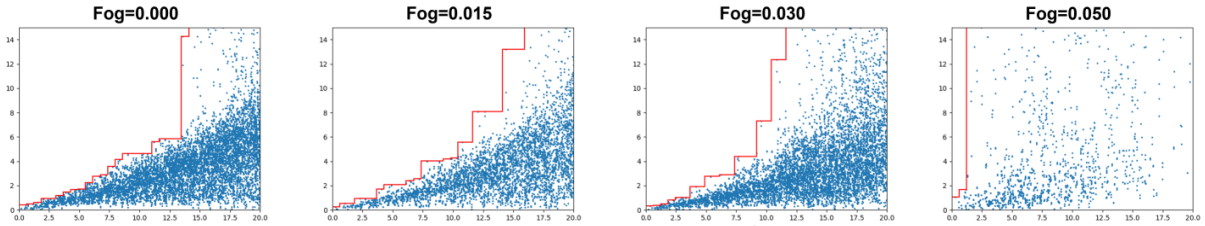


Figure 8.6: Perception error distances $\|\hat{q}_{ij} - q_{ij}\|$ and empirical piecewise constant upper bound with respect to true relative distances $\|q_{ij}\|$ under varying fog levels.

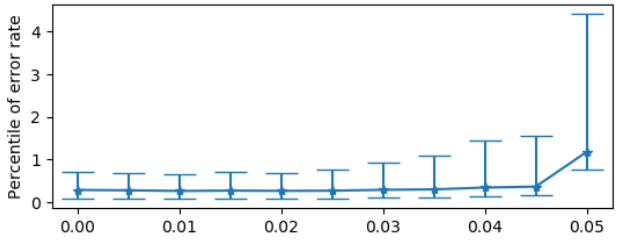


Figure 8.7: 25%, 50%, and 75% percentiles on *relative perception error* $\frac{\|\hat{q}_{ij}-q_{ij}\|}{\|q_{ij}\|}$ with respect to different fog levels under LandscapeMountains.

50% value (median) mostly remains the same except until fog level is 0.05, but the value of 75% percentile increases more significantly. This is consistent with the observation on the approximate abstraction.

8.4 ULTIMATE BOUND ON RELATIVE POSITIONS

The controller in Equation (8.5) assumes the true relative positions q_{ij} are used. However, we already show in Section 8.3 that the estimated relative positions from vision algorithms are imprecise and affected by the ground truth and environmental variations. In this section, we first assume that the maximum perception error from vision algorithms are bounded. We show that the true relative distance between agents is ultimately bounded around the desired formations, and the ultimate distance bound is limited by a constant multiplied to the perception error bound. We then discuss the analysis using the approximate abstraction obtained in Section 8.3.

First, we introduce the graph induced Laplacian matrix, the error dynamics without perception error, and the Lyapunov function for reference in the subsequent proofs. The graph

induced Laplacian matrix $\boldsymbol{L}$ is defined using the adjacency matrix W as:

$$l_{ij} = \begin{cases} \sum\limits_{k \in \{1 \dots N\}} \omega_{ik} & i = j \\ -\omega_{ij} & i \neq j \end{cases} \tag{8.8}$$

$\boldsymbol{L}$ is symmetric and positive semi-definite for a simple undirected graph. There exists an eigendecomposition $\boldsymbol{L} = Q \Lambda Q^T$, where Λ is a diagonal matrix $diag(\lambda_1, \lambda_2, \dots, \lambda_N)$ with eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_N$ and Q is orthogonal. The smallest eigenvalue λ_1 of $\boldsymbol{L}$ is always 0 with the eigenvector $\vec{\mathbf{1}} = [1, 1, \dots, 1]^T$, and the second smallest eigenvalue λ_2 of $\boldsymbol{L}$ is called the *algebraic connectivity* of the interaction graph [175]. Further, if the graph is connected, then $rank(\boldsymbol{L}) = N - 1$ and $\lambda_2 > 0$.

Given the error state defined as $\mathrm{e_q} = \mathrm{q}^* - \mathrm{q}$, the error dynamics without perception error is derived as:

$$\dot{\mathrm{e_q}} = -\boldsymbol{L} \mathrm{e_q} \tag{8.9}$$

We use the quadratic function V below to prove the Lyapunov stability of the system without perception error and the ultimate boundedness of the system with perception error:

$$V(\mathrm{e_q}) = \frac{1}{2} \sum_{i \in \{1 \dots N\}} \sum_{j \in \{1 \dots N\}} \omega_{ij} \| q_{ij} - q_{ij}^* \|^2 = \frac{1}{2} \mathrm{e_q}^T \boldsymbol{L} \mathrm{e_q} \tag{8.10}$$

Theorem 8.1. *If the communication graph is connected, i.e., $\lambda_2 > 0$, and there is no perception error, i.e., $\hat{q}_{ij} = q_{ij}$, then the system in Equation* (8.9) *achieves state synchronization, i.e., the error state* $\mathrm{e_q}$ *converges to vectors in span$\{\vec{\mathbf{1}}\}$ with the exponential rate λ_2. Subsequently, the state of the formation system* q *converges to desired formations in* $\mathrm{E_{q^*}}$.

Proof. The Lyapunov stability for the controller in [174] is a well-established result. Here we provide an alternative proof from [176] to set the stage for the next theorem on the ultimate boundedness. Let the eigendecomposition of the Laplacian matrix be $\boldsymbol{L} = Q \Lambda Q^T$ where Q is an orthogonal matrix and Λ is the diagonal matrix from the eigenvalues $\Lambda = diag(0, \lambda_2, \dots, \lambda_N)$. Let $\eta = Q \mathrm{e_q}$. We can rewrite the system in Equation (8.9) as:

$$\dot{\eta} = Q^T(-\boldsymbol{L} Q \eta) = -\Lambda \eta \tag{8.11}$$

We then let $\tilde{\Lambda}^{\frac{1}{2}} = diag(\sqrt{\lambda_2}, \dots, \sqrt{\lambda_N})$ and let $x = \tilde{\Lambda}^{\frac{1}{2}} [\eta_2 \ \dots \ \eta_N]^T = [\sqrt{\lambda_2}\eta_2 \ \dots \ \sqrt{\lambda_N}\eta_N]^T$. The subsystem for $i \in \{2 \dots N\}$ is derived as:

$$\dot{x} = -x \tag{8.12}$$

Theorem 2.5 in [176, Section 2.4.1] shows that the original system of Equation (8.9) converges to an equilibrium if and only if the system of Equation (8.12) is asymptotically stable. An intuitive explanation of the above necessary and sufficient condition is as follows: the orthogonal matrix Q is a rotation or reflection matrix for coordinate transformation, and moving along the axis of η_1 is equivalent to moving $\mathrm{e_q}$ along the direction of $\vec{\mathbf{1}}$ in the system of Equation (8.9). Hence, it has no effect on either converging to or diverging from desired formations in $\mathrm{E_{q^*}}$.

We use the quadratic Lyapunov function $\tilde{V}(x) = \frac{1}{2}x^T x$, and the derivative of $\tilde{V}$ is:

$$\dot{\tilde{V}}(x) = -x^T x = -\|x\|^2 \tag{8.13}$$

It guarantees that the system of Equation (8.12) is exponentially stable with the convergence rate 1. Further, recall that $\eta = Q\mathrm{e_q}$ and $\Lambda = diag(0, \lambda_2, \ldots, \lambda_N)$,

$$\tilde{V}(x) = \frac{1}{2}x^T x = \frac{1}{2}\eta^T \Lambda \eta = \frac{1}{2}\mathrm{e_q}^T Q^T \Lambda Q \mathrm{e_q} = \frac{1}{2}\mathrm{e_q}^T \boldsymbol{L} \mathrm{e_q} = V(\mathrm{e_q}) \tag{8.14}$$

Because $x = \tilde{\Lambda}^{\frac{1}{2}}[\eta_2 \ \ldots \ \eta_N]^T$, we can derive

$$V(\mathrm{e_q}) = -\|\Lambda^{\frac{1}{2}} Q \mathrm{e_q}\|^2 \leq -\|\Lambda^{\frac{1}{2}}\|^2 \|Q\mathrm{e_q}\|^2 \leq -\lambda_2 \|\mathrm{e_q}\|^2 \tag{8.15}$$

Therefore, the original system of Equation (8.9) converges to desired formations in $\mathrm{E_{q^*}}$ with the convergence rate λ_2. QED.

Now we analyze the system with perception error $\hat{q}_{ij} \neq q_{ij}$ following the analysis steps for ultimate boundedness [166]. We define the perception error $\gamma_{ij} \in \mathbb{C}$ as $\gamma_{ij} = \hat{q}_{ij} - q_{ij}$. Further, the perception errors for all pairs of agents is denoted as a $N \times N$ matrix $\Gamma \in \mathbb{C}^{N \times N}$. We now rewrite the controller in Equation (8.5):

$$u_i = \sum_{j \in \{1 \ldots N\}} \omega_{ij}(\hat{q}_{ij} - q^*_{ij}) = \sum_{j \in \{1 \ldots N\}} \omega_{ij}(q_{ij} + \gamma_{ij} - q^*_{ij}) \tag{8.16}$$

Given error state $\mathrm{e_q} = \mathrm{q}^* - \mathrm{q}$, the error dynamics from Equation (8.9) is modified as:

$$\dot{\mathrm{e_q}} = -(\boldsymbol{L}\mathrm{e_q} + (W \odot \Gamma)\vec{\mathbf{1}}) \tag{8.17}$$

In short, $(W \odot \Gamma)\vec{\mathbf{1}}$ represents the perception error induced disturbance. The element-wise multiplication $(W \odot \Gamma)$ states that each pairwise error γ_{ij} is amplified by the weighted edge ω_{ij}. The product of $(W \odot \Gamma)$ and $\vec{\mathbf{1}}$ then aggregates the disturbance for agent i from all

neighbors j. We now show that this system is ultimately bounded.

Lemma 8.2. *If the communication graph is connected, i.e., $\lambda_2 > 0$, and all perception errors are bounded $\|\gamma_{ij}\| \leq \gamma_{\max}$, then the system in Equation 8.17 is* exponentially input-to-state stable *with the convergence rate λ_2, and the error state is ultimately bounded by the sublevel set $\{\mathrm{e_q} \mid V(\mathrm{e_q}) \leq \rho\}$ where*

$$\rho = \frac{N}{2\lambda_2} \cdot (N-1)^2 \cdot \gamma_{\max}^2 \tag{8.18}$$

Proof. From Theorem 8.1, we have already shown the system without noise is exponentially stable with the convergence rate λ_2. Therefore, we focus on deriving the ultimate bound for the Lyapunov function value. Similar to the proof for Theorem 8.1, we can derive

$$\dot{x} = -x + (\tilde{\Lambda}^{\frac{1}{2}})^{-1} P Q^T (W \odot \Gamma)\vec{\mathbf{1}} \tag{8.19}$$

where P is the $(N-1) \times N$ matrix, $[0\, I_{N-1}]$, for removing the first row.

First, because P is removing the first row and Q is orthogonal, we know $\|PQ^T(W \odot \Gamma)\vec{\mathbf{1}}\| \leq \|Q^T(W \odot \Gamma)\vec{\mathbf{1}}\| = \|(W \odot \Gamma)\vec{\mathbf{1}}\|$. We thus derive the bound on $\|(W \odot \Gamma)\vec{\mathbf{1}}\|$. By definition,

$$(W \odot \Gamma)\vec{\mathbf{1}} = \begin{bmatrix} 0 & \omega_{12}\gamma_{12} & \cdots & \omega_{1N}\gamma_{1N} \\ \omega_{21}\gamma_{21} & 0 & \cdots & \omega_{2N}\gamma_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{N1}\gamma_{N1} & \omega_{N2}\gamma_{N2} & \cdots & 0 \end{bmatrix} \vec{\mathbf{1}} = \begin{bmatrix} \sum_{j \neq 1} \omega_{1j}\gamma_{1j} \\ \vdots \\ \sum_{j \neq N} \omega_{Nj}\gamma_{Nj} \end{bmatrix} \tag{8.20}$$

Because $\|\gamma_{ij}\| \leq \gamma_{\max}$ and $\omega_{ij} \in \{0, 1\}$, the norm value is bounded by:

$$\|(W \odot \Gamma)\vec{\mathbf{1}}\| = \sqrt{\sum_i (\sum_{j \neq i} \omega_{ij}\gamma_{ij})^2} \leq \sqrt{N((N-1)\gamma_{\max})^2} = \sqrt{2\lambda_2 \rho} \tag{8.21}$$

Further, by the Input-to-State Stability for linear systems [177], we know when $t \to \infty$,

$$\begin{aligned} \|x(t)\| &\leq e^{-t}\|x(0)\| + \|(\tilde{\Lambda}^{\frac{1}{2}})^{-1}\|\|PQ^T(W \odot \Gamma)\vec{\mathbf{1}}\| \\ &\leq \sqrt{\lambda_2^{-1}} \cdot \sqrt{2\lambda_2\rho} = \sqrt{2\rho} \qquad \text{when } t \to \infty \end{aligned} \tag{8.22}$$

Therefore, the value of the Lyapunov function V is ultimately bounded by

$$V(\mathrm{e_q}(t)) = \tilde{V}(x(t)) = \frac{1}{2}\|x(t)\|^2 \leq \rho \tag{8.23}$$

QED.

Theorem 8.3. *If the communication graph is a* complete graph*, the true relative position q_{ij} of each pair of drones is ultimately bounded around the desired relative position q^*_{ij}. Formally, for all i, j,*

$$\|q_{ij} - q^*_{ij}\| \leq b. \tag{8.24}$$

where $b = \sqrt{\frac{\rho}{N-1}} = \gamma_{\max}\sqrt{\frac{N(N-1)}{2}}$ is the ultimate bound.

Proof. Recall the Lyapunov function at Equation (8.10). When the system enters the ultimate bound $\{\mathrm{e_q} \mid V(\mathrm{e_q}) \leq \rho\}$,

$$V(\mathrm{e_q}) = \frac{1}{2}\sum_{i\in\{1\dots N\}}\sum_{j\in\{1\dots N\}} \omega_{ij}\|q_{ij} - q^*_{ij}\|^2 \leq \rho \tag{8.25}$$

We can see the sum of square for all $\|q_{ij} - q^*_{ij}\|$ is bounded. Without loss of generality, we assume $\|q_{ij} - q^*_{ij}\| = 0$ for $i \neq 1$ and $j \neq 1$ to find the upper bound on $\|q_{1j} - q^*_{1j}\|$. This implies all drones except drone 1 are in the desired formation up to translation, that is, $(q_2 - q^*_2) = \cdots = (q_N - q^*_N) = \Delta q$. We can find for $j \neq 1$

$$\begin{aligned} q_{1j} - q^*_{1j} &= (q_j - q_1) - (q^*_j - q^*_1) = (q_j - q^*_j) - (q_1 - q^*_1) \\ &= \Delta q - (q_1 - q^*_1) \end{aligned} \tag{8.26}$$

For a complete graph, we can simplify $V(\mathrm{e_q})$ as

$$\begin{aligned} V(\mathrm{e_q}) &= \frac{1}{2}(\sum_{j\neq 1}\|q_{1j} - q^*_{1j}\|^2 + \sum_{i\neq 1}\|q_{i1} - q^*_{i1}\|^2) \\ &= \sum_{j\neq 1}\|q_{1j} - q^*_{1j}\|^2 = \sum_{j\neq 1}\|\Delta q - (q_1 - q^*_1)\|^2 \\ &= (N-1)\|\Delta q - (q_1 - q^*_1)\|^2 \leq \rho \end{aligned} \tag{8.27}$$

Hence, $\|q_{1j} - q^*_{1j}\| = \|\Delta q - (q_1 - q^*_1)\| \leq \sqrt{\frac{\rho}{N-1}} = b$ QED.

We can use the ultimate bound from Theorem 8.3 to provide the safety guarantees.

Proposition 8.4. *Given the interaction graphs is a complete graph, all perception errors are bounded $\|\gamma_{ij}\| \leq \gamma_{\max}$, when the system has stabilized to the ultimate bound, the relative distance is both upper and lower bounded. Formally, for all $i, j \in \{1 \dots N\}$ and $i \neq j$*

$$\|\gamma_{ij}\| \leq \gamma_{\max} \implies \|q^*_{ij}\| - b \leq \|q_{ij}\| \leq \|q^*_{ij}\| + b. \tag{8.28}$$

Further, if $\|q_{ij}^*\| > b$, the system stays in the safe set Safe.

$$\|q_{ij}^*\| > b \implies \|q_{ij}\| > 0 \tag{8.29}$$

Proof. The proof is to apply the triangle inequalities, e.g., for the lower bound, $\|q_{ij}^*\| - \|q_{ij}\| \leq \|q_{ij}^* - q_{ij}\| \leq b$. Dually, we can derive the upper bound. QED.

Finally, the above analysis of the ultimate boundedness assumes a bound $\gamma_{\max}$ on maximum perception error for the worst case analysis. As a result, this perception error bound and the derived ultimate bound on distance can be overly conservative if we use the global maximum value regardless of the ground truth. To calculate a more practical bound, our main insight is to use a tighter perception error bound around the desired formation where the ultimate boundedness from Theorem 8.3 holds locally. This ensures that not only the system stays within the neighborhood around the desired formation but also, because it stays in the neighborhood, the perception error is not becoming worse and hence bounded locally at the same time.

Given the desired relative position q_{ij}^* and the approximate abstraction $\gamma_{\max}(q_{ij})$ in Equation (8.7), we linearly search for a pair of d_k and γ_k from $\gamma_{\max}$ such that:

$$\|q_{ij}^*\| + \gamma_k \sqrt{\frac{N(N-1)}{2\lambda_2}} \leq d_k \tag{8.30}$$

If a pair is found, γ_k is the local perception error bound, and $\gamma_k \sqrt{\frac{N(N-1)}{2}}$ is the local ultimate bound on $\|q_{ij}^* - q_{ij}\|$.

Note that we may fail to find any pair of d_k and γ_k from $\gamma_{\max}$ and fall back to the maximum perception error. A common case is when the desired distance $\|q_{ij}^*\|$ is specified in the range where the perception performs poorly. For instance, we can choose $\|q_{ij}^*\|$ over 14 meters, and it is expected there is no good local bound according to Figure 8.4.

8.5 EXPERIMENTS

In our experiments, we aim to validate two claims:

(1) Safe separation and formation are ensured by Theorem 8.3 and Proposition 8.4.

(2) The analysis of ultimate bounds is robust against varying approximate abstractions from different environments.

Experiment Setup We present the results from two experiment settings with three and four drones. For three drones, the drones will be placed initially in a line with 6 meters interval between drones and form an equilateral triangle with edge length 5 meters. For four drones, the drones are placed initially in a line with 4 meters interval and form a square with edge length 5 meters. Then, we simulate each formation for 10 runs in the environment AirSimNH under different environmental parameters by varying the fog level in the scenario. The approximate abstractions are obtained by random sample pairs of images in each of the environment.

We conduct our experiments on a workstation with Intel Core i7-10700K @ 3.80GHz, 32 GB main memory, and Nvidia GTX1080Ti GPU installed with Ubuntu 20.04 LTS, Python 3.8, and OpenCV 4.6.0. We use the Python API for the v1.8.1-Linux version of the AirSim simulator [47].

8.5.1 Safety Assured Formation

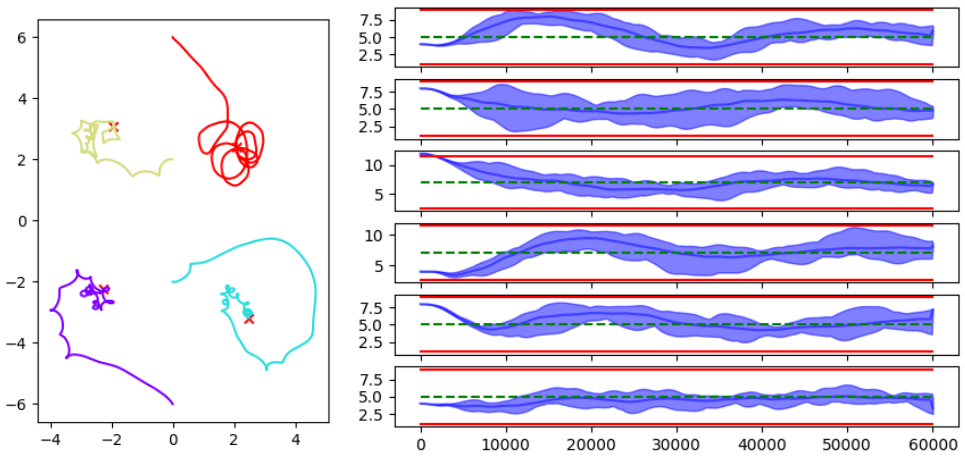


Figure 8.8: Absolute position (*Left*) from one simulation run and relative distance between pairs of drones for 10 simulation runs (*Right*) for four drones in environment AirSimNH with no fog. For the plot on the left, each color represents the absolute position of a drone. For the plot on the right, each row shows true relative distance $\|q_{ij}\|$ (Blue region), desired distance $\|q^*_{ij}\|$ (Green dashed lines), and derived ultimate bounds for safety and closeness using approximate abstractions (Red lines).

We validate our claim that each pair of drones will maintain a relative distance $\|q_{ij}\|$ around the distance $\|q^*_{ij}\|$ specified for the formation. Figure 8.8 shows results for four drones formation. The trajectories of drones are on the left, and true relative distances evolving with time for all six pairs of drones are on the right. The trajectories show the drone still attempting to form the desired shapes under perception error. On the right, we see that the

true relative distance can get really close to the ultimate bound in several simulation runs. Nevertheless, the system never leaves the ultimate bound, and this is consistent with our analysis that entering the ultimate bound ensures formations and safe separation.

8.5.2 Robustness of Ultimate Bound Analysis

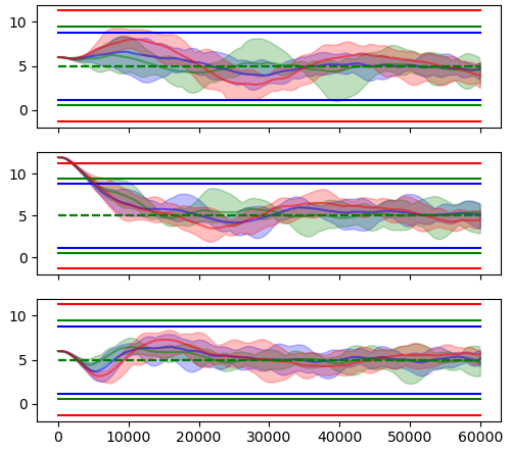


Figure 8.9: Relative distance between three pairs of drones for 10 simulation runs under three different levels of fog (level 0 Blue, level 0.01 Green, and level 0.02 Red). The upper and lower bound on distances is not violated with the change of environmental parameters.

In this section, we look at how robust is our analysis across environmental variations. We vary the weather condition by adding different amounts of fog. We experiment with three different fog levels, 0, 0.01, and 0.02, and derive the approximate abstraction from sampled data under each weather condition. We obtain the ultimate bound and repeat the same simulation. The simulated result is shown in Figure 8.9. We observe that even with the fog introduced, the true relative distance between each pair of drones will still not exceed the ultimate bound derived for the corresponding environment. We also observe that in a foggier environment, where the performance of the vision component worsens, the formation becomes less stable. In the meantime, the ultimate bound becomes more conservative, preventing the relative distance from violating the bound. Note that when the bound becomes larger than the desired relative distance between drones, the lower bound on distances can go below 0. In this case, the premise of Proposition 8.4 is no longer satisfied, and the drones may actually collide.

8.6 DISCUSSIONS AND FUTURE DIRECTIONS

We presented an analysis for the safety of a vision-based formation control system. To tackle the vagaries of the perception component, our approach uses an approximate abstraction. This piecewise constant approximation captures the worst perception error in relative position estimates from the vision component, which is then used to prove that the drones are safely separated and stay close to the desired formation. The analysis uses lower and upper bounds on the distance between agents, and both bounds are derived from the ultimate bound that reuses the Lyapunov-style stability proof. We also systematically studied the impact of environmental variations on the approximate abstraction. Our evaluation with AirSim simulator validates the guarantee on safe separation while achieving formations. We show that the analysis is robust across different environmental variations.

Our study assumed that all drones run synchronously and exchange image feature descriptors instantly. This is obviously an idealization. Our analysis will work without this assumption by bounding the change in relative positions. Under a fixed communication delay, the change in positions can be bounded and modeled as part of the perception error.

The study suggests several future directions. First, our current analysis does not work under arbitrary switching communication graphs. One direction is to view our approximate abstraction, the piecewise constant function, as the quantization of the perception output, and pairs of agents are randomly selected to exchange perception output in each synchronous round. Existing analysis on quantized consensus algorithms can then provide probabilistic guarantees on the convergence rate [178]. Another direction is to explore the latest result on ultimate boundedness for state dependent disturbances [179].

Chapter 9: Conclusions and Future Directions

In this thesis, we presented several algorithms for the verification and simulation of autonomous systems via searching and constructing abstractions for the environment. we studied several autonomous systems including distributed coordination (Chapter 4), unmanned aircraft traffic management (Chapter 5), smart manufacturing (Chapter 6), autonomous vehicle with vision (Chapter 7), and vision-based formation systems (Chapter 8). We showed how a *good* decomposition help formally verify an autonomous system modeled as a reactive module and a model of the environment, and we further demonstrated how to derive statistical guarantees for the model of the environment with respect to the real environment.

For a distributed robotic system using shared variables, we use our formal analysis of our CPReact model and Koord semantics. We identified and decomposed platform-independent coordination logic from platform-dependent assumptions as abstractions. Our formal analyses facilitated inductive invariant checking on the coordination logic and data-driven reachability analysis for the state-space exploration with black-box dynamics. We extended the decomposed analysis and verify the message passing-based protocols for UAS Traffic Management (UTM), and we applied the same decomposed analysis to prove the safety and liveness of UTM protocols. We further investigated distributed systems with agents doing nonidentical tasks and studied smart manufacturing systems as an example. We showed that our CPReact model can capture the smart manufacturing systems, and provided a design and testing method via a parameterized simulation model. Although we did not produce safety verification results for the smart manufacturing systems, we believe it illustrated the path forward to a fully formal verification of smart manufacturing systems and more general distributed robotic systems.

For vision-based perception, our approach creates approximate abstractions for perception (AAP) that are safe by construction. We demonstrated this in the lane tracking control system with the lane detection DNN, LaneNet [146], and the agricultural robot system with CropFollow [156]. Both DNNs contain more than 18 internal layers and over 3 million neurons. To our knowledge, this is the *first* approach capable of handling these practical-sized DNNs for computer vision tasks. To investigate the safety under both complex perception and distributed robotics, we studied the vision-based drone formation system. We followed the same idea to search for AAP that captures the worst perception error in position estimates from vision, which is then used to ensure the safe separation and closeness to the desired formation between drones.

In summary, this thesis is a first step towards formalizing and systematically searching ab-

stractions for vision-based perception, black-box dynamics, and distributed communications in autonomous systems.

9.1 FUTURE RESEARCH DIRECTIONS

The results of this thesis suggest various promising and practical directions for the safety assurance of autonomous systems and general robotic systems.

Component-Level Verification and Monitoring Using Safe Abstractions One practical application of the safe abstractions is to use it for component-level verification and monitoring. For the perception component as an example, we can view the derived safe AAP as a *contract* or a component-level specification. This allows the developers to verify or monitor the perception component against the AAPs in isolation, and this does not involve other components such as software controller and vehicle dynamics. If the component always satisfies the AAP as the contract, it is guaranteed by the definition of AAPs that the system-level safety specification is preserved. Otherwise, the developers can investigate the perception component alone without running the testing or simulation of the entire autonomous system.

For offline analyses such as NN verification and testing, one direction is to reduce the safe abstraction to several local robustness checks for existing NN verification tools. the challenge is still the high dimensional image space. Another direction is to falsify the perception component via property guided fuzz testing. This requires generating images of target scenarios where the perception does not satisfy the contract. Existing image generators for specific application domains, such as Scenic [62] for autonomous vehicles and AirSim [47] for quadrotors, are readily available, but a guided search of images violating a contract remains an interesting research question.

For runtime monitoring, a major technical limitation is the lack of ground-truth information in runtime because our notion of AAPs is based on the ground-truth output of the perception task. To address this limitation, we may assume that multiple components are performing the same perception task, e.g., both GPS and visual positioning can provide positioning service. Runtime monitoring then can be achieved by considering the percept values obtained independently of other components as a noisy ground-truth value. The research question is then to account for noises in runtime monitoring.

Abstraction Synthesis as Specification Mining Problems Following the same view in the previous paragraph, if we consider the safe abstractions as the component-level speci-

fications, we can formulate the abstraction synthesis problem as a specification mining problem. At the time of writing, we have an ongoing collaboration with Prof. Parthasarathy and his PhD student Angello Astorga. We aim to use syntax guided synthesis techniques (SyGuS) [52] to infer safe AAPs that can cover all safe sampled percept values while avoiding overfitting to those samples. We are currently developing a teacher-learner based *exact learning* architecture to solve the SyGuS query based on a previous work for specification mining by Astorga et al. [180]. Our preliminary result shows this architecture can infer more flexible and customized abstractions such as non-convex shapes represented as decision trees or if-then-else expressions.

Simultaneous Invariant and Abstraction Synthesis Across the entire thesis, we have assumed the invariant is available for the abstraction synthesis problem. In Chapter 7, we however observed that the chosen inductive invariant and Lyapunov functions significantly impact the constructed approximation abstractions, and synthesizing the abstraction with respect to a stringent invariant actually can lead to an abstraction with low precision. It is therefore desirable to formalize the synthesis problem combining both the inductive invariant and safe abstraction, and design an efficient synthesis algorithm to search both simultaneously. A related approach is the generalized property-directed reachability analysis for hybrid systems [181], which incrementally infers multiple inductive invariant with respect to the safety property.

Abstractions Against Different System Specifications Finally, we focused on safety specification through the whole thesis and only briefly discussed liveness (progression) specifications in Chapter 5. However, the correctness of autonomous system also relies on liveness or eventuality properties, which indicate progress and self-stabilization, reach-avoid specifications, which can be seen as the conjunction of safety and liveness properties, and more generally variants of temporal logic such as Signal Temporal Logic [182]. Inductive invariant is no longer sufficient to ensure such properties. An interesting direction is then to answer whether the correctness proof can still be divided as independent proofs for the reactive module and the model of environment, and how to define and synthesize property-preserving abstractions.

Appendix A: Stanley Controller for GEM Cart

Table A.1: Description and values of constants for Polaris GEM e2 Electric Cart case study

Symbol	Value	Description
W	4.0	Width of the lane (m)
v_f	2.8	Constant forward velocity (m/s)
l_{WB}	1.75	Wheelbase (m)
ΔT	0.1	Time discretization (s)
δ_{max}	0.61	Steering angle limit (rad)
κ	0.45	Stanley controller gain

Non-increasing Cross Track Distance Following the proof in [145], when in the nominal region $|\psi + \arctan(\frac{\kappa \cdot d}{v_f})| < \delta_{max}$, we derive the system as the ODE in Equation A.1 below:

$$\begin{aligned} \dot{d} &= -v_f \cdot \sin(\arctan(\frac{\kappa \cdot d}{v_f})) \\ &= -\frac{\kappa \cdot d}{\sqrt{1 + (\frac{\kappa \cdot d}{v_f})^2}} \end{aligned} \tag{A.1}$$

Note that $|d|$ converges to zero because $-\frac{\kappa \cdot d}{\sqrt{1+(\frac{\kappa \cdot d}{v_f})^2}}$ is always the opposite sign of d. We can find the Lyapunov function for nominal region $V_2(d, \psi) = |d|$. This is however not entirely true in discrete dynamics because the value $|d|$ can cross zero and become larger in a discrete transition.

Non-increasing Vector Norm Value Following the proof in [145], when in the nominal region $|\psi + \arctan(\frac{\kappa \cdot d}{v_f})| < \delta_{max}$, we derive the system as the ODE in Equation A.2 below:

$$\begin{aligned} \dot{d} &= -\frac{\kappa \cdot d}{\sqrt{1 + (\frac{\kappa \cdot d}{v_f})^2}} \\ \dot{\psi} &= -\frac{v_f \cdot \sin(\psi + \arctan(\frac{\kappa \cdot d}{v_f}))}{l_{WB}} \end{aligned} \tag{A.2}$$

The sign of $\dot{\psi}$ is opposite of $(\psi + \arctan(\frac{\kappa \cdot d}{v_f}))$, so ψ approaches $\arctan(\frac{\kappa \cdot d}{v_f})$. Further, $\arctan(\frac{\kappa \cdot d}{v_f})$ converges to zero because d converges to zero as proven above. Therefore, the origin is the only equilibrium, and the 2D vector norm is non-increasing.

Table A.2: Definitions of tracking error functions.

Tracking error function	Description
$V_1(d,\psi) = \lvert\psi + \arctan(\frac{\kappa \cdot d}{v_f})\rvert$	Combined heading and distance error in [145]
$V_2(d,\psi) = \lvert d\rvert$	Distance error only
$V_3(d,\psi) = \lVert(d,\psi)\rVert$	Vector norm as error

A.1 C CODE ENCODING FOR CBMC

The controllers in Section 7.6 are represented by a mathematical function g. This function is used in the algorithms for computing the safe approximate abstractions. However, the actual implementation of g is in code and the two may have subtle discrepancies. To bridge this gap we verify the controller *code* composed with the computed approximate perception model M and the dynamics using CBMC. Figure A.1 provides the example C code for the controller and the vehicle dynamics of the LTC system in Section 7.6.

```
𝒰 g(𝒵 z) { // Stanley controller example code
  𝒰 δ = z.ψ + atan2(κ*z.d, v_f);
  if(δ >= δ_max)
     δ = δ_max;
  else if(δ <= -δ_max)
     δ = -δ_max;
  return δ;
}
𝒳 f(𝒳 x, 𝒰 δ) { // Bicycle model example code
  𝒳 new_x;
  new_x.x = x.x + v_f*cos(x.θ+δ)*ΔT;
  new_x.y = x.y + v_f*sin(x.θ+δ)*ΔT;
  new_x.θ = x.θ + v_f*sin(δ)/l_WB*ΔT;
  return new_x;
}
```

Figure A.1: Example C code for the vehicle dynamics and Stanley controller of the LTC system.

A.2 VARIATIONS WITH DIFFERENT SYSTEM INVARIANCE

We consider other invariants which uses different tracking error functions listed in Table A.2. V_1 is the original function in [145] to combine the heading and lane deviation as a single tracking error. V_2 considers only lane deviation error (d), and V_3 uses the vector norm as error. Both can be used to prove the same safe set $[\![Safe]\!]$. Three heatmaps for each tracking error function are shown in Figure A.2 for the same three partitions and with the

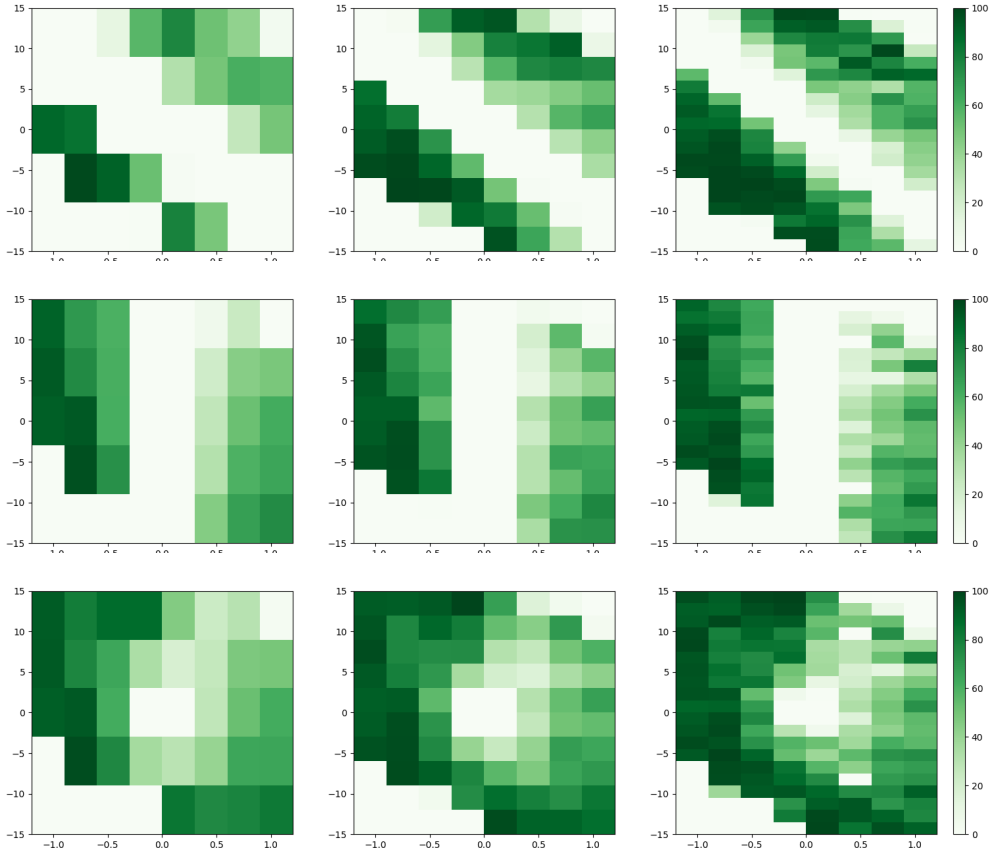


Figure A.2: Precision heatmaps for LaneNet with Stanley controller with only two-lane road for three error tracking functions V_1 (*Top*) vs V_2 (*Mid*) vs V_3 (*Bottom*).

testing set with two-lane road.

Low Precision Regions Shaped by Tracking Error Functions Along the diagonal line (through the origin) we have states where the vehicle's deviation from the lane center d and the heading ψ are in opposing direction. By observing V_1 from Table A.2, we know ψ and d are of opposite signs at the equilibrium points $V_1(d, \psi)$. Hence, the band of white cells goes from the second to the fourth quadrant. Therefore, the tracking error cannot be non-increasing in these states in one step as required by *Inv*.

By comparing heatmaps in Figures A.2, we see a white band surrounding the line $d = 0$ for V_2 and a white spot around the origin for V_3. This validates our explanation that the AAP breaks owing to the stringent requirement of non-increasing error.

Appendix B: Modified Stanley Controller with Farm Robots

Table B.1: Description and values of constant Symbols for the agricultural robot case study.

Symbol	Value	Description
W	0.76	Width of the corn row (m)
v_f	1.0	Constant forward velocity (m/s)
ΔT	0.05	Time discretization (s)
ω_{max}	0.5	Angular velocity limit (rad/s)
κ	0.1	Stanley controller gain

We provide the vehicle dynamics and the controller for the AgBot case study in Section 7.7. The dynamics $f(x, u)$ from [156] is given as Equation (B.1) below:

$$\begin{aligned} x_{t+1} &= x_t + v_f \cos(\theta_t)\Delta T \\ y_{t+1} &= y_t + v_f \sin(\theta_t)\Delta T \\ \theta_{t+1} &= \theta_t + \omega \Delta T \end{aligned} \tag{B.1}$$

The controller g is given as Equation (B.2) below:

$$g(d,\psi) = \begin{cases} \frac{\psi + \arctan\left(\frac{\kappa \cdot d}{v_f}\right)}{\Delta T}, & \text{if } \left|\psi + \arctan\left(\frac{\kappa \cdot d}{v_f}\right)\right| < \omega_{max} \cdot \Delta T \\ \omega_{max}, & \text{if } \psi + \arctan\left(\frac{\kappa \cdot d}{v_f}\right) \geq \omega_{max} \cdot \Delta T \\ -\omega_{max}, & \text{if } \psi + \arctan\left(\frac{\kappa \cdot d}{v_f}\right) \leq -\omega_{max} \cdot \Delta T \end{cases} \tag{B.2}$$

To cover the invariant and stay with in the safe set $[\![Safe]\!]$, we choose the whole space $\bigcup_{i=1}^{N} \mathcal{X}_i$ covers $\pm 0.3W = \pm 0.228$ meters in y and $\pm 30^\circ$ in θ. The error function $V_1(d, \psi) = |\psi + \arctan(\frac{\kappa \cdot d}{v_f})|$ in A.2 is used for this system.

We consider three different partitions $N \in \{5 \times 5, 10 \times 10, 20 \times 20\}$. We follow the same procedure to sample images and derive the safe neighbor function $\mathcal{R}_i$ for $\mathcal{X}_i$. For this case study, the environment parameter space $\mathcal{E}$ is defined by five different plant fields, including three stages of corns (baby, small, and adult) and two stages of tobaccos (early and late). We use the uniform distribution over the state space $\mathcal{X}_i$ and the five environments for both the training testing set.

The precision heatmaps in Figure B.1 shows exactly the same trend as the top row in Figure A.2 for V_1 in the GEM cart case study.

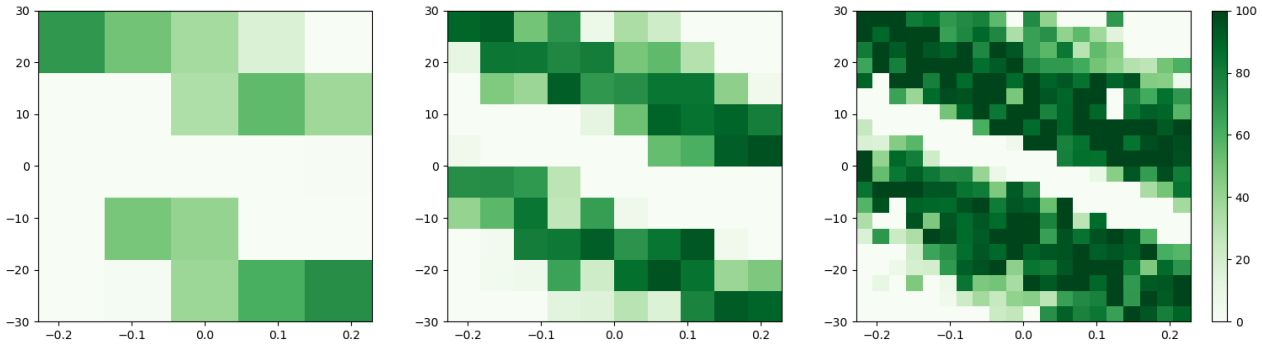


Figure B.1: Precision heatmap of AAPs inferred for CropFollow using $N \in \{5\times5, 10\times10, 20\times20\}$.

References

[1] J. N. Pires and J. M. G. Sá da Costa, "Object-oriented and distributed approach for programming robotic manufacturing cells," *Robotics and Computer-Integrated Manufacturing*, vol. 16, no. 1, pp. 29–42, Feb. 2000.

[2] T. Blender, T. Buchner, B. Fernandez, B. Pichlmaier, and C. Schlegel, "Managing a Mobile Agricultural Robot Swarm for a seeding task," in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, Oct. 2016, pp. 6879–6886.

[3] R. R. Shamshiri, C. Weltzien, I. A. Hameed, I. J. Yule, T. E. Grift, S. K. Balasundram, L. Pitonakova, D. Ahmad, and G. Chowdhary, "Research and development in agricultural robotics: A perspective of digital farming," *International Journal of Agricultural and Biological Engineering*, vol. 11, no. 4, pp. 1–14, Aug. 2018.

[4] P. J. Mosterman, D. Escobar Sanabria, E. Bilgin, K. Zhang, and J. Zander, "Automating humanitarian missions with a heterogeneous fleet of vehicles," *Annual Reviews in Control*, vol. 38, no. 2, pp. 259–270, Jan. 2014.

[5] G. Guo and W. Yue, "Autonomous Platoon Control Allowing Range-Limited Sensors," *IEEE Transactions on Vehicular Technology*, vol. 61, no. 7, pp. 2901–2912, Sep. 2012.

[6] M. Gerla, E.-K. Lee, G. Pau, and U. Lee, "Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Mar. 2014, pp. 241–246.

[7] Federal Aviation Administration, "FAA Aerospace Forecast Fiscal Year 2022-2042," 2022. [Online]. Available: https://www.faa.gov/sites/faa.gov/files/2022-06/FY2022_42_FAA_Aerospace_Forecast.pdf

[8] White House Office of Science and Technology Policy, "Blueprint for an AI Bill of Rights," Oct. 2022. [Online]. Available: https://www.whitehouse.gov/wp-content/uploads/2022/10/Blueprint-for-an-AI-Bill-of-Rights.pdf

[9] Future of Life Institute, "Pause Giant AI Experiments: An Open Letter," Mar. 2023. [Online]. Available: https://futureoflife.org/open-letter/pause-giant-ai-experiments/

[10] EU Aviation Safety Agency, "EASA Concept Paper First usable guidance for Level 1 machine learning applications," 2021. [Online]. Available: https://www.easa.europa.eu/sites/default/files/dfu/easa_concept_paper_first_usable_guidance_for_level_1_machine_learning_applications_-_proposed_issue_01_1.pdf

[11] P. Koopman, U. Ferrell, F. Fratrik, and M. Wagner, "A Safety Standard Approach for Fully Autonomous Vehicles," in *Proceedings of the 2nd International Workshop on Artificial Intelligence Safety Engineering (WAISE 2019)*, ser. Lecture Notes in Computer Science, vol. 11699. Cham: Springer International Publishing, 2019, pp. 326–332.

[12] Federal Aviation Administration, "Unmanned Aircraft System Traffic Management (UTM) Concept of Operations Version 2.0," Aug. 2022. [Online]. Available: https://www.faa.gov/sites/faa.gov/files/2022-08/UTM_ConOps_v2.pdf

[13] N. Kalra and S. M. Paddock, "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?" *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, Dec. 2016.

[14] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, no. 2, pp. 123–193, Apr. 1999.

[15] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, pp. 19:1–19:36, Oct. 2009.

[16] R. Alur and T. A. Henzinger, "Reactive Modules," *Formal Methods in System Design*, vol. 15, no. 1, pp. 7–48, July 1999.

[17] R. Alur, R. Grosu, I. Lee, and O. Sokolsky, "Compositional Refinement for Hierarchical Hybrid Systems," ser. HSCC '01. London, UK, UK: Springer-Verlag, 2001, pp. 33–48.

[18] C. S. Păsăreanu, D. Gopinath, and H. Yu, "Compositional Verification for Autonomous Systems with Deep Learning Components," in *Safe, Autonomous and Intelligent Vehicles*, ser. Unmanned System Technologies. Cham: Springer International Publishing, 2019, pp. 187–197.

[19] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[20] N. Lynch, R. Segala, and F. Vaandrager, "Hybrid I/O Automata," *Inf. Comput.*, vol. 185, no. 1, pp. 105–157, Aug. 2003.

[21] A. Donzé and G. Frehse, "Modular, hierarchical models of control systems in SpaceEx," in *2013 European Control Conference (ECC)*, July 2013, pp. 4244–4251.

[22] S. Bak and S. Chaki, "Verifying cyber-physical systems by combining software model checking with hybrid systems reachability," in *2016 International Conference on Embedded Software (EMSOFT)*, Oct. 2016, pp. 1–10.

[23] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, "DryVR: Data-Driven Verification and Compositional Reasoning for Automotive Systems," in *International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer, Cham, July 2017, pp. 441–461.

[24] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems*," *European Journal of Control*, vol. 18, no. 3, pp. 217–238, Jan. 2012.

[25] P. Koopman and M. Wagner, "Autonomous Vehicle Safety: An Interdisciplinary Challenge," *IEEE Intelligent Transportation Systems Magazine*, vol. 9, pp. 90–96, Mar. 2017.

[26] S. A. Seshia, D. Sadigh, and S. S. Sastry, "Toward verified artificial intelligence," *Commun. ACM*, vol. 65, no. 7, pp. 46–55, June 2022.

[27] T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. A. Seshia, "VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 432–442.

[28] S. M. Katz, A. L. Corso, C. A. Strong, and M. J. Kochenderfer, "Verification of Image-based Neural Network Controllers Using Generative Models," in *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, Oct. 2021, pp. 1–10.

[29] U. Santa Cruz and Y. Shoukry, "NNLander-VeriF: A Neural Network Formal Verification Framework for Vision-Based Autonomous Aircraft Landing," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 213–230.

[30] C. Hsieh, Y. Li, D. Sun, K. Joshi, S. Misailovic, and S. Mitra, "Verifying Controllers With Vision-Based Perception Using Safe Approximate Abstractions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4205–4216, Nov. 2022.

[31] W. Howden, "Theoretical and Empirical Studies of Program Testing," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 4, pp. 293–298, July 1978.

[32] R. Ghosh, C. Hsieh, S. Misailovic, and S. Mitra, "Koord: A language for programming and verifying distributed robotics application," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 232:1–232:30, Nov. 2020.

[33] R. Ghosh, "Separation of distributed coordination and control for programming reliable robotics," Ph.D. dissertation, University of Illinois at Urbana-Champaign, July 2020. [Online]. Available: http://hdl.handle.net/2142/108501

[34] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *Proceedings of the 2nd International Conference on Learning Representations (ICLR 2014)*, Feb. 2014. [Online]. Available: http://arxiv.org/abs/1312.6199

[35] J. Lu, H. Sibai, E. Fabry, and D. Forsyth, "NO Need to Worry about Adversarial Examples in Object Detection in Autonomous Vehicles," *arXiv:1707.03501 [cs]*, July 2017. [Online]. Available: http://arxiv.org/abs/1707.03501

[36] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," Apr. 2015.

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[38] S. A. Seshia, "Compositional verification without compositional specification for learning-based systems," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-164, Nov. 2017. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-164.html

[39] D. Harel and A. Pnueli, "On the Development of Reactive Systems," in *Logics and Models of Concurrent Systems*, ser. NATO ASI Series. Berlin, Heidelberg: Springer, 1985, pp. 477–498.

[40] O. Kupferman and M. Y. Vardi, "Verification of Open Systems," in *Interactive Computation: The New Paradigm.* Berlin, Heidelberg: Springer, 2006, pp. 97–117.

[41] C. Hsieh, H. Sibai, H. Taylor, Y. Ni, and S. Mitra, "SkyTrakx: A Toolkit for Simulation and Verification of Unmanned Air-Traffic Management Systems," in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, Sep. 2021, pp. 372–379.

[42] J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. von Stryk, "Comprehensive Simulation of Quadrotor UAVs Using ROS and Gazebo," in *Proceedings of the 3rd International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2012)*, ser. Lecture Notes in Computer Science, vol. 7628. Berlin, Heidelberg: Springer, 2012, pp. 400–411.

[43] 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)*, Mar. 2017, pp. 195–203.

[44] G. Ellingson and T. McLain, "ROSplane: Fixed-wing autopilot for education and research," in *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, June 2017, pp. 1503–1507.

[45] A. Downs, Z. Kootbally, W. Harrison, P. Pilliptchak, B. Antonishek, M. Aksu, C. Schlenoff, and S. K. Gupta, "Assessing Industrial Robot Agility through International Competitions," *Robotics and Computer-Integrated Manufacturing*, vol. 70, p. 102113, Aug. 2021.

[46] P. Du, Z. Huang, T. Liu, T. Ji, K. Xu, Q. Gao, H. Sibai, K. Driggs-Campbell, and S. Mitra, "Online Monitoring for Safe Pedestrian-Vehicle Interactions," in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, Sep. 2020, pp. 1–8.

[47] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," in *Field and Service Robotics*, ser. Springer Proceedings in Advanced Robotics. Cham: Springer International Publishing, 2018, pp. 621–635.

[48] H. Araujo, M. R. Mousavi, and M. Varshosaz, "Testing, Validation, and Verification of Robotic and Autonomous Systems: A Systematic Review," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, pp. 51:1–51:61, Mar. 2023.

[49] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede, "A Survey on Domain-specific Modeling and Languages in Robotics," *Journal of Software Engineering for Robotics*, 2016.

[50] S. A. Seshia and P. Subramanyan, "UCLID5: Integrating Modeling, Verification, Synthesis and Learning," in *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, Oct. 2018, pp. 1–10.

[51] E. Polgreen, K. Cheang, P. Gaddamadugu, A. Godbole, K. Laeufer, S. Lin, Y. A. Manerkar, F. Mora, and S. A. Seshia, "UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 538–551.

[52] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *2013 Formal Methods in Computer-Aided Design*, Oct. 2013, pp. 1–8.

[53] A. Desai, I. Saha, J. Yang, S. Qadeer, and S. A. Seshia, "DRONA: A Framework for Safe Distributed Mobile Robotics," in *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ser. ICCPS '17. New York, NY, USA: ACM, 2017, pp. 239–248.

[54] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: Safe asynchronous event-driven programming," *SIGPLAN Not.*, vol. 48, no. 6, pp. 321–332, June 2013.

[55] R. Ghosh, J. P. Jansch-Porto, C. Hsieh, A. Gosse, M. Jiang, H. Taylor, P. Du, S. Mitra, and G. Dullerud, "CyPhyHouse: A programming, simulation, and deployment toolchain for heterogeneous distributed coordination," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, May 2020, pp. 6654–6660.

[56] B. Bohrer, Y. K. Tan, S. Mitsch, M. O. Myreen, and A. Platzer, "VeriPhy: Verified controller executables from verified cyber-physical system models," *SIGPLAN Not.*, vol. 53, no. 4, pp. 617–630, June 2018.

[57] B. Bohrer, V. Rahli, I. Vukotic, M. Völp, and A. Platzer, "Formally verified differential dynamic logic," in *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, ser. CPP 2017. New York, NY, USA: Association for Computing Machinery, Jan. 2017, pp. 208–221.

[58] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer, "KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems," in *Automated Deduction - CADE-25*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 527–538.

[59] S. Ghosh, Y. V. Pant, H. Ravanbakhsh, and S. A. Seshia, "Counterexample-Guided Synthesis of Perception Models and Control," in *2021 American Control Conference (ACC)*, May 2021, pp. 3447–3454.

[60] C. S. Păsăreanu, R. Mangal, D. Gopinath, S. G. Yaman, C. Imrie, R. Calinescu, and H. Yu, "Closed-loop Analysis of Vision-based Autonomous Systems: A Case Study," Feb. 2023.

[61] D. J. Fremont, J. Chiu, D. D. Margineantu, D. Osipychev, and S. A. Seshia, "Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VerifAI," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 122–134.

[62] D. J. Fremont, E. Kim, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: A language for scenario specification and data generation," *Mach Learn*, Feb. 2022.

[63] Y. Ma, S. Soatto, J. Kosecka, and S. S. Sastry, *An Invitation to 3-D Vision: From Images to Geometric Models*, 1st ed., ser. Interdisciplinary Applied Mathematics. SpringerVerlag, 2003.

[64] H.-D. Tran, X. Yang, D. Manzanas Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson, "NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 3–17.

[65] R. Ivanov, T. J. Carpenter, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verifying the Safety of Autonomous Systems with Neural Network Controllers," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 1, pp. 7:1–7:26, Dec. 2020.

[66] K. D. Julian and M. J. Kochenderfer, "Guaranteeing Safety for Neural Network-Based Aircraft Collision Avoidance Systems," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, Sep. 2019, pp. 1–10.

[67] K. D. Julian and M. J. Kochenderfer, "Reachability Analysis for Neural Network Aircraft Collision Avoidance Systems," *Journal of Guidance, Control, and Dynamics*, vol. 44, no. 6, pp. 1132–1142, June 2021.

[68] S. Dutta, X. Chen, and S. Sankaranarayanan, "Reachability analysis for neural feedback systems using regressive polynomial rule inference," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 157–168.

[69] C. Huang, J. Fan, W. Li, X. Chen, and Q. Zhu, "ReachNN: Reachability Analysis of Neural-Network Controlled Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, pp. 106:1–106:22, Oct. 2019.

[70] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verisig: Verifying safety properties of hybrid systems with neural network controllers," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 169–178.

[71] H. Hu, M. Fazlyab, M. Morari, and G. J. Pappas, "Reach-SDP: Reachability Analysis of Closed-Loop Systems with Neural Network Controllers via Semidefinite Programming," in *2020 59th IEEE Conference on Decision and Control (CDC)*, Dec. 2020, pp. 5929–5934.

[72] M. Everett, G. Habibi, C. Sun, and J. P. How, "Reachability Analysis of Neural Feedback Loops," *IEEE Access*, vol. 9, pp. 163 938–163 953, 2021.

[73] T. P. Gros, H. Hermanns, J. Hoffmann, M. Klauck, and M. Steinmetz, "Deep Statistical Model Checking," in *Formal Techniques for Distributed Objects, Components, and Systems*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 96–114.

[74] R. Ivanov, K. Jothimurugan, S. Hsu, S. Vaidya, R. Alur, and O. Bastani, "Compositional Learning and Verification of Neural Network Controllers," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, pp. 92:1–92:26, Sep. 2021.

[75] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 97–117.

[76] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 3–18.

[77] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 41:1–41:30, Jan. 2019.

[78] M. N. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev, "PRIMA: General and precise neural network certification via scalable convex hull approximations," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 43:1–43:33, Jan. 2022.

[79] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Z. Kolter, "Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Neural Network Robustness Verification," in *Advances in Neural Information Processing Systems*, vol. 34. Curran Associates, Inc., 2021. [Online]. Available: https://proceedings.neurips.cc/paper/2021/hash/fac7fead96dafceaf80c1daffeae82a4-Abstract.html pp. 29 909–29 921.

[80] K. Xu, H. Zhang, S. Wang, Y. Wang, S. Jana, X. Lin, and C.-J. Hsieh, "Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=nVZtXBI6LNn

[81] S. Bak, C. Liu, and T. Johnson, "The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results," *arXiv:2109.00498 [cs]*, Aug. 2021. [Online]. Available: http://arxiv.org/abs/2109.00498

[82] M. N. Müller, C. Brix, S. Bak, C. Liu, and T. T. Johnson, "The Third International Verification of Neural Networks Competition (VNN-COMP 2022): Summary and Results," Feb. 2023.

[83] R. Alur, *Principles of Cyber-Physical Systems*. The MIT Press, 2015.

[84] S. Mitra, *Verifying Cyber-Physical Systems: A Path to Safe Autonomy*, ser. Cyber Physical Systems Series. Cambridge, MA, USA: MIT Press, Feb. 2021.

[85] C. Barrett and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Model Checking*. Cham: Springer International Publishing, 2018, pp. 305–343.

[86] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 337–340.

[87] A. Coddington and N. Levinson, *Theory of Ordinary Differential Equations*, ser. International Series in Pure and Applied Mathematics. McGraw-Hill, 1955. [Online]. Available: https://books.google.com/books?id=bPJQAAAAMAAJ

[88] A. Platzer, *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[89] S. Bak and P. S. 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*, ser. HSCC '17. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 173–178.

[90] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable Verification of Hybrid Systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 379–395.

[91] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An Analyzer for Non-linear Hybrid Systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 258–263.

[92] P. S. Duggirala, S. Mitra, and M. Viswanathan, "Verification of annotated models from executions," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, Sep. 2013, pp. 1–10.

[93] C. Fan, B. Qi, S. Mitra, M. Viswanathan, and P. S. Duggirala, "Automatic Reachability Analysis for Nonlinear Hybrid Models with C2E2," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 531–538.

[94] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's Decidable about Hybrid Automata?" *Journal of Computer and System Sciences*, vol. 57, no. 1, pp. 94–124, Aug. 1998.

[95] A. A. Kurzhanskiy and P. Varaiya, "Ellipsoidal Techniques for Reachability Analysis of Discrete-Time Linear Systems," *IEEE Transactions on Automatic Control*, vol. 52, no. 1, pp. 26–38, Jan. 2007.

[96] A. Girard, "Reachability of Uncertain Linear Systems Using Zonotopes," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 291–305.

[97] C. Fan, B. Qi, and S. Mitra, "Data-Driven Formal Reasoning and Their Applications in Safety Analysis of Vehicle Autonomy Features," *IEEE Design & Test*, vol. 35, no. 3, pp. 31–38, June 2018.

[98] G. Russo and J.-J. E. Slotine, "Symmetries, stability, and control in nonlinear systems and networks," *Phys. Rev. E*, vol. 84, no. 4, p. 041929, Oct. 2011.

[99] H. Sibai, N. Mokhlesi, C. Fan, and S. Mitra, "Multi-agent Safety Verification Using Symmetry Transformations," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 173–190.

[100] G. Roșu and T. F. Șerbănuță, "An overview of the K semantic framework," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, Aug. 2010.

[101] Federal Aviation Administration, "FAA Aerospace Forecast Fiscal Year 2020-2040," 2020. [Online]. Available: https://www.faa.gov/sites/faa.gov/files/2022-06/FY2022_42_FAA_Aerospace_Forecast.pdf

[102] Federal Aviation Administration, "UTM Pilot Program (UPP) Summary Report," Oct. 2019. [Online]. Available: https://www.faa.gov/uas/research_development/traffic_management/utm_pilot_program/media/UPP_Technical_Summary_Report_Final.pdf

[103] C. Hsieh and S. Mitra, "Dione: A Protocol Verification System Built with Dafny for I/O Automata," in *Proceedings of the 15th International Conference on Integrated Formal Methods (IFM 2019)*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 227–245.

[104] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2010)*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 348–370.

[105] Federal Aviation Administration, "Introduction to TCAS II version 7.1," Feb. 2011. [Online]. Available: https://www.faa.gov/documentlibrary/media/advisory_circular/tcas%20ii%20v7.1%20intro%20booklet.pdf

[106] M. J. Kochenderfer, J. E. Holland, and J. P. Chryssanthacopoulos, "Next-Generation Airborne Collision Avoidance System," Massachusetts Institute of Technology Lincoln Laboratory, Tech. Rep., 2012. [Online]. Available: https://apps.dtic.mil/sti/citations/AD1014875

[107] M. J. Kochenderfer, C. Amato, G. Chowdhary, J. P. How, H. J. D. Reynolds, J. R. Thornton, P. A. Torres-Carrasquillo, N. K. Ure, and J. Vian, "Optimized airborne collision avoidance," in *Decision Making under Uncertainty: Theory and Application*, 2015, pp. 249–276. [Online]. Available: https://ieeexplore.ieee.org/document/7288641

[108] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy compression for aircraft collision avoidance systems," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, Sep. 2016, pp. 1–10.

[109] J. K. Kuchar and L. C. Yang, "A review of conflict detection and resolution modeling methods," *Trans. Intell. Transport. Syst.*, vol. 1, no. 4, pp. 179–189, Dec. 2000.

[110] X. Yu and Y. Zhang, "Sense and avoid technologies with applications to unmanned aircraft systems: Review and prospects," *Progress in Aerospace Sciences*, vol. 74, pp. 152–166, Apr. 2015.

[111] J. Lygeros and N. Lynch, "On the formal verification of the TCAS conflict resolution algorithms," in *Proceedings of the 36th IEEE Conference on Decision and Control*, vol. 2, Dec. 1997, pp. 1829–1834 vol.2.

[112] C. Livadas, J. Lygeros, and N. Lynch, "High-level modeling and analysis of TCAS," in *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*, Dec. 1999, pp. 115–125.

[113] N. A. Lynch, "High-Level Modeling and Analysis of an Air-Traffic Management System (Abstract)," in *Proceedings of the Second International Workshop on Hybrid Systems: Computation and Control*, ser. HSCC '99. Berlin, Heidelberg: Springer-Verlag, Mar. 1999, p. 3.

[114] C. Livadas, J. Lygeros, and N. Lynch, "High-level modeling and analysis of the traffic alert and collision avoidance system (TCAS)," *Proceedings of the IEEE*, vol. 88, no. 7, pp. 926–948, July 2000.

[115] J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, R. Gardner, A. Schmidt, E. Zawadzki, and A. Platzer, "Formal verification of ACAS X, an industrial airborne collision avoidance system," in *2015 International Conference on Embedded Software (EMSOFT)*, Oct. 2015, pp. 127–136.

[116] J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, R. Gardner, A. Schmidt, E. Zawadzki, and A. Platzer, "A Formally Verified Hybrid System for the Next-Generation Airborne Collision Avoidance System," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2015, pp. 21–36.

[117] T. T. Johnson and S. Mitra, "A Small Model Theorem for Rectangular Hybrid Automata Networks," in *Formal Techniques for Distributed Systems*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 18–34.

[118] P. S. Duggirala, L. Wang, S. Mitra, M. Viswanathan, and C. Muñoz, "Temporal Precedence Checking for Switched Models and Its Application to a Parallel Landing Protocol," in *FM 2014: Formal Methods*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 215–229.

[119] H.-D. Tran, L. V. Nguyen, P. Musau, W. Xiang, and T. T. Johnson, "Decentralized Real-Time Safety Verification for Distributed Cyber-Physical Systems," in *Formal Techniques for Distributed Objects, Components, and Systems*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 261–277.

[120] M. Webster, M. Fisher, N. Cameron, and M. Jump, "Formal Methods for the Certification of Autonomous Unmanned Aircraft Systems," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 228–242.

[121] O. McAree, J. M. Aitken, and S. M. Veres, "A model based design framework for safety verification of a semi-autonomous inspection drone," in *2016 UKACC 11th International Conference on Control (CONTROL)*, Aug. 2016, pp. 1–6.

[122] S. Umeno and N. Lynch, "Safety Verification of an Aircraft Landing Protocol: A Refinement Approach," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 557–572.

[123] Y. V. Pant, H. Abbas, R. A. Quaye, and R. Mangharam, "Fly-by-Logic: Control of Multi-Drone Fleets with Temporal Logic Objectives," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, Apr. 2018, pp. 186–197.

[124] T. Schouwenaars, "Safe trajectory planning of autonomous vehicles," Thesis, Massachusetts Institute of Technology, 2006. [Online]. Available: https://dspace.mit.edu/handle/1721.1/35298

[125] S. Bharadwaj, S. Carr, N. Neogi, H. Poonawala, A. B. Chueca, and U. Topcu, "Traffic Management for Urban Air Mobility," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 71–87.

[126] S. Bharadwaj, S. Carr, N. Neogi, and U. Topcu, "Decentralized Control Synthesis for Air Traffic Management in Urban Air Mobility," *IEEE Transactions on Control of Network Systems*, vol. 8, no. 2, pp. 598–608, June 2021.

[127] C. Hsieh, D. Wu, Y. Koh, and S. Mitra, "Programming Abstractions for Simulation and Testing on Smart Manufacturing Systems," in *2022 IEEE 18th International Conference on Automation Science and Engineering (CASE)*, Aug. 2022, pp. 2287–2292.

[128] S. Süß, S. Magnus, M. Thron, H. Zipper, U. Odefey, V. Fäßler, A. Strahilov, A. Kłodowski, T. Bär, and C. Diedrich, "Test methodology for virtual commissioning based on behaviour simulation of production systems," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2016, pp. 1–9.

[129] P. Hoffmann, R. Schumann, T. M. A. Maksoud, and G. C. Premier, "Virtual Commissioning Of Manufacturing Systems A Review And New Approaches For Simplification," in *European Conference on Modelling and Simulation, ECMS 2010, Kuala Lumpur, Malaysia, June 1-4, 2010*. European Council for Modeling and Simulation, 2010, pp. 175–181.

[130] C. G. Lee and S. C. Park, "Survey on the virtual commissioning of manufacturing systems," *Journal of Computational Design and Engineering*, vol. 1, no. 3, pp. 213–222, July 2014.

[131] K. An, A. Trewyn, A. Gokhale, and S. Sastry, "Model-Driven Performance Analysis of Reconfigurable Conveyor Systems Used in Material Handling Applications," in *2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, Apr. 2011, pp. 141–150.

[132] F. Lopez, Y. Shao, Z. M. Mao, J. Moyne, K. Barton, and D. Tilbury, "A software-defined framework for the integrated management of smart manufacturing systems," *Manufacturing Letters*, vol. 15, pp. 18–21, Jan. 2018.

[133] E. C. Balta, D. M. Tilbury, and K. Barton, "A Centralized Framework for System-Level Control and Management of Additive Manufacturing Fleets," in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, Aug. 2018, pp. 1071–1078.

[134] Y. Qamsane, C.-Y. Chen, E. C. Balta, B.-C. Kao, S. Mohan, J. Moyne, D. Tilbury, and K. Barton, "A Unified Digital Twin Framework for Real-time Monitoring and Evaluation of Smart Manufacturing Systems," in *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, Aug. 2019, pp. 1394–1401.

[135] M. Potok, C.-Y. Chen, S. Mitra, and S. Mohan, "SDCWorks: A Formal Framework for Software Defined Control of Smart Manufacturing Systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, Apr. 2018, pp. 88–97.

[136] "ARIAC repository for CyPhyHouse," 2021. [Online]. Available: https://github.com/cyphyhouse/ARIAC

[137] D. Jin, P. O. Meredith, C. Lee, and G. Roşu, "JavaMOP: Efficient parametric runtime monitoring framework," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 1427–1430.

[138] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. Sipma, S. Mehrotra, and Z. Manna, "LOLA: Runtime monitoring of synchronous systems," in *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, June 2005, pp. 166–174.

[139] A. Adadi and M. Berrada, "Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)," *IEEE Access*, vol. 6, pp. 52 138–52 160, 2018.

[140] C. Baier, C. Dubslaff, F. Funke, S. Jantsch, R. Majumdar, J. Piribauer, and R. Ziemek, "From Verification to Causality-Based Explications," in *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 198. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 1:1–1:20.

[141] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Proceedings of the 10th Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, vol. 2988. Berlin, Heidelberg: Springer, 2004, pp. 168–176.

[142] G. Brat, J. A. Navas, N. Shi, and A. Venet, "IKOS: A Framework for Static Analysis Based on Abstract Interpretation," in *Proceedings of the 12th International Conference on Software Engineering and Formal Methods (SEFM 2014)*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 271–277.

[143] P. Koopman and F. Fratrik, "How Many Operational Design Domains, Objects, and Events?" in *Proceedings of the AAAI Workshop on Artificial Intelligence Safety 2019*, ser. CEUR Workshop Proceedings, vol. 2301. CEUR-WS.org, 2019. [Online]. Available: http://ceur-ws.org/Vol-2301/paper_6.pdf p. 4.

[144] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 33–55, Mar. 2016.

[145] G. M. Hoffmann, C. J. Tomlin, M. Montemerlo, and S. Thrun, "Autonomous Automobile Trajectory Tracking for Off-Road Driving: Controller Design, Experimental Validation and Racing," in *Proceedings of the 2007 American Control Conference (ACC 2007)*, July 2007, pp. 2296–2301.

[146] D. Neven, B. D. Brabandere, S. Georgoulis, M. Proesmans, and L. V. Gool, "Towards End-to-End Lane Detection: An Instance Segmentation Approach," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, June 2018, pp. 286–291.

[147] S. Prajna and A. Jadbabaie, "Safety Verification of Hybrid Systems Using Barrier Certificates," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 477–492.

[148] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Constructing invariants for hybrid systems," *Form Methods Syst Des*, vol. 32, no. 1, pp. 25–55, Feb. 2008.

[149] A. Platzer and E. M. Clarke, "Computing Differential Invariants of Hybrid Systems as Fixedpoints," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 176–189.

[150] Gurobi Optimization, LLC, "Gurobi optimizer reference manual," 2022. [Online]. Available: https://www.gurobi.com

[151] M. Fähndrich, "Static Verification for Code Contracts," in *Proceedings of the 17th International Static Analysis Symposium (SAS 2010)*, ser. Lecture Notes in Computer Science, vol. 6337. Berlin, Heidelberg: Springer, 2010, pp. 2–5.

[152] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for java," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 1–38, May 2006.

[153] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview," in *Proceedings of the 2004 International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, ser. Lecture Notes in Computer Science, vol. 3362. Berlin, Heidelberg: Springer, 2004, pp. 49–69.

[154] J. LaSalle, "Some Extensions of Liapunov's Second Method," *IRE Transactions on Circuit Theory*, vol. 7, no. 4, pp. 520–527, Dec. 1960.

[155] R. Brockett and D. Liberzon, "Quantized feedback stabilization of linear systems," *IEEE Transactions on Automatic Control*, vol. 45, no. 7, pp. 1279–1289, July 2000.

[156] A. N. Sivakumar, S. Modi, M. V. Gasparino, C. Ellis, A. E. B. Velasquez, G. Chowdhary, and S. Gupta, "Learned Visual Navigation for Under-Canopy Agricultural Robots," in *Robotics: Science and Systems XVII*, vol. 17, July 2021.

[157] C. Hsieh, Y. Li, Y. Koh, and S. Mitra, "Assuring safety of vision-based swarm formation control," Oct. 2022.

[158] V. Blondel, J. Hendrickx, A. Olshevsky, and J. Tsitsiklis, "Convergence in Multiagent Coordination, Consensus, and Flocking," in *Proceedings of the 44th IEEE Conference on Decision and Control*, Dec. 2005, pp. 2996–3000.

[159] R. Saber and R. Murray, "Flocking with obstacle avoidance: Cooperation with limited communication in mobile networks," in *42nd IEEE International Conference on Decision and Control (IEEE Cat. No.03CH37475)*, vol. 2, Dec. 2003, pp. 2022–2028 Vol.2.

[160] M. Mesbahi and M. Egerstedt, *Graph Theoretic Methods in Multiagent Networks*. Princeton University Press, July 2010.

[161] F. Bullo, J. Cortés, and S. Martínez, *Distributed Control of Robotic Networks: A Mathematical Approach to Motion Coordination Algorithms*. Princeton University Press, July 2009.

[162] E. Montijano, E. Cristofalo, D. Zhou, M. Schwager, and C. Sagüés, "Vision-Based Distributed Formation Control Without an External Positioning System," *IEEE Transactions on Robotics*, vol. 32, no. 2, pp. 339–351, Apr. 2016.

[163] K. Fathian, E. Doucette, J. W. Curtis, and N. R. Gans, "Vision-Based Distributed Formation Control of Unmanned Aerial Vehicles," Aug. 2018.

[164] M. M. H. Fallah, F. Janabi-Sharifi, S. Sajjadi, and M. Mehrandezh, "A Visual Predictive Control Framework for Robust and Constrained Multi-Agent Formation Control," *J Intell Robot Syst*, vol. 105, no. 4, p. 72, July 2022.

[165] M. Abraham, A. Mayne, T. Perez, I. R. De Oliveira, H. Yu, C. Hsieh, Y. Li, D. Sun, and S. Mitra, "Industry-track: Challenges in Rebooting Autonomy with Deep Learned Perception," in *2022 International Conference on Embedded Software (EMSOFT)*, Oct. 2022, pp. 17–20.

[166] G. Leitmann, "Guaranteed ultimate boundedness for a class of uncertain linear dynamical systems," *IEEE Transactions on Automatic Control*, vol. 23, no. 6, pp. 1109–1110, Dec. 1978.

[167] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004.

[168] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features," in *Computer Vision – ECCV 2006*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 404–417.

[169] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," in *2011 International Conference on Computer Vision*, Nov. 2011, pp. 2564–2571.

[170] M. Muja and D. G. Lowe, "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration," in *Proc. 4th Int. Conf. Computer Vision Theory and Applications*, vol. 1. Lisboa, Portugal: INSTICC Press, 2009, pp. 331–340.

[171] E. Malis and M. Vargas, "Deeper understanding of the homography decomposition for vision-based control," INRIA, Report, 2007. [Online]. Available: https://hal.inria.fr/inria-00174036

[172] D. Nister, "An efficient solution to the five-point relative pose problem," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 6, pp. 756–770, June 2004.

[173] H. Li and R. Hartley, "Five-Point Motion Estimation Made Easy," in *18th International Conference on Pattern Recognition (ICPR'06)*, vol. 1, Aug. 2006, pp. 630–633.

[174] D. V. Dimarogonas and K. J. Kyriakopoulos, "A connection between formation infeasibility and velocity alignment in kinematic multi-agent systems," *Automatica*, vol. 44, no. 10, pp. 2648–2654, Oct. 2008.

[175] M. Fiedler, "Algebraic connectivity of graphs," *Czechoslovak Mathematical Journal*, vol. 23, no. 2, pp. 298–305, 1973. [Online]. Available: https://eudml.org/doc/12723

[176] A. Saberi, A. A. Stoorvogel, M. Zhang, and P. Sannuti, *Synchronization of Multi-Agent Systems in the Presence of Disturbances and Delays*, ser. Systems & Control: Foundations & Applications. Cham: Springer International Publishing, 2022.

[177] E. D. Sontag, "Input to State Stability: Basic Concepts and Results," in *Nonlinear and Optimal Control Theory: Lectures given at the C.I.M.E. Summer School Held in Cetraro, Italy June 19–29, 2004*, ser. Lecture Notes in Mathematics. Berlin, Heidelberg: Springer, 2008, pp. 163–220.

[178] A. Kashyap, T. Başar, and R. Srikant, "Quantized consensus," *Automatica*, vol. 43, no. 7, pp. 1192–1203, July 2007.

[179] S. Olaru and H. Ito, "Characterization of Ultimate Bounds for Systems With State-Dependent Disturbances," *IEEE Control Systems Letters*, vol. 2, no. 4, pp. 797–802, Oct. 2018.

[180] A. Astorga, P. Madhusudan, S. Saha, S. Wang, and T. Xie, "Learning stateful preconditions modulo a test generator," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 775–787.

[181] K. Suenaga and T. Ishizawa, "Generalized Property-Directed Reachability for Hybrid Systems," in *Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings*. Berlin, Heidelberg: Springer-Verlag, Jan. 2020, pp. 293–313.

[182] O. Maler and D. Nickovic, "Monitoring Temporal Properties of Continuous Signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 152–166.