UNIVERSITY OF CALIFORNIA, SAN DIEGO

Verification of Sampled-Data Systems using Coq

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Daniel Ricketts

Committee in charge:

Professor Sorin Lerner, Chair
Professor Samuel Buss
Professor William Griswold
Professor Ranjit Jhala
Professor Todd Millstein

2017

The Dissertation of Daniel Ricketts is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2017

DEDICATION

To my wife, parents, and brother, for always believing in me

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I owe a tremendous debt to my adviser Sorin Lerner for supporting me both technically and otherwise throughout a long PhD. From taking me on as an untested student switching research areas to letting me spend half of grad school in Germany with my wife, he has always put my interests first.

A special thanks also goes to my original adviser Mohan Paturi, who always gave me the freedom to pursue what I enjoyed and supported me when switching to a new research area and adviser.

It was a great experience working with Gregory Malecha, who showed me the importance of technical simplicity and abstraction and who convinced me that everything is an adjoint functor. Don Jang's always positive mood and ability to finish any task with an impending deadline are an inspiration. Zachary "Danger" Tatlock's excitement for research, outgoing nature, and ability to bounce back from any setback are a model to emulate, as are his poncho-based outfits. Valentin Robert helped introduce me to the world of verification and always kept a relaxed attitude, no matter the situation.

I very much enjoyed working with the Ardupilot developers, particularly Leonard Hall, who always had time for a fun and enlightening conversation.

One of my earliest collaborators, Moshe Hoffman, showed me what passion for research looks like, while Petros Mol was always there to re mind me that there is more to life than work.

I want to thank Wilson Lian for going through le tour de grad school with me, for all the deep thought-filled conversations, the trap chair-based entertainment, and for always being a supportive and understanding friend.

I also want to thank Matt Der for being an honest and enthusiastic friend, and for always prioritizing soccer training over research so that we may guide the US to a World Cup victory.

I could not have done this without my parents, who taught me to persevere through all challenges, and my brother Scott, who always has my back and showed me that you should never worry about what others think.

And most importantly, I want to thank my wife Maja for being a never ending source of love and support through the highs and lows of Coq, for never letting me get too frustrated, and for always being proud of me no matter what.

Chapter 1, in full, is adapted from material as it appears in International Conference on Formal Methods and Models for System Design 2015. Ricketts, Daniel; Malecha, Gregory; Alvarez, Mario M.; Gowda, Vignesh; Lerner, Sorin, ACM Press, 2015. International Conference on Embedded Software 2016. Ricketts, Daniel; Malecha, Gregory; Lerner, Sorin, ACM Press, 2016. The dissertation author was the primary investigator and author of these papers.

Chapter 2, in full, is adapted from material as it appears in International Conference on Formal Methods and Models for System Design 2015. Ricketts, Daniel; Malecha, Gregory; Alvarez, Mario M.; Gowda, Vignesh; Lerner, Sorin, ACM Press, 2015. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is adapted from material as it appears in International Conference on Embedded Software 2016. Ricketts, Daniel; Malecha, Gregory; Lerner, Sorin, ACM Press, 2016. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is currently being prepared for submission for publication of the material. Ricketts, Daniel; Ghaffari, Azad; Imoleayo, Abel; Lerner, Sorin; Krstic, Miroslav. The dissertation author was the primary investigator and author of this material.

Chapter 5 includes and expands upon material as it appears in International Conference on Formal Methods and Models for System Design 2015. Ricketts, Daniel;

Malecha, Gregory; Alvarez, Mario M.; Gowda, Vignesh; Lerner, Sorin, ACM Press, 2015. International Conference on Embedded Software 2016. Ricketts, Daniel; Malecha, Gregory; Lerner, Sorin, ACM Press, 2016. The dissertation author was the primary investigator and author of these papers.

VITA

2009	Bachelor of Science, Brown University
2009-2010	Software Engineer, Inria
2011	Intern, Telefónica Research
2012	Intern, Facebook
2010–2017	Research Assistant, University of California, San Diego
2017	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Daniel Ricketts, Gregory Malecha, and Sorin Lerner. Modular deductive verification of sampled-data systems. In *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, pages 17:1–17:10, New York, NY, USA, 2016. ACM

D. Ricketts, G. Malecha, M. M. Alvarez, V. Gowda, and S. Lerner. Towards verification of hybrid systems in a foundational proof assistant. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 248–257, Sept 2015

Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Automating Formal Proofs for Reactive Systems. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 452–462, New York, NY, USA, 2014. ACM

Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA+ proofs. In *International Symposium on Formal Methods*, pages 147–154. Springer Berlin Heidelberg, 2012

Nuh Aygun Dalkiran, Moshe Hoffman, Ramamohan Paturi, Daniel Ricketts, and Andrea Vattani. Common knowledge and state-dependent equilibria. In *Algorithmic game theory*, pages 84–95. Springer Berlin Heidelberg, 2012

ABSTRACT OF THE DISSERTATION

Verification of Sampled-Data Systems using Coq

by

Daniel Ricketts

Doctor of Philosophy in Computer Science

University of California, San Diego, 2017

Professor Sorin Lerner, Chair

Due to their safety-critical nature, cyber-physical systems (CPS) demand the most rigorous verification techniques. However, the complexity of the domain puts many cyber-physical systems outside the scope of automated verification tools. Formal deductive proofs hold the potential to verify virtually any property of any system, but proofs for practical cyber-physical systems often require an impractical amount of manual effort. This proof burden can be mitigated by capturing common reasoning patterns in powerful higher-order proof rules. Existing work has focused on proof rules applicable to arbitrary hybrid systems (a formal model for CPS), but many systems actually fall into more constrained classes. One such class of systems are called sampled-data systems, in which a discrete controller runs periodically. In this dissertation, we complement general hybrid system proof rules with a series of rules that leverage the particular structure of sampled-data systems. We demonstrate the applicability of these rules on the double integrator, an important model in robotic and vehicle systems. All work is formalized in the Coq proof assistant, whose expressive logic is crucial to maintaining soundness while applying domain-specific proof rules for sampled-data systems. Finally, we experimentally evaluate our results by implementing verified controllers on a quadcopter and conducting flight tests.

Introduction

Errors in cyber-physical systems (CPS) can lead to disastrous consequences, including loss of life. These consequences mean that such systems demand the most rigorous verification techniques. There has been a variety of work on developing fully-automated tools for verification of cyber-physical systems [26, 36], but due to the complexity of the domain, these tools are only able to verify particular classes of systems and properties. On the other hand, all cyber-physical systems are in range for deductive verification in a proof assistant, at least in theory. In this technique, a user writes a formal model of a system in the language of the proof assistant and then interactively proves it correct. However, one of the typically-stated drawbacks of verification in proof assistants is the extremely high manual labor cost required to produce these proofs.

Mitigating this manual proof burden requires powerful higher-order proof rules that capture common proof strategies. Prior work in deductive verification has approached this task by designing general-purpose, complete proof calculi for hybrid systems [65, 48, 88]. Hybrid systems are CPS models comprised of both a discrete component (e.g. control software) and a continuous component (the physical world). While powerful, the generality of the proof calculi prevents them from leveraging particular common characteristics of cyber-physical systems. For example, no proof rule in [65] assumes that the time between executions of a controller is bounded because this is not true of all hybrid systems. However, this is true of many realistic hybrid systems, and proofs about such systems will tend to follow a similar proof structure. Thus, it is beneficial to complement general hybrid system proof rules with domain specific proof rules that capture common reasoning patterns.

This dissertation presents and applies a series of proof rules that capture common reasoning patterns in the important domain of *periodic sampled-data systems* [16]. In such a system, there is a digital controller that runs periodically. In between executions of the controller, the system evolves according to continuous physical dynamics. Many modern cyber-physical systems fit into this domain. The remainder of the introduction provides an overview of the proof rules. In general, the rules seek to leverage timing characteristics of systems and improve modularity of reasoning.

In **Chapter 2** we present two rules: one for verifying a single sampled-data component under assumption on the environment and another for composing such a component with another that satisfies this assumption. The first rule decomposes verification into a property of the discrete controller and the continuous dynamics, automatically handling the fact that the time between executions of the controller is bounded. This tailored decomposition eliminates some of the basic tedious manipulation common to every sampled-data system, allowing one to focus on the application specific aspects of verification. The second proof rule allows for component composition with non-cyclical assumptions – that is, a component $C1$ can guarantee an invariant Q while assuming and invariant P guaranteed by $C2$. However, $C2$ cannot rely on the invariance of P when guaranteeing Q. In spite of this restriction, we show that such a rule has important applications for verifying controllers in the presence of sensor error and delay.

Next, **Chapter 3** presents a general framework for building sampled-data systems *modularly*. This framework differs from the above composition approach by requiring that each component provide a stronger interface. In particular, rather than proving invariance of a property, each component provides preservation of an inductive invariant, and a notion of progress of the system under that inductive invariant. This stronger interface comes at a minor cost while proving two important benefits. First, it allows for cyclic dependencies between sampled-data components, thus removing the restriction from Chapter 2. Second, it allows us to explore a richer set of operators for modularly building and verifying sampled-data components, namely substitution, conjunction, and disjunction. It is this second benefit that we explore thoroughly by applying our framework to build verified 3-dimensional geofences for a UAV. We show that our theory can handle the important situation in which different components output to the same set of actuators, as exemplified by the geofence application.

Finally, in **Chapter 4**, we revisit verification of a single sampled-data system component. Contrary to our prior approaches, we began by building a geofence that was good enough to be adopted by the popular open source UAV autopilot called Ardupilot [85]. After building such a module (now available in the latest Ardupilot release), we attempted to formally verify a component of it in Coq, particularly the logic that prevents the vehicle from violating a boundary in a single spatial dimension. Similar logic is present in the atomic components from Chapters 2 and 3, but realistic performance requirements for Ardupilot resulted in considerably more complicated control logic. This additional complexity demanded better proof rules, and we built rules that improve upon the state of the art in formal verification in two ways.

First, deductive techniques typically involve some continuous analogue of induction, e.g. differential induction [63] or barrier certificates [69]. Recent work from the control theory community [40, 94, 60] has produced a new version of barrier certificates that are less conservative for closed properties. We provide the first implementation of this approach in a formal verification context, and demonstrate its ease of use on a component of the Ardupilot geofencing module.

Second, control systems are often designed under the assumption that controllers run continuously, while the actual implementation is typically a sampled-data system.

System designers can compensate for this (and other) approximations by adding a safety "buffer" to the system. For example, the Ardupilot geofence module stops the vehicle 1 meter prior to the actual safety boundary. We developed a proof rule that formalizes this design approach by decomposing verification into a condition on the continuous time approximation and another on the approximation error. This rule allows one to perform the majority of reasoning in a purely continuous model using powerful techniques resulting from over a century of control theory research [40, 94, 60].

As already mentioned, our running application in this work is a geofencing controller for UAVs, an important application due to their potential safety threat combined with widespread use by hobbyists and businesses alike. We formally model such systems using the double integrator from control theory, in which the controller affects the acceleration (second derivative) of position. This is an approximation of reality, as the actual quadcopter controller affects the angular torque of the vehicle. The angular torque is related to the second derivative of position via complex attitude dynamics. Although our model is an approximation, the double integrator is a benchmark problem for robotics, vehicles, and other systems that involve a point mass moving in space [70]. Thus, the double integrator serves as a canonical example for evaluating the expressiveness of the proof rules presented in this dissertation.

Since we use an approximate model of UAV dynamics, we would like to ensure that the controllers we build and verify are not toys. We have attempted to justify the realistic nature of our work by flying the controllers we verify on an actual quadcopter. Throughout this dissertation, we discuss lessons learned from this experience.

Rigorous verification requires that results be mechanically checked in some way. Rather than implementing a standalone tool for this task, we performed all verification within the foundational Coq proof assistant. Previous work [95] has empirically demonstrated that foundationally verified systems are highly reliable. While important, we would like to emphasize a less discussed benefit of higher-order proof assistants like Coq. A standalone domain specific verification tool would require adding each proof rule as an axiom. Moreover, a new proof rule might have side conditions not expressible in the tool's logic, requiring custom reasoning to handle these side conditions. In this context, the axiom and associated custom reasoning have the potential to compromise soundness.

On the other hand, the *expressiveness* of Coq allows one add new proof rules as theorems that are formally proven within Coq's logic. This means that one can extend any given set of general proof rules in Coq (e.g. general proof calculi for hybrid systems) with powerful domain specific proof rules (e.g. our sampled-data system rules), without compromising soundness. Such an extension improves verification productivity by ensuring that a user can apply the right domain-specific proof rule for his or her application while still being able to depend on the above mentioned reliability guarantees.

> **Thesis:** Domain-specific proof rules implemented in a higher-order proof assistant simplify reasoning without compromising soundness for sampled-data systems.

It is thus the view of the dissertation author that the expressive framework of a higher-order proof assistant is crucial to scaling formal verification to realistic cyber-physical systems, and we hope that the proof rules and associated applications in this document provide evidence for this claim.

Chapter 1

Preliminaries

In this chapter, we describe the logical framework used in Chapters 2 and 3. Chapter 4 uses a different framework, so the logical background is presented within that chapter. All of our work is formalized inside the Coq proof assistant, but for exposition purposes, we focus on the mathematical concepts rather than concrete Coq syntax.

1.1 Linear Temporal Logic

In Chapters 2 and 3, we encode sampled-data systems and their properties within discrete-time linear temporal logic (LTL). An LTL formula specifies the possible traces of a system. In our model, a trace is an infinite sequence of states representing observations of a system at discrete points in time. A state is a mapping from variables to real numbers. Inspired by Lamport's Temporal Logic of Actions (TLA) [44], formulas in our logic are classified into *state formulas* (predicates over a single state), *action formulas* (state relations specifying system transitions), and *trace formulas* (predicates over traces). In action formulas, the values of variables in the current state are notated using bold script, e.g. $\mathbf{x}$, while the values of variables in the next state use bold script with a prime, e.g. $\mathbf{x}'$. Variables not mentioned in a formula are unconstrained.

For example, the following formula describes a system where the initial value of

$\mathbf{x}$ is 0 and the value of $\mathbf{x}$ is incremented during each transition.

$$\mathbf{x} = 0 \wedge \Box\left(\mathbf{x}' = \mathbf{x} + 1\right)$$

The initial condition ($\mathbf{x} = 0$) is a state formula. The transition relation ($\mathbf{x}' = \mathbf{x} + 1$) is an action formula and refers to values in the next state using a prime, e.g. $\mathbf{x}'$. Both the transition relation and the property are lifted to trace formulas using the always modality ($\Box$). When always is applied to an action formula, it means that all pairs of temporally adjacent states are related by the action formula. When always is applied to a state formula, it means that all states satisfy the property.

For convenience, we also use an operator $\mathsf{Unchanged}(X)$, where X is a set of variables, to represent the LTL formula stating that each variable in X is equal to its primed counterpart:

$$\mathsf{Unchanged}(\{\mathbf{x_1}, \ldots, \mathbf{x_n}\}) \equiv \mathbf{x_1}' = \mathbf{x_1} \wedge \ldots \wedge \mathbf{x_n}' = \mathbf{x_n}$$

Finally, the semantics of formulas is defined in terms of two relations: "models" (written $tr \models P$) states when a predicate (P) holds on a trace (tr), and "entails" (written $P \vdash Q$, or just $\vdash Q$ when P is trivial) states when one predicate implies another on *all* traces, i.e.

Definition 1.1 (LTL Entailment)**.**

$$P \vdash Q \quad \equiv \quad \forall tr, tr \models P \rightarrow tr \models Q$$

For example, the following states that all traces of the above system have the property that $\mathbf{x}$ is always at least 0.

$$\vdash \mathbf{x} = 0 \wedge \Box \left(\mathbf{x}' = \mathbf{x} + 1\right) \rightarrow \Box \left(\mathbf{x} \geq 0\right)$$

The implication means that the traces of the system are a subset of the traces for which $\mathbf{x}$ is at least 0 in all states.

1.2 Sampled-data systems in LTL

In a periodic sampled-data system, the state repeatedly transitions either continuously according to some differential (in)equations or discretely according to the (possibly nondeterministic) controller. In addition, the elapsed time between discrete transitions of the controller is bounded by some constant. In LTL, we can model such systems using a formula of the form

$$I \wedge \Box(\mathsf{Sys}_\Delta\, D\, \mathscr{W})$$

Here, we use the action formula $\mathsf{Sys}_\Delta\, D\, \mathscr{W}$, specifying transitions of a sampled-data system, where: D is an action formula specifying the discrete controller, and $\mathscr{W}$ is a predicate over state variables and their derivatives. This can be used to express systems of differential equations $\dot{\mathbf{x}} = e_1 \wedge \dot{\mathbf{y}} = e_2$, differential inequalities $\dot{\mathbf{z}} \leq e$, and even pure state predicates restricting the evolution of variables with respect to each other $x \leq y$. These expressions can be conjoined to express all three concepts in the same continuous evolution. Formally,

Definition 1.2 (Sys abstraction)**.**

$$\begin{aligned} &\mathit{Sys}_\Delta\, D\, \mathscr{W} \triangleq \\ &\qquad\quad D \wedge \boldsymbol{\tau} = 0 \wedge 0 < \boldsymbol{\tau}' \leq \Delta \\ &\qquad \vee \ \ \mathsf{Continuous}\,(\mathscr{W} \wedge \dot{\boldsymbol{\tau}} = -1) \wedge \boldsymbol{\tau}' \geq 0 \end{aligned}$$

In this action formula, the disjunction captures the fact that the system transitions either continuously according to the physical world or discretely according to the controller. The definition encapsulates both the semantics of the continuous transition and the timing characteristics of the system.

Informally, $\mathsf{Continuous}(\mathscr{W})$ means that the state evolves for *some* amount of time according to a continuous function whose value and derivative at each point in time satisfy the predicate in $\mathscr{W}$. Formally, $\mathsf{Continuous}(\mathscr{W})$ is an LTL action formula, defined as follows:

Definition 1.3 (Continuous evolution)**.**

$$\begin{aligned}
&\mathsf{Continuous}\mathscr{W} \equiv \\
&\quad \exists (r:\mathbb{R})\,(f:\mathbb{R}\to \mathit{Var}\to\mathbb{R}), 0<r \\
&\qquad \wedge \forall 0\le t\le r, \mathscr{W}(f(t),\dot{f}(t)) \\
&\qquad \wedge x_1 = f(0,x_1)\wedge\ldots\wedge x_n = f(0,x_n) \\
&\qquad \wedge {x_1}' = f(r,x_1)\wedge\ldots\wedge {x_n}' = f(r,x_n)
\end{aligned}$$

Here, $x_1,\ldots,x_n$ are variables in the system, r is the amount of time that the system evolves for, and f is a differentiable function from time to state that gives the evolution of the system state during the continuous transition. The first conjunct expresses that the predicate $\mathscr{W}$ holds on the state and its derivative during the entire system evolution. The second and third conjuncts relate the current state to the value of the solution f at 0 and the next state to the value of f at time r.

At first glance, this definition of continuous transitions may look strange since it seems to allow the trace to "skip" states – that is, any single sequence of states satisfying

Sys does not capture all intermediate states of the system. Instead, the discrete trace captures finite observations along the continuous evolution of the physical world. In fact, it may seem as though a sequence of states is a poor fit for describing the continuously evolving physical world since time and other continuous variables can only advance in discrete steps. The core of the argument lies in the fact that in LTL we prove properties of *all* sequences of states rather than properties of a single sequence of states. When we prove $\vdash \mathsf{Init} \wedge \mathsf{Sys}_\Delta\ D\ \mathscr{W} \rightarrow P$ for some property P, we are proving that P holds on all sequences of states that satisfy $\mathsf{Init} \wedge \mathsf{Sys}_\Delta\ D\ \mathscr{W}$. In other words, we are proving properties of all possible sequences of observations of the hybrid system's state. While a single trace may skip a certain state during a continuous transition, another trace does include that state because the definition of $\mathsf{Continuous}$ captures *all* possible continuous transitions of any duration. The soundness of this encoding is argued by Lamport in [45].

Finally, the definition of Sys also expresses that at most Δ time elapse between executions of the controller. This timing constraint is captured using the variable $\boldsymbol{\tau}$ (not mentioned in D or $\mathscr{W}$), which tracks the time that elapses between successive transitions of the discrete controller. During the continuous evolution of the system, $\boldsymbol{\tau}$ decreases at the same rate as time, i.e. $\dot{\boldsymbol{\tau}} = -1$, and $\boldsymbol{\tau}' \geq 0$ ensures that no more than Δ time elapses between successive discrete transitions of the controller. The discrete transition occurs when the timer has expired ($\boldsymbol{\tau} = 0$); this transition resets the timer to a positive value that is at most Δ.

1.3 Acknowledgments

This chapter in full, is adapted from material as it appears in International Conference on Formal Methods and Models for System Design 2015. Ricketts, Daniel; Malecha, Gregory; Alvarez, Mario M.; Gowda, Vignesh; Lerner, Sorin, ACM Press, 2015. International Conference on Embedded Software 2016. Ricketts, Daniel; Malecha, Gregory;

Lerner, Sorin, ACM Press, 2016. The dissertation author was the primary investigator and author of these papers.

Chapter 2

Discrete Induction and Basic Composition

This chapter presents two proof rules for reasoning about sampled-data systems. The first rule is a discrete induction rule specialized to sampled-data systems. This rule decomposes an inductive safety proof into a proof about the initial state, a proof about the discrete transition, and a proof about the continuous transition. For each of the transitions, the proof rule introduces timing constraints that are guaranteed by the periodic sampled-data model. Such a rule automatically manages the aspects of an inductive safety proof that are common to all periodic sampled-data systems, allowing the user to focus on the application specific aspects of the proof. Without such a proof rule, this tedious decomposition would have to be done manually.

The second rule provides a simple mechanism for composing systems with non-cyclic dependencies. This allows one component to assume that invariant of another when proving its own invariant. We demonstrate how this can be used to reason about systems in the presence of both sensor error and delay by chaining instances of this rule together. Chapter 3 presents an alternate approach that removes the non-cyclic restriction.

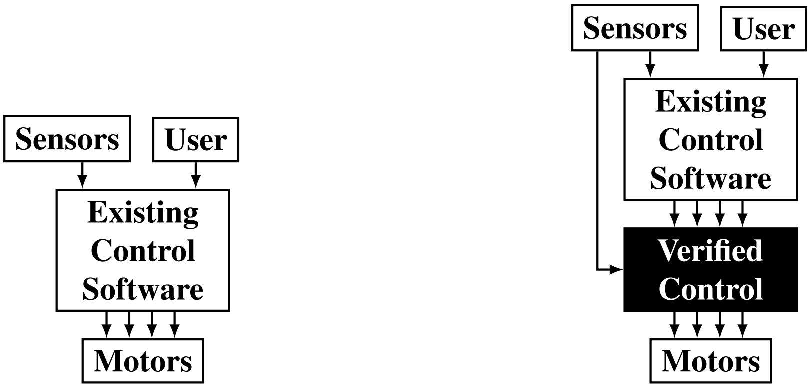
We introduce the proof rules along side a running example: a controller that prevents a simple model of a quadcopter from violating some maximum velocity. We additionally used the rules to verify a controller that prevents a quadcopter from violating some maximum height, and used our composition rule to verify them both in the presence of sensor error and delay.

In an effort to ensure that our examples are realistic, we implemented them on a actual quadcopter and performed flight tests. Doing so forced us to confront an important issue. Since our controllers are components of a much larger system containing numerous controllers, how do we architect the autopilot to incorporate our controllers, while still maintaining our safety guarantees? Throughout the chapter, we describe our solution and discuss engineering trade-offs. We also discuss discrepancies between our model and reality as well as lessons learned from doing foundational verification of sampled-data systems. All results in this chapter were implemented in Coq, and the development is available from: https://github.com/ucsd-pl/veridrone/tree/MEMOCODE-15.

2.1 Overview

We start by giving an overview of how our running controller examples fit into the architecture of a quadcopter's autopilot. We would like our two controllers to ensure that a quadcopter does not exceed some maximum velocity and some maximum height. These are practical restrictions from a safety standpoint. For example, in order to ensure that small UAVs do not interfere with larger aircraft, the FAA mandates that they do not fly more than 400 feet above ground level.

Our goal here will be to (1) add some simple logic to the control software to prevent violation of these safety properties and (2) formally verify that this logic guarantees that the safety properties hold, with respect to some model of the physical dynamics of the UAV. However, we would like to accomplish this while allowing the UAV to run its existing complex controllers that may not guarantee these properties – these sorts of controllers have the potential to achieve high performance, but their complexity



(a) (b)

Figure 2.1. A simplified depiction of UAV architecture 2.1a without and 2.1b with one of our controllers.

presents a challenge to formal verification.

We accomplish this by implementing our controllers as safety checks that run after all of the (potentially complex) controllers have run, but before any signals are sent to the motors. Our controllers perform a safety check on the outputs from the higher level controllers. In our height limit example, the controller checks whether the output of the higher level controllers would put the UAV in a state in which it could not stop before exceeding the upper bound. If the check passes then the controller issues exactly the same outputs as the higher level controllers. Otherwise, the controller issues a conservative action to ensure the desired safety property.

Our architecture, inspired by the simplex architecture [75], is depicted in Figure 2.1, which shows a simplified view of a UAV architecture with and without our controllers. This architecture allows us to focus our verification effort on our safety critical controllers, without needing to reason about complex existing controllers. This design makes the verification tractable, while retaining the benefits of state-of-the art controllers. We discuss this architecture in more detail in Section 2.4.2.

2.2 Model and Verification

We now focus on our controller that ensures the velocity of a simple one-dimensional model of a quadcopter never exceeds a constant upper bound $\mathbf{v}_{\mathsf{ub}}$. We start by modeling the system using our abstraction Sys (Definition 1.2), present our sampled-data induction rule, and demonstrate how to use this rule to verify the safety invariant of the velocity bounding controller. In Section 2.3, we will present our composition rule and show how to use it to verify safety in the presence of sensor error and delay.

2.2.1 Model

Our velocity controller makes use of the following variables (by convention, lower case variables are continuous and upper case variables are discrete): $\mathbf{v}$ is the actual velocity of the system, whose behavior will be specified by differential equations encoding the physics of the real world; $\mathbf{v_{max}}$ is an upper bound on $\mathbf{v}$ (e.g. produced by a sensor) and is an input to our controller; $\mathbf{T}_?$ is the thrust requested by the higher level controller, which is also an input to our controller; $\mathbf{a}$ is the thrust produced by our controller, which gets sent to the motors. Our controller is defined as follows:

$$\mathsf{VelCtrl} \equiv \mathsf{Sys}_\Delta\, D\, \mathscr{W}$$

where

$$\begin{array}{rcl} \mathscr{W} & \equiv & \dot{\mathbf{v}} \leq \mathbf{a} - g \\ D & \equiv & ((\mathbf{T}_? - g) \cdot \Delta + \mathbf{v_{max}} \leq \mathbf{v}_{\mathsf{ub}} \wedge \mathbf{a}' = \mathbf{T}_?) \vee \mathbf{a}' = g \end{array}$$

$\mathscr{W}$ states the differential inequality capturing how velocity is related to thrust ($\mathbf{a}$) and gravity (g). This relationship is an inequality rather than an equality because we are modeling a quadcopter, whose vertical thrust is only upper-bounded by the thrust produced by the motors. We discuss this further in Section 2.4. The action formula D captures the control logic of our controller. The disjunction encodes, essentially, the following conditional statement: $\texttt{if}\,\mathbf{T_?}\cdot\Delta+\mathbf{v_{max}}\leq\mathbf{v}_{\mathsf{ub}}\,\texttt{then}\,\mathbf{a}'=\mathbf{T_?}\,\texttt{else}\,\mathbf{a}'=g$; though it also allows executing the safe action ($\mathbf{a}'=g$) when the check succeeds.

Since Sys unfolds directly to an LTL formula, we can express the correctness of our controller directly in LTL as follows, where $\vdash$ (Definition 1.1) represents entailment in LTL, expressing that the formula to the right of the $\vdash$ holds on all traces satisfying the formula to the left of the $\vdash$:

$$\Box\mathbf{v}\leq\mathbf{v_{max}}\vdash\mathsf{Init}\wedge\mathsf{VelCtrl}\rightarrow\Box\mathbf{v}\leq\mathbf{v}_{\mathsf{ub}} \tag{2.1}$$

$$\mathsf{Init}\equiv\max(0,\mathbf{a}-g)\cdot\Delta+\mathbf{v}\leq\mathbf{v}_{\mathsf{ub}}$$

This formula states that if $\mathbf{v_{max}}$ is always an upper bound on the actual velocity ($\Box\mathbf{v}\leq\mathbf{v_{max}}$), then the velocity controller ensures that the velocity of the system is always less than or equal to $\mathbf{v}_{\mathsf{ub}}$. This requires a predicate on the initial state, given by Init, which states that the velocity is at most $\mathbf{v}_{\mathsf{ub}}$ and furthermore is small enough that it will still be at most $\mathbf{v}_{\mathsf{ub}}$ when the controller first runs (which will be at most Δ time units away). For now, we do not specify how the value of $\mathbf{v_{max}}$ is produced; that is, $\mathbf{v_{max}}$ does not appear in any action formulas (transitions). Instead, we simply assume that such a value is provided to the controller. In Section 2.3, we will show how to specify systems that produce a $\mathbf{v_{max}}$ satisfying this assumption and we will show how to compose them with the velocity controller.

2.2.2 Proof

We now show how we prove the above LTL formula in Coq. To do so, we need an inductive invariant that is preserved both by the continuous transitions (those of the world) and the discrete program. Although we have implemented several mechanisms for simplifying reasoning about Sys, we currently do not infer invariants automatically. In the case of the velocity controller, the inductive invariant is relatively simple: the velocity cannot violate the upper bound before the next execution of the controller:

$$\max(0, \mathbf{a} - g) * \boldsymbol{\tau} + \mathbf{v} \leq \mathbf{v}_{\mathsf{ub}}$$

Here, the variable $\boldsymbol{\tau}$, introduced in Definition 1.2 tracks the maximum amount of time until the next execution of the controller.

In order to prove that this formula is an inductive invariant, we use the the SYSIND proof rule (Theorem 2.1), which is a special case of discrete induction tailored to systems that are described by our Sys construct. Proof rules such as this one allow us to abstract the implementation of Sys and make for overall cleaner proofs. Informally, SYSIND states that a formula P is an (inductive) invariant of a system if P holds on all possible initial states of the system, and P is preserved by the two possible transitions that the system can make. We use P' to denote the formula P with all unprimed variables $\mathbf{x}$ replaced with their primed counterpart $\mathbf{x}'$.

Theorem 2.1 (SYSIND)**.** *For state formulas P, Q, I, action formula D, constant $\Delta \in \mathbb{R}$, and evolution predicate $\mathscr{W}$, if the following conditions hold:*

i) $Q \vdash 0 \leq \boldsymbol{\tau} \leq \Delta \wedge I \rightarrow P$

ii) $Q \vdash \boldsymbol{\tau} = 0 \wedge P \wedge D \wedge 0 \leq \boldsymbol{\tau}' \leq \Delta \rightarrow P'$

iii) $Q \vdash \boldsymbol{\tau} \leq \Delta \wedge 0 \leq \boldsymbol{\tau}' \leq \boldsymbol{\tau} \wedge P \wedge \mathsf{Continuous}(\mathscr{W}) \rightarrow P'$

then

$$\Box Q \vdash I \wedge \mathit{Sys}_{\Delta}\, D\, \mathscr{W} \rightarrow \Box P$$

To illustrate how the proof works, we walk through the proof obligations obtained by applying SYSIND to our velocity controller. First, we must prove that P holds on all initial states of the system:

$$\mathbf{v} \le \mathbf{v_{max}} \;\vdash \left[\begin{array}{ll} & 0 \le \boldsymbol{\tau} \le \Delta \\ \wedge & \max(0, \mathbf{a} - g) \cdot \Delta + \mathbf{v} \le \mathbf{v}_{\mathsf{ub}} \\ \rightarrow & \max(0, \mathbf{a} - g) \cdot \boldsymbol{\tau} + \mathbf{v} \le \mathbf{v}_{\mathsf{ub}} \end{array}\right]$$

Proving this requires first order reasoning over real arithmetic in Coq. We can solve simple obligations such as this one, using existing Coq real arithmetic decision procedures [12] that produce foundational Coq proofs completely automatically. While these procedures are not complete, they are still able to discharge many obligations that arise in practice. When they are unable to completely prove a goal, we are forced to manually construct a machine-checked proof of the remaining obligations. This requires manual application of real arithmetic lemmas such as transitivity of comparison operators. This can become quite tedious as the system becomes more complex. The compositional reasoning techniques, which we describe in Section 2.3, help to reduce this burden by producing smaller arithmetic goals that only deal with a part of the system. We discuss this benefit in more detail in Section 2.3.4.

Next, we prove that the inductive invariant is preserved by discrete steps of the system. There are actually two cases to prove: when the proposed thrust passes the controller's safety check and when the controller issues a thrust equal to gravity. In the first case, we are left to prove the following proof obligation (the reasoning in the second case is simpler):

$$\mathbf{v} \le \mathbf{v_{max}} \;\;\vdash\;\; \left[\begin{array}{ll} & \boldsymbol{\tau} = 0 \\ \wedge & \max(0, \mathbf{a} - g) * \boldsymbol{\tau} + \mathbf{v} \le \mathbf{v}_{\mathsf{ub}} \\ \wedge & (\mathbf{T}_{?} - g) * \Delta + \mathbf{v_{max}} \le \mathbf{v}_{\mathsf{ub}} \\ \wedge & \mathbf{a}' = \mathbf{T}_{?} \\ \wedge & \mathbf{v}' = \mathbf{v} \\ \wedge & 0 \le \boldsymbol{\tau}' \le \Delta \\ \rightarrow & \max(0, \mathbf{a}' - g) * \boldsymbol{\tau}' + \mathbf{v}' \le \mathbf{v}_{\mathsf{ub}} \end{array}\right]$$

Proving this obligation requires first order reasoning over real arithmetic, but fits into the automation described above.

Finally, we prove that the inductive invariant is preserved by continuous transitions. This proof obligation is slightly more difficult:

$$\mathbf{v} \le \mathbf{v_{max}} \;\;\vdash\;\; \left[\begin{array}{ll} & \boldsymbol{\tau} \le \Delta \wedge 0 \le \boldsymbol{\tau}' \le \boldsymbol{\tau} \\ \wedge & \max(0, \mathbf{a} - g) \cdot \boldsymbol{\tau} + \mathbf{v} \le \mathbf{v}_{\mathsf{ub}} \\ \wedge & \mathsf{Continuous}(\mathscr{W}) \\ \rightarrow & \max(0, \mathbf{a}' - g) \cdot \boldsymbol{\tau}' + \mathbf{v}' \le \mathbf{v}_{\mathsf{ub}} \end{array}\right]$$

Continuous evolution of the physical world is expressed by the formula $\mathsf{Continuous}(\mathscr{W})$ (Definition 1.3). We prove this obligation using our adaptation of Platzer's differential induction proof rule [62, 63], which justifies a technique for proving invariants of a system of differential equations without computing an explicit solution. Roughly speaking, differential induction captures the fact that $e_1 \le e_2$ is preserved by a continuous transition (e.g. $\mathsf{Continuous}(\mathscr{W})$) if the derivative of e_1 is less than or equal to the derivative of e_2, under the constraints given by $\mathscr{W}$. Applying differential induction leaves us to prove a first order formula over real arithmetic that the automation can solve with a minimal amount of assistance.

Proving these four goals completes the proof of (2.1). Since the inductive invariant was simple and the arithmetic reasoning was within the scope of Coq's built-in automation, this proof was relatively easy. However, the proof demonstrates the general mechanics of proving an invariant of a periodic sampled-data system and illustrates how SYSIND abstracts some of the tedious reasoning common to all such systems.

2.3 Composition

While the velocity controller does guarantee the safety property, it requires an assumption about an input, namely that $\mathbf{v} \leq \mathbf{v_{max}}$. We could modify the specification of the velocity controller so that it specifies the transition behavior of $\mathbf{v_{max}}$ in a way that guarantees that $\mathbf{v} \leq \mathbf{v_{max}}$, thus removing the assumption. That is, we could modify the controller specification to show how $\mathbf{v_{max}}$ is produced by adding $\mathbf{v_{max}}$ to action formulas of the specification. However, this would require reproving the safety theorem of the velocity controller (formula (2.1)). By leaving the sensor under-specified in the velocity controller, we are able to compose the velocity controller with *any* system that guarantees $\Box\mathbf{v} \leq \mathbf{v_{max}}$, without needing to reprove the safety theorem of the velocity controller. In this section, we show how to do this for several examples, using a general composition rule for our Sys abstraction.

2.3.1 Sensor Error

Real sensors always have some notion of error. For example, the readings from a barometer may be affected by pressure differences due to local weather or GPS may only give accurate information to within several feet. In addition, sensors, detect physical, continuous values and compute discrete approximations of these values, for example using fixed- or floating point. These finite representations can not hope to be exactly the true values of the variables they represent. While the first type of error can be modeled probabilistically, we assume that a sensor can be assigned a deterministic conservative upper bound on the error. Thus, it is easy to model both kinds of error in our framework since we only need to specify predicates on the sensed values.

We start with a simple specification of a sensor that can read the value of $\mathbf{v}$ to within some error ε. To do so, we first define $\mathsf{Sensor}\big(S,\mathscr{W},\Delta\big)$, which takes an LTL state formula S and a positive number Δ, and produces an LTL formula:

$$\mathsf{Sensor}\big(S,\mathscr{W},\Delta\big) \equiv \mathsf{Sys}_{\Delta}\ (\mathsf{Unchanged}(vars(\mathscr{W} \wedge S)))\ (\mathscr{W} \wedge S)$$

This formula expresses the system in which S holds throughout every continuous transition, and all variables in the continuous transition are unchanged by the discrete transition. The action formula expressing that all variables in the continuous transition are unchanged is given by $\mathsf{Unchanged}(vars(\mathscr{W}\ \wedge\ S))$. In the actual Coq implementation, the set of variables must be supplied manually.

Intuitively, S is intended to express the relationship between the physical variable that the sensor is tracking and the actual value it reads. For a sensor of some physical variable $\mathbf{x}$, this relationship is $\mathbf{x}-\varepsilon \leq \mathbf{x_{sense}} \leq \mathbf{x}+\varepsilon$, where $\mathbf{x_{sense}}$ is the sensed value. However, for our purposes, we actually need an upper bound on $\mathbf{x}$, which we accomplish by offsetting $\mathbf{x_{sense}}$ by ε:

$$\mathsf{Sense}(\mathbf{x},\mathbf{x}_{max}) \equiv \mathbf{x}-\varepsilon \leq \mathbf{x_{sense}} \leq \mathbf{x}+\varepsilon\ \wedge\ \mathbf{x}_{max} = \mathbf{x_{sense}}+\varepsilon$$

In order to satisfy the assumption of the velocity controller, we instantiate Sense with $\mathbf{v}$ and $\mathbf{v_{max}}$ and need to prove that for any $\mathscr{W}$, Δ, and $\varepsilon \geq 0$,

$$\vdash \mathsf{Sense}(\mathbf{v}, \mathbf{v_{max}}) \ \wedge \ \mathsf{Sensor}\big(\mathsf{Sense}(\mathbf{v}, \mathbf{v_{max}}), \mathscr{W}, \Delta\big) \rightarrow \Box \mathbf{v} \leq \mathbf{v_{max}} \tag{2.2}$$

This theorem follows from SYSIND and simple reasoning about linear real arithmetic.

2.3.2 Composition

We are now in a position to compose the sensor module with our velocity controller. First, let Sys composition ($\otimes$) be defined by conjoining corresponding formulas:

Definition 2.1. *Conjunctive composition*

$$\mathit{Sys}_\Delta\, D_1\, \mathscr{W}_1 \otimes \mathit{Sys}_\Delta\, D_2\, \mathscr{W}_2 \equiv \mathit{Sys}_\Delta\, \left(D_1 \wedge D_2\right)\, \left(\mathscr{W}_1 \wedge W_2\right)$$

Note that since all LTL formulas operate on the same state variables, conjunction is a very general notion of composition.

Using the definition of $\otimes$, we can state the theorem that the composition of our sensor with our velocity controller satisfies the safety property $\Box \mathbf{v} \leq \mathbf{v}_{\mathsf{ub}}$ without any assumptions on $\mathbf{v_{max}}$:

$$\vdash \mathsf{Sensor}\big(\mathsf{Sense}(\mathbf{v}, \mathbf{v_{max}}), \mathscr{W}, \Delta\big) \otimes \mathsf{VelCtrl} \rightarrow \Box \mathbf{v} \leq \mathbf{v}_{\mathsf{ub}}$$

This theorem follows immediately from SYSCOMPOSE (Theorem 2.2). SYSCOMPOSE states that if the first system guarantees an invariant, then the second system can assume that invariant when it proves its safety condition. The combined system does not need the assumption; it has been satisfied by the first system, and has both properties. Similar to SYSIND, SYSCOMPOSE abstracts all of the reasoning for manipulating the internals of the Sys abstraction. Crucially, when we apply SysCompose, we do not need to reprove any theorems about the two systems. Instead we can simply use the soundness proofs of the components to satisfy the premises of SysCompose.

Theorem 2.2 (SysCompose)**.** *For state formulas P, Q, I_a, I_b, action formulas D_a, D_b, constant $\Delta \in \mathbb{R}$, and evolution predicate $\mathscr{W}$, if the following conditions hold:*

i) $\vdash I_a \ \wedge \ \mathit{Sys}_\Delta \, D_a \, \mathscr{W} \rightarrow \Box P$

ii) $\Box P \vdash I_b \ \wedge \ \mathit{Sys}_\Delta \, D_b \, \mathscr{W} \rightarrow \Box Q$

then

$$\vdash I_a \ \wedge \ I_b \ \wedge \ \mathit{Sys}_\Delta \, D_a \, \mathscr{W} \otimes \mathit{Sys}_\Delta \, D_b \, \mathscr{W} \rightarrow \Box (P \wedge Q)$$

2.3.3 Delay Compensation

When we compose the sensor specification with the velocity controller, we implicitly assume that the velocity controller can instantaneously read and compute with the value produced by the sensor module, $\mathbf{v_{max}}$. In reality, due to communication or computation time, this may not be the case. It may be the case that there is some delay between when the sensor module produces a value and when it can actually be used in the controller's safety check. For example, suppose that the sensor module actually outputs some value, represented by the variable $\mathbf{v_{max\_pre}}$ that cannot instantaneously be used in a safety check. The following system compensates for this delay

$$\mathsf{DelayComp} \equiv \mathsf{Sys}_\Delta \, D \, \mathscr{W}$$

where $\mathscr{W} \equiv \dot{\mathbf{v}} \leq \mathbf{a} - g$ as before and

$$D \equiv \mathbf{v_{max}}' = \mathbf{v_{max\_pre}} + \Delta \cdot \mathsf{max}(0, \mathbf{a}' - g)$$

In this system, D uses the current value of $\mathbf{v_{max\_pre}}$ to compute an upper bound on $\mathbf{v}$ for the next Δ time.

The correctness property for this system is

$$\Box\mathbf{v} \leq \mathbf{v_{max\_pre}} \vdash I \,\wedge\, \mathsf{DelayComp} \rightarrow \Box\mathbf{v} \leq \mathbf{v_{max}} \tag{2.3}$$
$$I \equiv \mathbf{v_{max}} = \mathbf{v} + \Delta \cdot \mathsf{max}(0, \mathbf{a} - g)$$

Notice that this property relies on the assumption $\Box\mathbf{v} \leq \mathbf{v_{max\_pre}}$. However, we can use the sensor module above to satisfy this assumption and use SYSCOMPOSE to verify the combined system without any assumptions, again without reproving the properties of the individual systems:

$$\vdash \mathsf{Sensor}\big(\mathsf{Sense}(\mathbf{v}, \mathbf{v_{max\_pre}}), \mathscr{W}, \Delta\big) \otimes \mathsf{DelayComp} \rightarrow \Box\mathbf{v} \leq \mathbf{v_{max}}$$

Now we have a new system that guarantees the assumption of the velocity controller, so we can compose them and easily prove the theorem:

$$\vdash \mathsf{Sensor}\big(\mathsf{Sense}(\mathbf{v}, \mathbf{v_{max\_pre}}), \mathscr{W}, \Delta\big) \otimes \mathsf{DelayComp} \otimes \mathsf{VelCtrl} \rightarrow \Box\mathbf{v} \leq \mathbf{v}_{\mathsf{ub}}$$

This approach can be continued for any other sensors or full-blown controllers that can be specified and verified within the Sys abstraction.

2.3.4 Height controller

In addition to controlling velocity through a first-derivative, we have used our deductive approach to control position through a second derivative. In this section we describe our implementation of a controller to enforce an upper bound on height by controlling acceleration. Note that if we were able to directly set the velocity then we could reuse the velocity controller (and its proof) simply by renaming the variables. Since directly setting velocity is unrealistic, we built a new controller that bounds position by setting its second derivative.

$$\mathsf{AltCtrl} \equiv \mathsf{Sys}_{\Delta}\, D\, \mathscr{W}$$

where

$$\begin{aligned}
\mathscr{W} &\equiv \dot{\mathbf{y}} = \mathbf{v},\ \dot{\mathbf{v}} \leq \mathbf{a} - g \\
D &\equiv \quad \mathsf{td}(\mathbf{v_{max}}, a_c, \Delta) + \mathsf{sd}(\mathbf{v_{max}} + a_c\Delta) + \mathbf{y_{max}} \leq \mathbf{y}_{\mathsf{ub}} \ \wedge\ \mathbf{a}' = \mathbf{T}_{?} \\
&\quad \vee\ \mathbf{a}' = \mathbf{a}_{\mathsf{min}} \\
\mathsf{td}(\mathbf{v}, \mathbf{a}, \Delta) &\equiv \mathbf{v} \cdot \Delta + \tfrac{\mathbf{a}\Delta^2}{2} \\
\mathsf{sd}(\mathbf{v}) &\equiv -\tfrac{\mathbf{v}^2}{2 \cdot (\mathbf{a}_{\mathsf{min}} - g)} \\
a_c &\equiv \max(0, \mathbf{T}_{?} - g) \qquad \mathbf{a}_{\mathsf{min}} < g
\end{aligned}$$

The approach is similar to the approach of the velocity controller. Each time this controller runs, it checks whether it will be able to stop in time if it issues the maximum breaking acceleration ($\mathbf{a}_{\mathsf{min}}$) the next time the controller runs. The function td computes a conservative upper-bound on the height at the end of Δ time and sd computes the stopping distance assuming $\mathbf{a}_{\mathsf{min}}$ breaking acceleration. We have formally proven that the height controller guarantees that $\mathbf{y}$ never exceeds $\mathbf{y}_{\mathsf{ub}}$, under the assumption that $\mathbf{y_{max}}$ and $\mathbf{v_{max}}$

are bounds on their respective physical variables. Formally,

$$\Box(\mathbf{y} \leq \mathbf{y_{max}} \wedge \mathbf{v} \leq \mathbf{v_{max}}) \vdash \mathsf{AltCtrl} \rightarrow \Box\mathbf{y} \leq \mathbf{y}_{\mathsf{ub}}$$

As with the velocity controller, we can compose the height controller with modules guaranteeing the assumptions that the height controller makes on $\mathbf{y_{max}}$ and $\mathbf{v_{max}}$. Using SYSCOMPOSE, we can easily prove that the composed system guarantees $\Box\mathbf{y} \leq \mathbf{y}_{\mathsf{ub}}$ from the individual proofs, without reasoning simultaneously about the entire composed system and without reproving anything about the individual parts.

Verifying the height controller differs from the velocity controller in two ways. First, the differential equations describing the physical evolution of the system, the controller logic, and therefore the inductive invariant are all more complex. This in turn means that the real arithmetic proof obligations are substantially more intricate. In practice, this means that the existing foundational, nonlinear real arithmetic decision procedure is not able to solve all of the goals, even though the unverified SMT solver Z3 [23] solves all goals quickly. Second, the verification used history variables (omitted from the specification in this chapter for simplicity) to record the value of each physical variable in the last discrete transition. We use these values to describe the safety buffer that the system consumes during the continuous transition. These variables do not change the behavior of the controller in any way; they are used only for reasoning.

Benefits of Composition When verifying our two controllers, the vast majority of the verification effort was devoted to foundationally reasoning about real arithmetic proof obligations. Our composition technique takes a step towards reducing that burden. As a point of comparison with the non-compositional approach, our first implementation of the height controller was monolithic, including all of the code for reasoning about delay compensation (but not sensor error). The result was more complex real arithmetic goals containing larger expressions and more variables. When we verified the height controller compositionally, the arithmetic proof obligations were simpler and, as a result, required less manual proof effort to simplify the goals into a form that the foundational decision procedures could handle. Moreover, verifying the height controller with the noisy sensor was simply a matter of combining the independent proofs using SYSCOMPOSE. This is a promising result considering that the number of variables influences the complexity of the inductive invariant which is directly related to complexity of automatic verification and difficulty of manual arithmetic proofs when automation fails.

Finding Inductive Invariants In general, one of the challenges of formal verification lies in building a suitable inductive invariant, and hybrid systems are no exception. However, we have found that developing the inductive invariant is actually a part of the process of developing the controller. For example, when building the velocity controller, we first built the inductive invariant stating that the thrust is safe until the next time the controller runs. We then built a controller to compute a thrust that will be safe until the next time it runs; this followed naturally from the inductive invariant. This means that we have not found the task of finding an inductive invariant to be an *additional* burden on top of the necessary task of building the controller itself. Instead, we found these two tasks to be naturally related, regardless of whether or not one performs foundational verification.

2.4 From Model to Reality

In addition to modeling and verifying our systems, we also implemented both the velocity and height controllers to run on a 3D Robotics Iris+. Running the system whose model we verify is important because it allows us to experimentally evaluate the gap between the model and the actual system that we run. We can divide this gap into two pieces: the gap due to our model of the physical world (Section 2.4.1), and the gap due to our model of the discrete controller (Section 2.4.2). We conclude by discussing some of the insights that we gained from running our code on an actual quadcopter (Section 2.4.3).

2.4.1 A Small Model in a Big World

The primary gap between the physical world and our model lies in the fact that our model captures only the vertical dimension. In particular, it does not model the orientation of the quadcopter and therefore does not capture the direction of thrust. However, a model that includes attitude is a refinement of the world model that our specifications use. This means that all traces allowed by a model including attitude are also allowed by the world model used in our specifications. As we explained in Section 2.2, this is because our specifications model the world using differential *inequalities* stating that the vertical thrust is upper-bounded by the thrust produced by the motors. This relaxation of the specification means that we must prove properties of a more liberal system, but it allows us to use our results in the more constrained, richer model which includes attitude.

The other discrepancies between our model and the real world are common simplifying assumptions of models for verification purposes. For example, modeling external factors such as wind, air resistance, etc. would be possible by adding extra terms into the differential world description. Finally, our model also relies on the common assumption of instantaneous change of discrete variables such as thrust. In principle, output values such as thrust actually change over a very small amount of time. While this may seem like a reasonable assumption, it nonetheless constitutes a formal discrepancy.

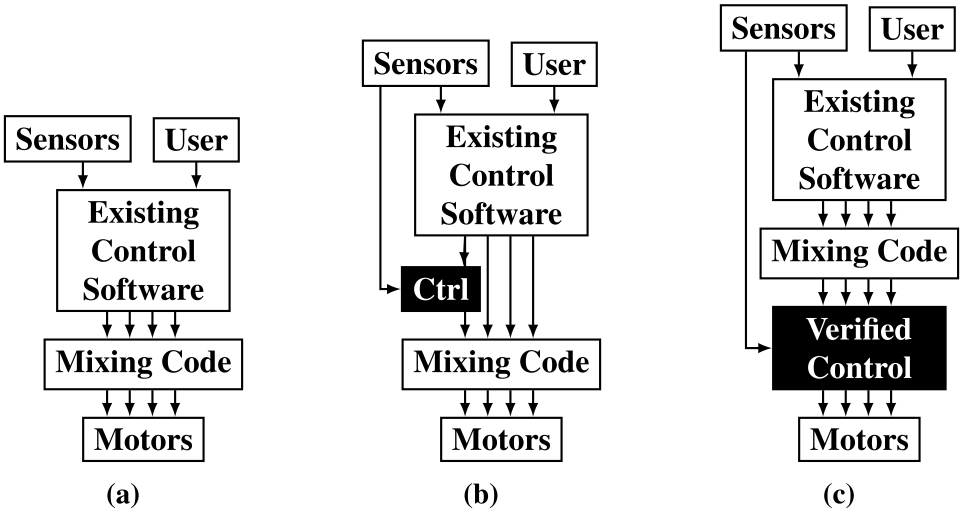


Figure 2.2. A depiction of UAV architecture 2.2a without any modification, 2.2b with our controller running before the motor mixing code, and 2.2b with our controller running after the motor mixing code. Black boxes denote verified code.

2.4.2 From Relations to Bits

Our description of the autopilot architecture in Section 2.1 describes the verified controller as the last piece of code that runs before signals are sent to the motors. This is not strictly necessary. There are advantages and disadvantages to running the verified controller at different points.

Figure 2.2a depicts a version of the ArduPilot architecture without any of our modifications. The existing control software takes input from the user and sensors and outputs a desired throttle, roll torque, pitch torque, and yaw torque to the "motor mixer" module. This module then computes the signals to send to each of the four motors to best approximate the desired behavior. The approximation is necessary because it is not always possible to achieve all four desired values simultaneously since the quadcopter relies on differences between motor speeds to induce non-zero torques.

One place to execute our controller is before the motor mixing module (Figure 2.2b). At this point, our controller executes directly on the desired throttle of the higher-level controller. To meet the interface of our specifications of both the height and velocity controllers, we must convert this input into a desired vertical thrust ($\mathbf{T}_?$). We must also convert the thrust output by the controllers (**a**) into a desired throttle that serves as the input to the mixer module. We accomplish both tasks by multiplying the throttle by a constant which we determined empirically. We discuss the consequence of this choice in Section 2.4.3. As we have already discussed, this will actually provide an upper bound on vertical acceleration, an acceptable input to our controllers. Placing the controller here allows the motor mixer to optimize the engine outputs to achieve the other parameters (attitude torques) as best it can. However, it also requires us to trust that the motor mixer module never exceeds the desired throttle, a property that we believe to hold but have not formally verified.

Running after the motor mixer (Figure 2.2c) allows us to remove the mixer from the trusted computing base. To meet the interface at this level, we must translate between the motor signals and the induced thrust. We again accomplish this using an empirically determined constant that we use to scale each of the motor signals. If the controller rejects the proposed thrust, then it must compute new signals for the motors to induce a safe thrust. There are many ways to achieve a particular total thrust by adjusting four motor signals. In order to minimize the affect of the controller on attitude dynamics, we linearly scale back each of the motor values to achieve the thrust output by the controller.

Regardless of where we insert the controller, the trusted computing base still includes the sensor fusion code that runs on the quadcopter. This code takes input from sensors and computes an estimation of the state (e.g. position, velocity, attitude, etc.) of the vehicle. We use this code to provide bounds ($\mathbf{v_{max}}$ and $\mathbf{y_{max}}$) on physical variables, in essence treating the sensor fusion code as an unverified sensor module. This code is substantially larger and more complex than the motor mixing code and we are interested in applying our techniques to reason about it in the future. We are also trusting our (currently manual) translation of the discrete controller model from LTL to C code. This translation includes picking an appropriate value for Δ, the maximum time between discrete transitions. However, any upper bound suffices since our proofs hold for all Δ. Finally, we ignore the formal gap between real arithmetic used in our models and floating-point arithmetic used in the running code. Future work can investigate closing this formal gap using Coq's library for reasoning about floating point computation [14].

2.4.3 Empirical Results

Evaluating empirical results of this nature is important when exploring models. For example, when we first described our controller logic to an expert pilot he was concerned that disengaging the motors so harshly might have a destabilizing effect on the quadcopter. It was only experimentally that we learned that this was not the case.

Experimentally, both the velocity and the height controller enforce their respective safety properties. The height and velocity controllers allowed the quadcopter to go right up to the provided height or velocity bounds. In some rare cases, the quadcopter went above the bounds, by a small amount, for example about ten centimeters for a height bound of 30 meters. We attribute these small violations to un-modeled forces such as wind, sensor inaccuracy, and inaccuracy in the measured relationship between throttle/motor signals and thrust induced. In fact, we found the measured relationship between signals and thrust to have a significant impact on the behavior of the quadcopter and was perhaps the greatest source of error that we noticed. We were careful to be conservative when measuring these constants; since our controllers both provide upper bounds, we can safely err on the side of constants that provide upper bounds on acceleration. Future work can investigate running our controllers on top of closed-loop acceleration controllers to avoid the need for these empirical constants.

We flew the quadcopter in both loiter and stabilize modes, and also had the quadcopter approach the height and velocity bounds from a variety of velocities and orientations. In loiter mode, the pilot sends desired velocity commands to the vehicle, while in stabilize mode, the pilot commands desired attitudes. In all cases, the controllers enforced their safety properties with rare, small violations, and allowed us to retain control over the quadcopter. We never ran both controllers simultaneously because we have no verification results for this scenario. Fundamentally, to compose the controllers we are obliged to show that the controllers do not conflict with one another, e.g. by requiring different remedial actions. We investigate this further in Chapter 3.

Also, recall that our height controller is conservative in the way that it estimates upward thrust: it assumes that the thrust requested by the higher-level controller would be applied directly in the upward direction, even if the attitude of the quadcopter is not upward. As a result, if the attitude is not level, our height controller assumes that there is a larger upward thrust than really occurs, and so it will engage earlier than it needs to. We noticed this effect experimentally: when approaching the height through a non-level attitude, the quadcopter would stop ascending at a lower height than the actual bound. Furthermore, when the quadcopter is at the height bound with the controller engaged and the upward throttle stick engaged to the maximum, if we start rolling or pitching, the quadcopter will not only move in the x-y direction, but it will also descend slightly, since as the orientation changes, the height controller becomes more conservative.

Finally, we also tried our velocity controller with a small negative velocity as the bound. With the throttle stick engaged to the maximum, this caused the quadcopter to land while allowing us to control other aspects of the flight such as attitude and x-y positioning.

2.5 Discussion

In this section we discuss the motivation for our design decisions. We begin by motivating the use of foundational proof assistants including some anecdotal evidence for the benefits of foundational proofs. Next, we describe the challenge of real arithmetic reasoning within a foundational proof assistant. Finally, we discuss the design of our logic, particularly the decision to use a global state.

Benefits of foundational proof assistants Foundational proof assistants provide strong guarantees: they require all proofs to be done in full formal detail, with an unparalleled attention to every single detail. This attention to detail can find subtle but critical problems that otherwise might go unnoticed. For example, we initially axiomatized differential induction instead of proving it sound within our framework. We used this axiom to "verify" a version of the height controller that was in fact not safe (note also that the height controller itself is not trivial, and it is actually quite easy to get it wrong). This left us in an unfortunate state of blissful ignorance: we had a subtle bug in our controller, but we did not uncover it at first, *even though we were applying formal methods*, because our statement of differential induction was subtly incorrect, and we had not yet done the foundational proof for differential induction. It was only when we attempted to foundationally prove, rather than axiomatize, differential induction that we found our initial phrasing of the proof rule to be unsound. Fixing the statement led us to find several issues with earlier versions of our formalism that appeared reasonable but were, in fact, insufficiently constrained and therefore not correct. These anecdotes underscore the key benefit of foundational proofs: they provide a high level of confidence, as has also been shown experimentally in previous studies [95].

In addition to the foundational guarantees, proof assistants provide very expressive, general purpose logics, which can serve as the foundation of a wide range of interesting theories. We have leveraged the standard library's real analysis theories to reason about real arithmetic and calculus. Future work can also make formal connections with actual controller implementations, as the semantics of C [46] and floating point arithmetic [14] are expressible. Furthermore, as this dissertation emphasizes and future chapters reiterate, using Coq allows us to use powerful, domain specific proof rules without sacrificing soundness. In general, the full power of Coq's rich logic and all of the theories built up in it are at our disposal.

The deductive reasoning style embodied in proof assistants provides useful feedback when verification fails since the user is guiding the proof. When a proof does not work, the user is aware of all of the steps taken in the development and often learns enough from the failed proof to be able to fix the system. This feedback is complementary to counter-examples provided by tools such as Z3 [23], which do not provide foundational proofs for nonlinear real arithmetic. This process of building proofs also makes the developer more aware of the minimal assumptions that a hybrid system is making and the maximal guarantees that it can ensure. In practice, we have found that proofs lead us to find more general, and therefore compositional, interfaces that are easier to satisfy.

Proof assistants also allow us to build automation without sacrificing foundational proofs. In this work, much of that automation surrounds our embedding of temporal logic. This automation is scripted in $\mathscr{L}_{\text{tac}}$, which means that the automation lies completely outside of our trusted computing base. This property has allowed us to rapidly iterate on the automation without needing to worry about its soundness.

Automation for real arithmetic The biggest drawback to the use of foundational proof assistants for formalizing hybrid systems is the lack of good automation for reasoning about real arithmetic. For example, the only nonlinear real arithmetic decision procedure currently implemented for Coq is `psatz` [12] which can be extremely slow, even for simple goals. This is especially true when the goal is unprovable; `psatz` can run for hours before overflowing the stack. Moreover, `psatz` never returns a counter example, making it even more challenging to debug false arithmetic conjectures. This means that we spent a lot of time trying to prove goals that are ultimately unprovable. However, modern SMT solvers such as Z3, Yices, and CVC4 have good support for reasoning about real arithmetic. We eventually started passing real arithmetic goals to Z3 to sanity check them before running them through `psatz`. The value of doing this throughout the development process could easily be measured in hours (or days) of productivity gained. Our composition theorems are another step towards reducing the proof burden by factoring the problem into more manageable pieces. We believe that even more abstraction is possible by codifying the "tricks of the trade" as formal, general-purpose proof rules.

Global state and unconstrained transitions Unlike in a programming language, where `x = y + 1` means that `x` changes and the rest of the variables stays the same, in our logic (like in TLA), $\mathbf{x}' = \mathbf{y} + 1$ says nothing about the behavior of variables other than $\mathbf{x}$ and $\mathbf{y}$. Thus, other variables can transition arbitrarily. This allows other components to execute "in parallel" on their own state simply by joining the programs using logical conjunction. In other words, parallel composition is trivial to express. This, in turn, made it simple to express the definition of $\otimes$ and reason about the composition of our height and velocity controllers with sensor error and delay compensation. We will see further benefits of this approach in Chapter 3.

The primary drawback to this approach is that care must be taken to avoid these programs from interfering with one another in unexpected ways. We avoided this problem by enhancing our formalism with a mechanism for renaming variables in LTL formulas.

When one composes several components, variable renaming must occur at the global level to avoid unintentional conflicts. While this can be very tedious for programming languages, we feel that for specifications like the ones in this dissertation, the benefits outweigh the drawbacks. However, more significantly, Chapter 3 will describe a specific pitfall that occurs when composing components that *intentionally* write to the same set of variables.

2.6 Acknowledgments

This chapter, in full, is adapted from material as it appears in International Conference on Formal Methods and Models for System Design 2015. Ricketts, Daniel; Malecha, Gregory; Alvarez, Mario M.; Gowda, Vignesh; Lerner, Sorin, ACM Press, 2015. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Modular Verification

Chapter 2 introduced two atomic controllers that enforced desired safety properties when run separately. That chapter also described how to compose components, but informally followed the convention that components being composed do not constrain overlapping sets of variables. However, such a restriction is natural in sampled-data systems, for example when composing two controllers outputting to the same sets of actuators. This raises the question: when can two such controllers be run in parallel in order to enforce the conjunction of their respective safety properties? More generally, when can a sampled-data system be built up modularly from smaller components while ensuring the properties guaranteed by each of the components and avoiding inconsistencies in the composed system? This chapter provides a series of proof rules to address that question.

As alluded to above, one of the challenges is in ensuring that the specification of the discrete controller is always enabled, i.e. it always specifies at least one successor state. While this might seem trivial, consider the following scenario. Suppose we have built a module that prevents a quadcopter from exceeding some maximum altitude. Furthermore, suppose we have also built another module that prevents the quadcopter from violating some *minimum* altitude. If we have separately verified that these two modules enforce their respective properties, we would like to compose them in parallel to guarantee both properties simultaneously. That is, we would like the composed system to guarantee that the system never goes too high or too low. However, this is not always possible; a module could enforce the upper bound on altitude by always accelerating downwards. Likewise, a module could enforce the lower bound on altitude by always accelerating upwards. Clearly, naïvely composing the controllers of these modules in parallel would result in a system that gets stuck – there is never an acceleration that both controllers can agree on.

In this chapter, we present sound techniques for resolving this and other potential pitfalls for reusing and composing modules for sampled-data systems. We observed that modularity is facilitated by separating verification into two parts: *preservation* and *progress*. Preservation ensures that the model guarantees the safety property inductively, while progress ensures that the system model is always enabled. This separation facilitates the application of several simple, general, and powerful operators, namely substitution, conjunction, and disjunction. More precisely, we state sufficient conditions for applying these operators to individual modules to produce a new sampled-data system with the desired properties (e.g. the conjunction of the safety properties of conjoined modules). Crucially, these sufficient conditions are in terms of preservation and progress.

To validate the expressiveness of our theorems, as in the rest of the dissertation we apply them in the context of *quadcopters*, by showing how to compose several simple verified controllers together in different ways to produce many different verified composed controllers. To ensure that our controllers are practical, we run them on an actual quadcopter. In summary, the contributions of this chapter are:

- We implement in the Coq proof assistant a general approach for modular verification of sampled-data systems by separately exposing proofs of preservation and progress. The development is available from: https://github.com/ucsd-pl/veridrone/tree/EMSOFT-16.

- We apply this approach to build and verify arbitrary 3D geofences for a quadcopter, including walls, boxes, and rectangular donuts, starting with two simple verified 1D controllers. We show that our modular verification techniques keep the proof burden manageable.
- We evaluate our geofences by running them on an actual quadcopter, and show that they work in practice.
- We discuss the capabilities of three state-of-the-art fully-automated tools (SpaceEx, Flow*, and dReach) in verifying our geofence controllers.

3.1 Overview

We start with an informal description of the operators that our theory covers: substitution, conjunction, and disjunction. We then give an overview of verifying controllers using these operators. Finally, we describe how we applied this to build a verified family of geofences for a quadcopter.

Operators *Substitution* of expressions for variables represents a form of reuse, allowing us to transform systems and their properties into a different coordinate system. For example, given a model of a system defined in the x-y plane, the substitution $\{\mathbf{x} \mapsto \mathbf{r}\cos\phi,\ \mathbf{y} \mapsto \mathbf{r}\sin\phi\}$ transforms the model to polar coordinates, the substitution $\{\mathbf{x} \mapsto \mathbf{y},\ \mathbf{y} \mapsto \mathbf{x}\}$ rotates the system, and the substitution $\{\mathbf{x} \mapsto \mathbf{x}+5\}$ translates the system by 5 units in the $\mathbf{x}$ dimension. However, not all substitutions soundly transport both systems *and* their properties; our theory (Section 3.3.1) gives formal conditions under which substitutions are permitted.

Disjunction of two systems represents nondeterministic choice between the controllers of the system. For example, if we have a controller that prevents a quadcopter from flying too far to the west and another controller that prevents a quadcopter from flying too far north, then their disjunction enforces a north-west no-fly zone – the quadcopter must stay to the north *or* to the west of the no-fly zone. Unlike conjunction, there is no risk of the composed system getting stuck. Instead, the challenge with disjunction is in constraining the nondeterministic choice between the controllers. Our theorems and definitions in Section 3.3.2 make this formal by including the inductive invariants of each system within the composed controller.

Conjunction of two systems represents parallel composition of these systems. For example, if we have a system that enforces an upper bound on velocity and a system that enforces a lower bound on velocity, then their conjunction enforces both an upper and a lower bound on velocity. We can also conjoin systems that control or restrict different variables, such as a system that enforces a bound on velocity and a system that enforces a bound on position. However, as discussed in the introduction, the challenge of applying this operator is in ensuring that the conjoined systems never get stuck, e.g. when the controller of one system requires positive acceleration while the other requires negative acceleration. Again, our theory (Section 3.3.3) gives formal conditions under which conjunctive composition is possible.

Note that disjunctive and conjunctive composition are related to alternative and parallel composition.

Controller Verification To illustrate how the operators work, we explain the construction and modular verification of several general purpose controllers for enforcing state constraints, depicted in Figure 3.1. We begin with a simple verified module: a controller that enforces an upper bound on position in one spatial dimension by controlling acceleration (depicted by (a) in Figure 3.1). We use substitution to “mirror” this module and its correctness property, thus obtaining a module (b) that enforces a lower bound on position,

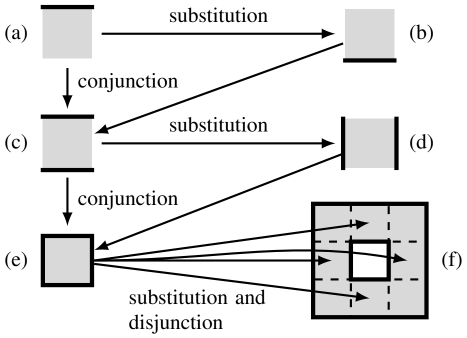


Figure 3.1. Overview of construction and verification of position bounding controllers.

again in one dimension. We conjoin these two modules to form a controller (c) enforcing upper and lower bounds on position, still in one dimension. We use substitution to rotate this interval controller into a second, orthogonal dimension (d), then conjoin (c) and (d) to form a controller (e) enforcing a 2 dimensional rectangle, i.e. upper and lower bounds on position in two dimensions. Finally, we use substitution to build and verify four translated copies of (e) and disjunction of these four copies to enforce a rectangular donut (f). We use disjunction to enforce that the system must, at all times, be in the first copy of (e), the second, the third, *or* the fourth. Moreover, since the rectangles are overlapping, the system can transition from one rectangle to another.

Quadcopter Although the above approach can enforce state constraints for a variety of applications (e.g. trains, intelligent cruise control), we evaluate our approach in the context of *quadcopter* controllers that enforce position and velocity bounds. We performed this verification modularly starting from two simple verified modules that we ported from prior work: one enforcing an upper bound on velocity and another enforcing an upper bound on position, both in one spatial dimension. Connecting the verification methodology above to quadcopters required application of the three operators under the complex, coupled dynamics of a quadcopter, thus showcasing the applicability of our rules to solve complex problems. Crucially, our Coq theorems for each of these operators give formal conditions under which this is sound.

Ultimately, we were able to use the verification techniques in Figure 3.1 to build a three dimensional bounding box of both position and velocity for the quadcopter. This bounding cube provides a powerful building block for constructing "pixelated shapes" (analogous to (f) in Figure 3.1), which can be used to enforce interesting shapes such as a flying around and over but not near the pilot. The results of our verification along with actual flight tests are in Section 3.4. A primary takeaway is that substitution, conjunction, and disjunction are powerful operations that can take simple controllers verified with respect to simple dynamics and turn them into controllers that enforce complex constraints in complex dynamics.

3.2 A Modular Basis for Reasoning

In this section, we present the framework that we use for modular reasoning about sampled-data systems: separating proofs into preservation and progress. This foundation will allow us to build the theory for applying substitution, conjunction, and disjunction (presented in Section 3.3). We start by formally illustrating the difficulty of modular reasoning in this domain. We use the notation and definitions from Chapter 1. As a small note, throughout this section when X is a system, we use $\mathscr{D}_X$ and $\mathscr{W}_X$ to denote the discrete and continuous transitions of X, respectively. Also, we use the inductive invariant of a system as its initial condition; thus we use I to denote an inductive invariant and I_X to denote the inductive invariant of system X. In practice, one must prove that the initial condition of a system implies the inductive invariant.

3.2.1 Stuck Specifications

The physical world always evolves because time always evolves. Cyber-physical system specifications should adhere to this property – the specification should never reach a state in which it is stuck, i.e. in which a transition is impossible. For example, consider the system $\mathsf{Sys}_{\Delta}\ \mathsf{False}\ \mathscr{W}$. In this system, there is never a discrete transition (expressed using the unsatisfiable action formula False). Since a discrete transition never occurs, a continuous transition is not possible once time reaches Δ. Readers familiar with Zeno specifications [1] will note that Sys specifications that are never stuck are non-Zeno.

We rule out such specifications using a new abstraction called System, defined as follows:

$$\mathsf{System}_{\Delta}\ D\ \mathscr{W} \triangleq \mathsf{Sys}_{\Delta}\ D\ \mathscr{W} \vee \neg\mathsf{Enabled}\left(\mathsf{Sys}_{\Delta}\ D\ \mathscr{W}\right)$$

In the above, Enabled takes an action formula and returns a state formula. In particular, $\mathsf{Enabled}(A)$ holds on a given state st iff there exists a next state st' such that $(st, st') \in A$, i.e. the system can take an A transition. A specification whose transition is built using System can never become stuck; if the underlying Sys becomes stuck (not Enabled), then the clause $\neg\mathsf{Enabled}\left(\mathsf{Sys}_{\Delta}\ D\ \mathscr{W}\right)$ conservatively expresses that anything can happen. Informally, we will not be able to prove any interesting global properties of a System when the underlying Sys can reach a state in which it is not Enabled since we will know nothing about the next state.

It may seem trivial to avoid writing stuck specifications for sampled-data systems, and thus the distinction between Sys and System appears to be only theoretical. However, avoiding stuck specifications is a core challenge of building sampled-data systems modularly. To see why, consider the following. In general, our end goal is to prove properties of the form:

$$\vdash I \wedge \Box(\mathsf{System}_{\Delta}\ D\ \mathscr{W}) \rightarrow \Box S$$

This property states that, starting with initial condition I, condition S always holds if at each point in the trace, the transition relation is described by $(\mathsf{System}_{\Delta}\ D\ \mathscr{W})$. Unfortunately, properties like the one above are not modular. For example, suppose we have two discrete transitions D_1 and D_2 which independently ensure S_1 and S_2, i.e.

$$\vdash I \wedge \Box(\mathsf{System}_\Delta\, D_1\, \mathscr{W}) \rightarrow \Box S_1$$

$$\vdash I \wedge \Box(\mathsf{System}_\Delta\, D_2\, \mathscr{W}) \rightarrow \Box S_2$$

We would like to combine these proofs to show that $S_1 \wedge S_2$ is an invariant of the conjoined system $\mathsf{System}_\Delta\, (D_1 \wedge D_2)\, \mathscr{W}$. Unfolding the definition of System reveals that this is not, in general, true. The problem is that $\mathsf{Enabled}\, D_1 \wedge \mathsf{Enabled}\, D_2$ does not necessarily imply $\mathsf{Enabled}\,(D_1 \wedge D_2)$ (Consider $\mathsf{Enabled}\,(\mathbf{x}' = 1 \wedge \mathbf{x}' = 0)$). This means that the following two formulas are not necessarily equivalent:

$$\mathsf{System}_\Delta\, D_1\, \mathscr{W} \wedge \mathsf{System}_\Delta\, D_2\, \mathscr{W}$$

$$\mathsf{System}_\Delta\, (D_1 \wedge D_2)\, \mathscr{W}$$

This formalizes the challenge described in the introduction – naïve parallel composition (conjunction) of controllers can result in a controller that gets stuck.

Crucially, $\mathsf{Enabled}$ is inherently non-modular, so global invariant proofs of systems specified using System are inherently non-modular. By ruling out stuck specifications, we also rule out the modularity of *global* invariant proofs.

3.2.2 Regaining Modularity

The key to regaining modularity is a shift from global proofs to local ones. In particular, we will make the inductive invariant of the system explicit and use it to prove two properties independently: preservation of the invariant, and progress of the system under the invariant. As we will see in Section 3.3, this decomposition of the global property into local ones makes it much easier to combine and re-use systems and their proofs.

Preservation Preservation of property (I) under an action formula states if I holds in the current state then it holds in the next state. Formally,

$$\mathsf{SysPreserves}\, I \left(\mathsf{Sys}_\Delta\, D\, \mathscr{W}\right) \triangleq I \wedge \mathsf{Sys}_{\mathsf{inv}} \wedge \mathsf{Sys}_\Delta\, D\, \mathscr{W} \rightarrow I'$$

where I' represents the state formula I with all variables primed. The $\mathsf{Sys}_{\mathsf{inv}}$ premise expresses the invariants guaranteed by the Sys abstraction, namely that no more than Δ time elapses between discrete transitions.

Progress Progress under an invariant justifies that the system is Enabled assuming the invariant. Formally,

$$\begin{aligned} &\mathsf{SysProgress}\, I\ (\mathsf{Sys}_\Delta\, D\, \mathscr{W}) \triangleq \\ &\qquad I \wedge \mathsf{Sys}_{\mathsf{inv}} \rightarrow \mathsf{Enabled}\ (\mathsf{Sys}_\Delta\, D\, \mathscr{W}) \end{aligned}$$

This condition allows us to prove that a Sys and a System describe exactly the same system.

Note that, here, progress is a safety property that is closely related to the notion of progress in programming languages. It is different than progress properties in distributed systems, and it is different than convergence to an equilibrium in control theory.

Combining Preservation & Progress Preservation of and progress under the *same* inductive invariant is sufficient to prove that the invariant is a global invariant of the corresponding System, which is ultimately our goal. This is captured by the following theorem:

Theorem 3.1. LOCALTOGLOBAL

$$\begin{array}{ll} & \mathsf{SysPreserves}\, I\ (\mathit{Sys}_\Delta\ D\ \mathscr{W}) \\ \wedge & \mathsf{SysProgress}\, I\ (\mathit{Sys}_\Delta\ D\ \mathscr{W}) \\ \wedge & I \to S \\ \vdash & I \wedge \Box(\mathsf{System}_\Delta\ D\ \mathscr{W}) \to \Box S \end{array}$$

3.3 Modular Sampled-data Systems

In this section, we show how to use preservation and progress to reason modularly about sampled-data systems. In particular, for each of our three operators (substitution, disjunction, and conjunction), we present theorems that state formal conditions under which application of the operator guarantees preservation and progress. We illustrate each of the operators and corresponding theorems by building verified state-constraining controllers for quadcopters. This allows us to construct and verify controllers enforcing policies such as “do not fly above 400 feet” (FAA regulation for recreational drones), “do not fly within 5 miles of an airport”, and “do not fly within 5 feet of the pilot.”

It is important to note that all of the state-constraining controllers that we verify are *non-deterministic*. This means that the discrete transitions do not compute a single value for each control variable (e.g. acceleration) but instead describe a set of allowed values that ensure the desired state-constraint. As we will see, this non-determinism is crucial for conjunctive composition (Section 3.3.3). In Section 3.4, we discuss how the actual implementation of these controllers resolves this non-determinism.

Building blocks As the basic building blocks of our development, we start with the two sampled-data systems that are minor variants on the ones from Chapter 2. Both

First-Derivative Controller ($\mathsf{M}_{\partial} = \mathsf{Sys}_{\Delta}\; D_{\partial}\; W_{1D}$)

$$\begin{aligned} D_{\partial} &\triangleq \; \mathscr{C}_a{}' \wedge (\mathbf{a}' \cdot \Delta + \mathbf{v} \leq ub \vee \mathbf{a}' \leq 0) \\ I_{\partial} &\triangleq \; (\mathbf{a} < 0 \rightarrow \mathbf{v} \leq ub) \wedge (\mathbf{a} \geq 0 \rightarrow \mathbf{a} \cdot \boldsymbol{\tau} + \mathbf{v} \leq ub) \end{aligned}$$

Second-Derivative Controller ($\mathsf{M}_{\partial^2} = \mathsf{Sys}_{\Delta}\; D_{\partial^2}\; W_{1D}$)

$$\begin{aligned} D_{\partial^2} \triangleq \; & (0 \leq \mathbf{v} + \mathbf{a}' \cdot \Delta \rightarrow \\ & \quad \mathsf{td}(\mathbf{v}, \mathbf{a}', \Delta) + \mathsf{sd}(\mathbf{v} + \mathbf{a}' \cdot \Delta) + \mathbf{y} \leq \mathbf{y}_{\mathsf{ub}}) \\ & (\mathbf{v} + \mathbf{a}' \cdot \Delta \leq 0 \wedge 0 < \mathbf{v} \rightarrow \\ & \quad \mathsf{td}(\mathbf{v}, \mathbf{a}', \tfrac{-\mathbf{v}}{\mathbf{a}'}) + \mathbf{y} \leq \mathbf{y}_{\mathsf{ub}}) \wedge \mathscr{C}_a{}' \\ I_{\partial^2} \triangleq \; & \forall t : \mathbb{R}, 0 \leq t \leq \boldsymbol{\tau} \rightarrow \\ & \quad \mathbf{y} + \mathsf{td}(\mathbf{v}, \mathbf{a}, t) + \mathsf{sd}(\max(0, \mathbf{v} + \mathbf{a} \cdot t)) \leq \mathbf{y}_{\mathsf{ub}} \end{aligned}$$

$$\begin{aligned} \mathsf{td}(\mathbf{v}, a, \Delta) &\triangleq \mathbf{v} \cdot \Delta + \tfrac{a \cdot \Delta^2}{2} & \mathsf{sd}(\mathbf{v}) &\triangleq -\tfrac{\mathbf{v}^2}{2 \cdot a_{\mathsf{min}}} \\ \mathscr{C}_a &\triangleq \mathbf{a}_{\mathsf{min}} \leq \mathbf{a} & \mathbf{a}_{\mathsf{min}} &< 0 \end{aligned}$$

Figure 3.2. Discrete transitions and inductive invariants for our building blocks.

operate in one spatial dimension, i.e.

$$\mathscr{W}_{1D} \;\; \triangleq \;\; \dot{\mathbf{y}} = \mathbf{v} \wedge \dot{\mathbf{v}} = \mathbf{a} \wedge \dot{\mathbf{a}} = 0$$

The two controllers each enforce constant bounds on a state variable by controlling acceleration (**a**). The first-derivative controller (M_{∂}) bounds velocity using acceleration (the first-derivative of velocity). The second-derivative controller (M_{∂^2}) bounds position using acceleration (the second-derivative of position). To ensure that M_{∂^2} can stop before violating the boundary, the controller is parameterized by $\mathbf{a}_{\mathsf{min}}$ which represents the braking acceleration and smallest possible acceleration (and is negative). Figure 3.2 gives the discrete transitions and inductive invariants for the two systems. Each inductive invariant states that, given the time until the next discrete transition, the system can stop before the boundary.

In Chapter 2, we verified variants on both of these controllers in a global style

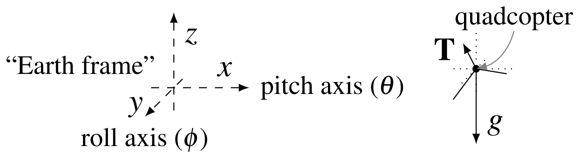


Figure 3.3. Free-body diagram and dynamics of the quadcopter.

but did *not* compose them. For the work in this chapter, we ported each of the global proofs to our local, modular specification by extracting the inductive invariant (which was stated explicitly in the proof) and the preservation proof (which formed the inductive case). Beyond extracting the safety proofs, we also had to verify progress, which was not addressed in Chapter 2, but is trivial for such basic modules. In the remainder of this section, we denote the preservation and progress proofs of the two controllers by: ∂-PRESERVES, ∂-PROGRESS, ∂^2-PRESERVES, and ∂^2-PROGRESS.

Quadcopter Controller To build and verify controllers for quadcopters, we need a model of the physical dynamics of a quadcopter, called $\mathscr{W}_{QC}$:

$$\mathscr{W}_{QC} \triangleq \left(\begin{array}{ll} & \dot{\mathbf{x}} = \mathbf{v_x} \wedge \dot{\mathbf{y}} = \mathbf{v_y} \wedge \dot{\mathbf{z}} = \mathbf{v_z} \\ \wedge & \dot{\mathbf{v_x}} = \mathbf{T} \cos\phi \sin\theta \\ \wedge & \dot{\mathbf{v_y}} = -\mathbf{T} \sin\phi \\ \wedge & \dot{\mathbf{v_z}} = \mathbf{T} \cos\phi \cos\theta - g \\ \wedge & \dot{\phi} = 0 \wedge \dot{\theta} = 0 \wedge \dot{\mathbf{T}} = 0 \end{array} \right)$$

Here **T** represents the combined thrust of the motors (normalized with respect to the mass of the quadcopter), θ represents the pitch (the angle around the y-axis), and ϕ represents the roll (the angle around the x-axis). Figure 3.3 depicts this graphically in a free-body diagram. Our model is based on the simplifying assumption (called the "small angle condition") that a trusted attitude controller can instantaneously achieve any pitch and roll within the bounds -30° to 30° with a thrust greater than or equal to 0, while holding yaw constant at 0.

$$\mathscr{C}_{\theta\phi} \triangleq |\theta| \leq 30^\circ \wedge |\phi| \leq 30^\circ \wedge 0 \leq \mathbf{T}$$

Prior work has suggested that this is a reasonable approximation under this small-angle condition ($\mathscr{C}_{\theta\phi}$), since the attitude dynamics are significantly faster than the velocity and position dynamics [30]. We capture this condition by requiring that all quadcopter controllers are enabled under $\mathscr{C}_{\theta\phi}$. That is, our goal is to build controllers D such that

$$\begin{aligned}
&\mathsf{SysPreserves}\, I\, (\mathsf{Sys}_\Delta\ (D \wedge \mathscr{C}_{\theta\phi}{}')\ \mathscr{W}_{QC}) \wedge \\
&\mathsf{SysProgress}\ I\ (\mathsf{Sys}_\Delta\ (D \wedge \mathscr{C}_{\theta\phi}{}'))\ \mathscr{W}_{QC})
\end{aligned}$$

For some state-constraints and their corresponding controllers, it is only necessary to reason about an abstraction of the quadcopter dynamics $\mathscr{W}_{QC}$. For example, reasoning about a controller that enforces a bound on the vertical position $\mathbf{z}$ might only require reasoning about the portion of the dynamics on which $\mathbf{z}$ depends. We formalize this with:

$$\begin{aligned}
&\mathsf{SysPreserves}\, I\, (\mathsf{Sys}_\Delta\ (D \wedge \mathscr{C}_{\theta\phi}{}'))\ \mathscr{W}) \wedge \\
&\mathsf{SysProgress}\ I\ (\mathsf{Sys}_\Delta\ (D \wedge \mathscr{C}_{\theta\phi}{}')\ \mathscr{W}) \\
&(\mathscr{W}_{QC} \rightarrow \mathscr{W}) \wedge \\
\vdash\ &\mathsf{SysPreserves}\, I\, (\mathsf{Sys}_\Delta\ (D \wedge \mathscr{C}_{\theta\phi}{}')\ \mathscr{W}_{QC}) \wedge \\
&\mathsf{SysProgress}\ I\ (\mathsf{Sys}_\Delta\ (D \wedge \mathscr{C}_{\theta\phi}{}')\ \mathscr{W}_{QC})
\end{aligned}$$

where $\mathscr{W}_{QC} \rightarrow \mathscr{W}$ states that $\mathscr{W}$ is an abstraction of $\mathscr{W}_{QC}$.

3.3.1 Reuse via Substitution

Substitution of expressions for variables is a simple but powerful operator that allows us to reuse controllers and their properties. For example, substitution allows us to perform familiar geometric transformations such as translations, reflections, scaling, and rotations. In addition, substitution allows us to project simple dynamics onto more complex dynamics; a technique we use to build verified state-constraining controllers for the quadcopter. We will explain the general technique by using it to transport (re-use) the second-derivative controller (M_{∂^2}), and its safety proof, to enforce a maximum altitude for our quadcopter.

For a formula P and substitution σ (map from variables to expressions), the semantic definition of substitution is:

$$tr \models \{\sigma\}P \quad \triangleq \quad \{\sigma\}tr \models P$$

which states that a substituted formula ($\{\sigma\}P$) holds on a trace (tr) if the formula (P) holds on the renamed trace ($\{\sigma\}tr$). Under this definition, application of substitution always guarantees preservation:

Theorem 3.2. SUBSTPRESERVES

$$\frac{\vdash \mathsf{SysPreserves}\, I\, S}{\vdash \mathsf{SysPreserves}\left(\{\sigma\}I\right)\left(\{\sigma\}S\right)}$$

This proof rule allows us to easily transport ∂^2-PRESERVES to the quadcopter. For example, using it we can conclude

$$\vdash \mathsf{SysPreserves}\left(\{\sigma_{\partial^2\to QC}\}I_{\partial^2}\right)\left(\{\sigma_{\partial^2\to QC}\}\mathsf{M}_{\partial^2}\right)$$

$$\sigma_{\partial^2\to QC} \equiv \{\mathbf{a} \mapsto \mathbf{T}\cos\phi\cos\theta - g,\ \mathbf{y} \mapsto \mathbf{z},\ \mathbf{v} \mapsto \mathbf{v_z}\}$$

Note that the first argument to $\mathsf{SysPreserves}$ is the inductive invariant for the new system, and can be read directly from the conclusion of the preservation theorem. This is the case for all of our preservation theorems.

Next, we need to justify the progress of the substituted system. The interaction between substitution and progress is a bit subtle because substitutions can introduce coupling between values that were uncoupled before the substitution. For example, $(\mathbf{x}' = 1 \wedge \mathbf{y}' = 0)$ is $\mathsf{Enabled}$ while $\{\mathbf{x} \mapsto \mathbf{z},\ \mathbf{y} \mapsto \mathbf{z}\}(\mathbf{x}' = 1 \wedge \mathbf{y}' = 0)$, which equals $\mathbf{z}' = 1 \wedge \mathbf{z}' = 0$, is not.

However, we can prove that *invertible* substitutions preserve progress. This is because the inverse of the substitution is a function for computing the $\mathsf{Enabledness}$ witness for the substituted system from the $\mathsf{Enabledness}$ witness for the original system. Because of this, we can actually state a stronger progress theorem for substitution, which captures the fact that the inverse substitution preserves known constraints on $\mathsf{Enabledness}$ witnesses, such as $\mathscr{C}_a$ for the first and second derivative controllers. As we will see, this is crucial for proving the small-angle constraint ($\mathscr{C}_{\theta\phi}$). Formally,

Theorem 3.3. SUBSTPROGRESS *For all formulas S, state formulas Q and R, and substitutions σ, if there exists a σ^{-1} such that $R \vdash (\sigma \circ \sigma^{-1})\mathbf{x} = \mathbf{x}$ for all variables $\mathbf{x}$ that occur primed in S, and if $R \vdash \{\sigma^{-1}\}Q$ then*

$$\frac{\vdash \mathsf{SysProgress}\, I\ (S \wedge R')}{\vdash \mathsf{SysProgress}\ \big(\{\sigma\}I\big)\ \big(\{\sigma\}S \wedge Q'\big)}$$

When we apply this theorem to prove the progress of the quadcopter altitude controller, the following inverse works:

$$\sigma^{-1}_{\partial^2 \to QC} \triangleq \phi \mapsto 0, \theta \mapsto 0, \mathbf{T} \mapsto \mathbf{a} + g, \mathbf{z} \mapsto \mathbf{y}, \mathbf{v_z} \mapsto \mathbf{v}$$

We instantiate Q with $\mathscr{C}_{\theta\phi}$ and R with $\mathscr{C}_a$ to guarantee

$$\mathsf{SysProgress}\ I\ \left(\mathsf{Sys}_{\Delta}\ (\{\sigma_{\partial^2\to QC}\}D_{\mathsf{M}_{\partial^2}} \wedge \mathscr{C}_{\theta\phi}{}')\ (\{\sigma_{\partial^2\to QC}\}\mathscr{W}_{1D})\right)$$

Finally, as noted above, we need to prove that $\mathscr{W}_{QC} \to \{\sigma_{\partial^2\to QC}\}\mathscr{W}_{1D}$, i.e. the continuous dynamics produced by the substitution is an abstraction of the full quadcopter dynamics. This reasoning involves standard substation within differential equations, which we have formalized and proved sound in Coq, and mechanical arithmetic reasoning.

Enforcing Planar Boundaries Using our two substitution theorems, we can map the first- and second-derivative controllers onto the quadcopter dynamics in many ways, allowing us to verify many properties with relatively little effort. We showed how to use it to implement an upper bound on altitude. In general, we can use substitution on the second derivative controller to enforce that the quadcopter stays on one side of any 2D plane in 3D space. For example, we can enforce a maximum-west boundary to prevent the quadcopter from flying into a building. Similarly, by applying substitution to the first-derivative controller we can place bounds on velocity in any direction. The key is that our two substitution theorems allow us to transfer the correctness of any controller to work on a new dynamics that can be constructed from an invertible substitution.

3.3.2 Disjunctive Composition

In this section we present rules to compose systems using disjunction. For example, suppose that we wish to enforce a rectangular no-fly zone centered around the origin (depicted in Figure 3.4). A system can avoid the no-fly zone if at all times it is to the north, the south, the east, *or* to the west of the rectangle. We can build such a system by disjoining subsystems M_N, M_S, M_E, and M_W, which are each built from a substitution applied to M_{∂^2} to enforce the northern, southern, eastern, and western boundaries of the box, respectively. As we will see, separately exposing preservation and progress allows us to define a disjunction operator that is fully compositional and that permits the system to transition from an inductive invariant of one subsystem to another (e.g. north of the no-fly zone to west of the no-fly zone) during a single trace; this would not be possible with global invariant proofs.

The disjunctive composition of two systems is defined by the $\oplus$ operator, which is indexed by the inductive invariants of the two systems. Formally,

Definition 3.1. *Disjunctive composition*

$$\begin{aligned}(\mathsf{Sys}_\Delta\, D_1\, \mathscr{W})\ {}^{I_1}\!\oplus^{I_2} (\mathsf{Sys}_\Delta\, D_2\, \mathscr{W}) &\triangleq \\ \mathsf{Sys}_\Delta\, \big((I_1 \wedge D_1) \vee (I_2 \wedge D_2)\big)\, \mathscr{W}\end{aligned}$$

The inclusion of the *inductive* invariants in this definition is essential to enforce the disjunction of the properties. To see why, consider our example and suppose the system is currently along the western edge of the no-fly zone. Since the system is outside of the inductive invariant of the eastern controller, the eastern controller could allow the system to do anything, including moving east into the no-fly zone. Thus, at each discrete transition, the composed controller must consider the discrete transitions of a sub-controller whose inductive invariant currently holds. This guarantees that the inductive invariant of that subsystem holds after the transition. Moreover, it means that the system can transition from one inductive invariant to another, only where the inductive invariants overlap.

This definition of disjunction is fully compositional with both SysPreserves and SysProgress. In particular, the disjunctive composition of two systems that independently preserve I_A and I_B preserves $I_A \vee I_B$:

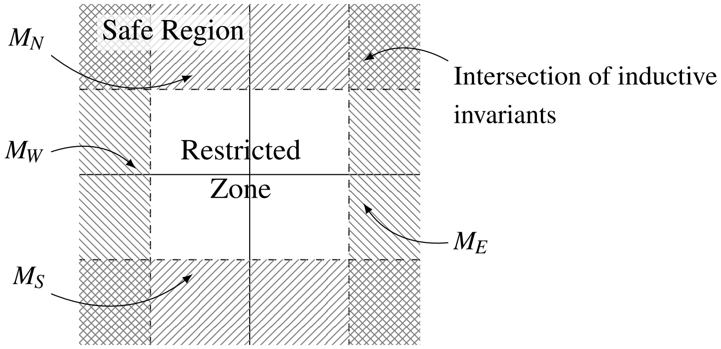


Figure 3.4. Staying out of restricted airspace using the disjunction of four controllers.

Theorem 3.4. DISJOINPRESERVES

$$\begin{aligned} & \mathsf{SysPreserves}\, I_A\, A \;\wedge\; \mathsf{SysPreserves}\, I_B\, B \\ \vdash\;\; & \mathsf{SysPreserves}\, \big(I_A \vee I_B\big)\, \big(A \;{}^{I_A}\!\oplus^{I_B}\, B\big) \end{aligned}$$

Using only this theorem we can easily construct a proof that the disjunctive composition of our four controllers enforces the no-fly zone property.

A similar theorem states that $\oplus$ guarantees SysProgress.

Theorem 3.5. DISJOINPROGRESS

$$\begin{aligned} & \mathsf{SysProgress}\, I_A\, A \;\wedge\; \mathsf{SysProgress}\, I_B\, B \\ \vdash\;\; & \mathsf{SysProgress}\, (I_A \vee I_B)\, \big(A \;{}^{I_A}\!\oplus^{I_B}\, B\big) \end{aligned}$$

The proof follows from the fact that I_A ensures the Enabledness of A and I_B ensures the Enabledness of B. Since the new inductive invariant $(I_A \vee I_B)$ ensures that at least one of I_A or I_B holds, at least one of A or B must be Enabled.

Disjunction is a very powerful composition mechanism that is applicable in a wide variety of circumstances. For example, we can use it to guarantee that a train can only have a high velocity when it is not in a curve by composing a maximum velocity controller with a controller that stops the train before curves. A controller such as this one could have prevented the Amtrak derailment in Philadelphia in 2015 that killed 8 people and injured 200.

3.3.3 Conjunctive Composition

In this section, we present rules to compose systems to ensure the conjunction of their properties. For example, one might want to ensure that a system's upward velocity does not exceed 1 m/s *and* that the system stays below 100m by conjoining the first and second derivative controllers. Unlike disjunctive composition, conjunctive composition of two systems satisfying progress does not guarantee a system satisfying progress, due to coupling. However, our ability to separately prove progress allows us to push forward. We show that, even in coupled domains, conjunctive composition can lead to substantial savings in proof effort, and demonstrate some clever tricks that allow us to decouple domains that, on the surface, seem intricately linked.

We use the same definition for conjoining systems as in Chapter 2. We restate the definition here for clarity:

Definition 2.1. *Conjunctive composition*

$$\mathsf{Sys}_\Delta\, D_1\, \mathscr{W}_1 \otimes \mathsf{Sys}_\Delta\, D_2\, \mathscr{W}_2 \equiv \mathsf{Sys}_\Delta\, \left(D_1 \wedge D_2\right)\, \left(\mathscr{W}_1 \wedge W_2\right)$$

The crucial feature of this definition is how it interacts with $\mathsf{SysPreserves}$. Note that while disjunctive composition requires that both systems have the same continuous transition, conjunctive composition does not impose this restriction.

Theorem 3.6. CONJOINPRESERVES

$$\begin{aligned} & (I_B \to \mathsf{SysPreserves}\, I_A\, A) \;\wedge\; (I_A \to \mathsf{SysPreserves}\, I_B\, B) \\ \vdash\; & \mathsf{SysPreserves}\,\big(I_A \wedge I_B\big)\,\big(A \otimes B\big) \end{aligned}$$

Intuitively, if we start in a state satisfying both I_A and I_B, A guarantees that we stay in I_A, and B guarantees that we remain in I_B, then if both A and B hold, we must remain in the intersection of I_A and I_B. Note that Theorem 3.6 removes the restriction of acyclic parallel composition from Theorem 2.2 in Chapter 2. This is possible because the inductive invariants are made explicit.

The difficulty of conjunctive composition lies in justifying progress. Even though A and B may independently be Enabled under the inductive invariant, there is no guarantee that their conjunction is Enabled. For example, suppose that we wish to compose an overly-conservative upper-bound controller that insists on a negative acceleration and a similarly conservative lower-bound controller that insists on a positive acceleration. Since acceleration can not be simultaneously positive and negative, the conjunction of these controllers does not satisfy progress.

Nevertheless, all is not lost when conjoining two systems. There are a variety of techniques for proving Enabledness of conjunctions. We will illustrate these techniques through a sequence of examples, ultimately culminating in a 3D bounding box for both position and velocity.

Example: Staying within an Interval Consider constructing a controller that enforces both an upper and a lower bound on both position (y) and velocity (v) in a single spatial dimension. We can build such a controller (which we call Int) using $\otimes$ and an application of our substitution operator to the second- and first-derivative controllers:

$$\begin{aligned}\mathsf{Int} &\equiv \mathsf{M}_{\partial^2} \otimes \{\sigma_-\}\mathsf{M}_{\partial^2} \otimes \mathsf{M}_{\partial} \otimes \{\sigma_-\}\mathsf{M}_{\partial} \\ \sigma_- &\equiv \{\mathbf{y} \mapsto -\mathbf{y},\, \mathbf{v} \mapsto -\mathbf{v},\, \mathbf{a} \mapsto -\mathbf{a}\}\end{aligned}$$

Here, the σ_- substitution mirrors the controller's logic so that rather than enforcing an upper bound on $\mathbf{y}$ of $\mathbf{y}_{\mathsf{ub}}$ (resp. v_y of v_{ub}), the substituted controller enforces a lower bound on $\mathbf{y}$ of $-\mathbf{y}_{\mathsf{ub}}$ (resp. v_y of $-v_{\mathsf{ub}}$).

The preservation of this composition follows immediately from ConjoinPreserves, ∂^2-Preserves, ∂-Preserves, and SubstPreserves. However, since conjoined systems are not guaranteed to satisfy progress, we must prove this separately. Formally, we must prove progress of Int under the conjunction of the inductive invariants of the subsystems: [1]

$$\mathsf{SysProgress}\ \left(I_{\partial^2} \wedge \{\sigma_-\}I_{\partial^2} \wedge I_{\partial} \wedge \{\sigma_-\}I_{\partial}\right)\ \mathsf{Int}$$

Informally, this states that, under the inductive invariant, there exists an action that is acceptable to all of the (non-deterministic) controllers. More formally, we must justify that there does not exist any state satisfying the inductive invariant for which there is not an action that all controllers accept. Unfolding the definitions and applying the substitution (σ_-) reveals that the progress of Int reduces to first-order reasoning over real arithmetic. At first glance, it may seem as though modularity was a failure here; however, by separating preservation and progress, the non-modularity of progress did not prevent us from modularly proving preservation. Moreover, our split crucially allows us to assume the inductive invariant when proving progress.

[1] For brevity, I_{∂^2} and I_{∂} refer to the inductive invariants of M_{∂^2} and M_{∂}.

This is in contrast to other work [4] where conjunctive (parallel) composition is only allowed when the individual modules output to disjoint sets of variables. Furthermore, we found the proof of SysProgress I_{Int} Int to be *less than one third* the size of the proof of preservation of the individual second-derivative controller M_{∂^2}. This means that $\otimes$ greatly reduces the cost of conjoining systems, even when these systems are tightly coupled.

It is important to note that under-specification of controllers is essential to composition in this case. If both systems were completely specified then the two systems would be forced to make identical choices in order for them to compose. By composing under-specified controllers, we retain the flexibility to find a solution that reconciles the restrictions of both controllers.

In certain cases, our proof rules can be used to verify progress compositionally. To demonstrate this, we turn to the task of using our interval controller to modularly build a bounding rectangular prism. We approach the problem in three steps. First, we compose two interval controllers to enforce a bounding box in two spatial dimensions. In the next section, we adapt this controller to the quadcopter by incorporating the small-angle constraint. Finally, we apply the same technique to extend the 2D box into a 3D prism.

Example: Conjunction of Independent Systems We construct the box by conjoining two instances of Int, using substitution to map them to the x- and z-dimensions respectively.

$$\begin{aligned} \mathsf{Box} &\equiv \{\sigma_x\}\mathsf{Int} \otimes \{\sigma_z\}\mathsf{Int} \\ \sigma_x &\equiv \{\mathbf{y} \mapsto \mathbf{x},\ \mathbf{v} \mapsto \mathbf{v_x},\ \mathbf{a} \mapsto \mathbf{a_x}\} \\ \sigma_z &\equiv \{\mathbf{y} \mapsto \mathbf{z},\ \mathbf{v} \mapsto \mathbf{v_z},\ \mathbf{a} \mapsto \mathbf{a_z} - g\} \end{aligned}$$

Verifying that Box enforces a bounding box is simply a matter of using CONJOINPRESERVES, SUBSTPRESERVES, and the preservation proof of Int.

As others have noted [4], progress of conjoined systems is compositional, if the two systems output to disjoint sets of variables. In our logic, the output variables of a formula are the next-state (primed) variables, whose disjointness is expressed by $A \perp' B$. Formally,

Theorem 3.7. ConjoinProgressDisjoint *For all systems A and B such that $A \perp' B$, and for all state formulas I_A and I_B,*

$$\begin{aligned} & \textsf{SysProgress}\ I_A\ A\ \wedge\ \textsf{SysProgress}\ I_B\ B \\ \vdash\ & \textsf{SysProgress}\ \left(I_A \wedge I_B\right)\ \left(A \otimes B\right) \end{aligned}$$

While we can not express disjointness of primed variables ($A \perp' B$) formally within our temporal logic, we can capture it in Coq's logic. Further, in practice the disjointness of two formulas is easily decidable in many cases meaning that discharging this side condition is often trivial. From this theorem and variable disjointness, Box satisfies progress for the dynamics with independent $\mathbf{a_x}$ and $\mathbf{a_y}$.

Now suppose that instead of rectangular dynamics with independent control inputs $\mathbf{a_x}$ and $\mathbf{a_y}$, we want a controller for polar coordinates with independent control inputs $\mathbf{a}$ and θ. For example, these are the dynamics of a 2D version of the quadcopter (with ϕ fixed at 0), in the absence of the small-angle constraint. While the transformation to polar coordinates seems to couple the x and z instantiations of Int, there is always an invertible map from rectangular to polar coordinates. This connection between polar and rectangular coordinates allows us to use the substitution theorems from Section 3.3.1 to prove both preservation and progress for a version of Box that operates using polar thrust.

Note that to arrive at this result, we first used disjoint composition to justify the syntactically independent system satisfies progress. We then applied our substitution theorems and the invertibility of the rectangular-to-polar transformation to show that

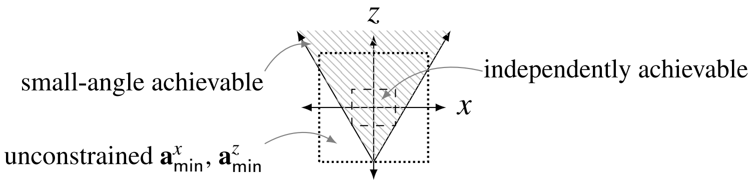

z
x
small-angle achievable
independently achievable
unconstrained $\mathbf{a}^x_{\mathsf{min}}$, $\mathbf{a}^z_{\mathsf{min}}$

Figure 3.5. Decoupling of $\mathbf{a_x}$ and $\mathbf{a_z}$.

the resulting system was the same as the quadcopter dynamics modulo the small-angle constraint. The takeaway is that although the system ($\{\sigma_\theta\}\mathsf{Box}$) is not superficially composed of disjoint subsystems, we can still verify both progress and preservation fully modularly.

Example: Incorporating the Small-angle Constraint The previous example ignored the small-angle constraint, which is critical to the accuracy of our model of the quadcopter's dynamics. In this section we address this shortcoming and complete the verification of Box with respect to a 2D version of $\mathscr{W}_{QC}$ (again, with ϕ fixed at 0) *in a fully modular way*. Figure 3.5 shows how the small-angle constraint introduces coupling into the independent box. In particular, the hatched region corresponds to the accelerations achievable under the small-angle constraint while the unconstrained region corresponds to the accelerations necessary for Box developed in the previous section.

Our approach relies on a detail of the verification which we glossed over in the previous presentation. In particular, because the second-derivative controller is *parameterized by* $\mathbf{a}_{\mathsf{min}}$, Int is parameterized by an $\mathbf{a}_{\mathsf{min}}$ and Box is parameterized by a pair of $\mathbf{a}_{\mathsf{min}}$'s in the x and z dimensions. The right side of Figure 3.5 shows how we leverage the parameterization of Box to logically decouple the two dependent dimensions. Our insight is the following. If we can pick values for $\mathbf{a}^x_{\mathsf{min}}$ and $\mathbf{a}^z_{\mathsf{min}}$ (note that $\mathbf{a}^x_{\mathsf{min}}$ and $\mathbf{a}^z_{\mathsf{min}}$ are negative) such that all accelerations a_x and a_z where $|a_x| \leq -\mathbf{a}^x_{\mathsf{min}}$ and $|a_z| \leq -\mathbf{a}^z_{\mathsf{min}}$ are achievable by $\mathbf{T}$ and θ under the small-angle constraint then the values of $\mathbf{a_x}$ and $\mathbf{a_z}$

can be chosen independently *if they fall within the bounds*. Purely trigonometric reasoning reveals that the constraint on θ is achieved for any values of $\mathbf{a_x}$ and $\mathbf{a_z}$ satisfying $\mathscr{C}_x \wedge \mathscr{C}_z$ defined as:

$$\mathscr{C}_x \equiv -(\mathbf{a}_{\mathsf{min}} + g)\tan 30^\circ \leq \mathbf{a_x} \leq (\mathbf{a}_{\mathsf{min}} + g)\tan 30^\circ$$

$$\mathscr{C}_z \equiv \mathbf{a}_{\mathsf{min}} \leq \mathbf{a_z} \leq -\mathbf{a}_{\mathsf{min}}$$

We can formalize this reasoning by noting that the previous example (Box), instantiated with the above choices of $\mathbf{a}^x_{\mathsf{min}}$ and $\mathbf{a}^z_{\mathsf{min}}$, satisfies progress under the constraint $\mathscr{C}_x \wedge \mathscr{C}_z$. Thus, we can use the substitution proof rule SUBSTPROGRESS with a rectangular-to-polar substitution, instantiating R with $(\mathscr{C}_x \wedge \mathscr{C}_z)$ and Q with the small-angle condition (ignoring ϕ since it is assumed to be 0 in this 2D example).

The important point of this verification is that the separation of preservation and progress allowed for a relatively small and modular proof of a complex property of a *highly coupled system*. Without the separation of preservation and progress, the coupling would have forced us to re-prove many of the intermediate properties.

Example: Adding the Third Dimension We can build a controller enforcing a cube by conjoining Box with another instance of Int substituted to reflect the y dimension. We can then apply this cube to the 3D version of $\mathscr{W}_{QC}$ (including ϕ and the coupling small-angle constraint). Again, the modularity of our proofs shields us from much of the complexity. Box enforced the constraints for $\mathbf{x}$ and $\mathbf{z}$ using $\mathbf{a_x}$ and $\mathbf{a_z}$. Extending this to handle the third dimension simply requires that we use substitution to view $\mathbf{a_x}$ and $\mathbf{a_z}$ as a single unit and carry out the same reasoning, independently controlling that composed unit and $\mathbf{a_y}$.

Table 3.1. Systems implemented and proved correct. The first column is the name of the system. The second column shows how each system was built and proved correct from smaller components. $\{\}i$ indicates a substitution applied to system i (we have omitted the specific substitution); $\otimes$ represents conjunction composition; and $\oplus$ represents disjunction composition. The third and fourth columns show the number of symbols in the discrete controller and the number of lines of proofs, respectively.

id: Name	Built How?	# of Symbols	Lines of Proof
a: 1D velocity bound	From Scratch	43	130
b: 1D position bound	From Scratch	126	484
c: 1D interval	$\{\}b \otimes \{\}b \otimes \{\}a \otimes \{\}a$	323	194
d: 2D square	$\{\}(\{\}c \otimes \{\}c)$	805	258
e: 3D cube	$\{\}(d \otimes \{\}c)$	1353	201
f: 3D square donut	$\{\}e \oplus \{\}e \oplus \{\}e \oplus \{\}e$	5412	23
g: 3D +, T, $\perp$	$\{\}e \oplus \{\}e \oplus \{\}e \oplus \{\}e$	5412	23
h: 3D pilot box	$\{\}e \oplus \{\}e \oplus \{\}e \oplus \{\}e$	5412	23

3.4 Evaluation

We evaluate our approach with four criteria: (1) proof effort, (2) expressiveness, (3) flight behavior, and (4) comparison to fully automated tools.

3.4.1 Proof Effort

Table 3.1 lists all of the geofencing controllers that we verified and flew, along with the composition mechanism, proof size, and number of symbols in the discrete transition for each one. The first 2 rows list controllers that we ported from prior work; these are our atomic building blocks. The next 3 rows list controllers that we built and verified using a combination of substitution and conjunction. The remaining rows list controllers that we built using disjunction and substitution.

Note that our 1D position controller (b in Table 3.1) involves 3 variables and 126 symbols and is a relatively simple discrete transition requiring only multiplication and addition. However, verifying this position controller required substantial, tedious, manual proof effort because of the difficulty of reasoning about nonlinear real arithmetic. On the other hand, our 3D cube controller (*e* in Table 3.1) contains 9 variables, 1353 symbols, trigonometric functions, and angular constraints. This increase in complexity is actually quite daunting for two reasons: (1) arithmetic decision procedures have very bad worst case complexity, and (2) trigonometric functions make the problem undecidable [34]. Despite the substantial increase in complexity, our modular verification approach allowed us to verify the 3D version with only a modest amount of additional manual proof effort.

Moreover, note that all controllers built using disjunction require negligible proof effort on top of the effort required to verify the individual components. This is because disjunction composes proofs of both preservation and progress. As we discuss in the next subsection, this is a very powerful tool.

Finally, it is worth noting that we have several admits in our Coq development: (1) basic arithmetic theorems, (2) some theorems bridging the gap between $\mathscr{W}_{QC}$ and its abstractions, (3) theorems stating progress of the continuous transitions in the quadcopter model. Crucially, these admits do not interfere with the core ideas that we are exploring, namely modular reasoning about sampled-data systems.

3.4.2 Expressiveness

Disjunction is extremely powerful when building real-world controllers. For example, we can build up pixelated versions of arbitrary shapes by composing instances of our cube controller. Interesting properties that we can build with this include: (1) avoid no-fly zones such as airports, the white house, etc, (2) do not fly within a box surrounding the pilot, (3) stay away from static barriers such as trees and buildings, and (4) avoid skyscrapers in a city. In fact, our disjunction rules can be used to compose *any* controller with any other controller, not just a cube. For example, if we were able to verify a controller enforcing a triangular prism or a sphere, we could compose instantiations of these controllers with our existing cube, with each other, or with any other controller.

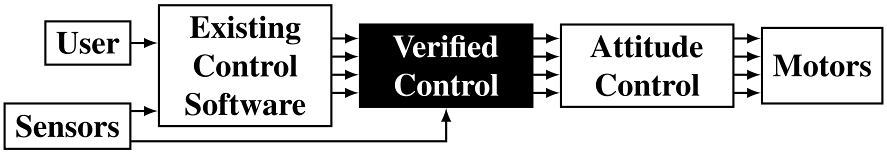


Figure 3.6. System architecture.

Again, all of this requires negligible additional proof burden.

Conjunctive composition is similarly expressive – the box and the cube both make heavy use of conjunctive composition. The proof burden when using conjunction comes from justifying progress (progress) of the resulting systems, which is not as compositional, especially in highly coupled domains. Nevertheless, Our separation of progress (progress) and preservation makes it possible to build compositional proofs of preservation even in highly coupled domains such as the box and cube. Finally, substitution allows us to reuse all controllers by translating and rotating them, and by transforming them into controllers for different physical dynamics.

3.4.3 Behavior in Actual Flight

To make sure that our controllers work well in practice, we manually translated our models to C and ran them on a 3DR Iris+ quadcopter. This allows us to evaluate the accuracy of our approximations. Figure 3.6 depicts the architecture of the system with one of our controllers inserted. Our controllers check the outputs of the pre-existing control software, potentially replacing them with default safe values; this resolves the nondeterminism of the controller specifications. This architecture means that the behavior of much of the existing control software (left of "Verified Control") cannot cause a violation of the fence. Moreover, if the quadcopter remains sufficiently far from the fence boundaries, the system behaves exactly as it would without our controllers, so any performance properties of the existing control software are preserved.

We flew our quadcopter with all controllers listed in Table 3.1. All of the controllers enforced their respective bounds with only minor violations, which can be attributed to un-modeled forces such as wind and to modeling approximations.

Perhaps the largest gap between our temporal logic model and the real world comes from our assumption that the controllers can instantaneously set the orientation, or attitude, of the quadcopter (Section 3.3). This is not possible in reality. Instead, an attitude controller controls attitude indirectly by sending voltage signals to the quadcopter's motors. Thus, our controllers output desired attitude values (i.e. roll and pitch) to an attitude controller, and assume that these values are achieved instantaneously. Prior work has suggested that this is a reasonable approximation, under the small angle assumption, since the attitude dynamics are significantly faster than the velocity and position dynamics [30]. Nevertheless, one of the reasons for running our controllers on a real quadcopter is to experimentally evaluate whether the attitude assumption works well in our context.

Manually translating our models to C introduces a potential source of errors. The two main aspects of this translation are: (1) approximating infinite precision real numbers in the logic using floating point numbers, and (2) validating the assumptions about the runtime of code using worst-case execution time analysis on the resulting code. Both of these problems have been explored in a variety of contexts [22, 53, 89], and are orthogonal to modularity.

Our experiences flying the controllers from this chapter suggest several interesting avenues for future study. First, our controllers can cause the quadcopter to oscillate near the boundaries. If the pilot commands maximum acceleration towards a boundary, then our controllers, rather than converging smoothly to zero acceleration at the boundary, ultimately oscillate quickly between the pilots desired acceleration and the controller's breaking acceleration. This is due to a large discontinuity in control input (desired attitude) near the boundaries. We address this shortcoming in Chapter 4.

Second, an early C implementation of our multi-cube controller (formed from a disjunction of cubes), was overly conservative due to the logic it used for responding to violations of both controllers. In particular, it sometimes prevented the quadcopter from leaving a particular cube, even though the desired quadcopter path was safe for another cube. The solution was to break ties in favor of the least invasive choice, which is a heuristic not expressed in the model. This experience suggests the some sort of liveness property is an interesting avenue for future work in this domain.

Finally, our results suggest that minor violations of even formally verified properties are inevitable since no useful model of the physical world can capture everything. Most of the time our controllers work well outside the boundaries, but our experiments do show that occasionally they cause the quadcopter to temporarily get stuck outside the boundaries. This means that it would be valuable to prove stronger versions of our state invariants expressing a form of robustness. Here, we can build on robustness notions from control theory called input-to-state stability [81] and input-to-state dynamical stability [32]. These definitions state that the behavior of a system *degrades gracefully* as disturbances grow and that the effect of individual disturbances decreases as time passes. For example, a quadcopter running our controllers may violate the boundary by a small amount if there is a small amount of wind and by a large amount if there is a strong wind, but in the absence of wind the vehicle must re-enter the boundary.

3.4.4 Comparison with fully automated tools

There are a number of state-of-the-art tools that attempt to automatically verify hybrid systems [26, 17, 41, 36], but due to the complexity of the domain, they are limited to certain classes of systems and properties.

PHAVer [26], which we ran through the SpaceEx tool platform [25], is able to verify only one of our systems, namely the combined upper and lower bound on velocity.

However, PHAVer is not able to verify the upper bound on velocity in isolation, probably because there are fewer constraints on acceleration to limit the search space, compared to the controller that bounds velocity from above and below. Finally, PHAVer is not able to run any of our other systems because the discrete transitions involve non-linear arithmetic; analyzing these systems would require manual construction of a linear overapproximation of the discrete transitions.

dReach [41] and Flow* [17] are *bounded* model checkers, which means that they can conclude safety of a system within a user-specified time bound. Our Coq proofs, on the other hand, guarantee safety for infinite runs. It is possible to use dReach and Flow* to guarantee safety for all runs by manually providing the inductive invariant. In this way, our results are complementary, as they provide a decomposition technique to manage scalability issues of these tools. However, these tools are currently unable to handle universally quantified variables with unbounded domains. This prevents them from verifying safety of systems with symbolic parameters, such as $\mathbf{a}_{\mathsf{min}}$ – they require concrete numerical bounds on these parameters, which weakens the safety theorem.

3.5 Acknowledgments

This chapter, in full, is adapted from material as it appears in International Conference on Embedded Software 2016. Ricketts, Daniel; Malecha, Gregory; Lerner, Sorin, ACM Press, 2016. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Barrier Certificates

In this chapter, we return to a controller that prevents a vehicle from violating a boundary in one spatial dimension. Controllers with this objective were described in Chapters 2 and 3. However, as described in Section 3.4.3 of Chapter 3, vehicles running those controllers exhibit significant oscillation near the boundary due to a discontinuity in the value output by the controllers. This oscillation, known as chattering in control theory [86], occurs in practice in the presence of discontinuous control because of a combination of un-modeled dynamics and sampled-data implementation. The oscillation is undesirable from the perspective of the pilot. More significantly, it results in a discrepancy at the boundary between the position dynamics in the model and the position dynamics in reality, partially causing the boundary violations that we experienced in actual flight tests.

Our goal was to build a controller whose behavior at the boundaries was acceptable for the popular open source UAV autopilot called Ardupilot [85]. We worked with Ardupilot developers to build such a controller as a component of their new geofence module, which prevents vehicle from exiting a specified safe zone, regardless of what the pilot does. Crucially the controller that we built did not have a discontinuity at the boundary, and flight tests confirmed that this eliminated violations. After building the controller, we attempted to verify safety of a model of the controller in one spatial dimension. However, while attempting this task, we found several important gaps in existing work on formal verification of cyber-physical systems. In this chapter, we present several proof rules and techniques that address these shortcomings, and apply them to verify a double integrator model of the Ardupilot controller in one spatial dimension.

First, deductive techniques for hybrid systems typically involve some continuous analogue of induction, such as differential induction [63], whose mechanical verification we discussed in Chapter 2, or barrier certificates [69]. These techniques provide conditions for verifying that a state predicate is an invariant of a system of differential equations, without actually solving the system of equations. However, differential induction and the original formulation of barrier certificates from [69] are too weak to verify invariants for certain systems, particularly those whose solutions exponentially decay towards the invariant boundary. For example, these techniques cannot verify invariance of $y \leq 0$ along solutions of the system $\dot{y} = -y$. Systems of this form arise naturally when building a controller like the one designed for Ardupilot – the controller should allow the vehicle to approach the boundary and smoothly come to a stop at the boundary. In order to verify this, differential induction must be augmented with another proof rule called differential auxiliaries [64].

On the other hand, recent work from the control theory community [40, 94, 60] has produced a new version of barrier certificates that is less conservative than prior work. In particular, the condition required for invariance captures exponential decay towards the invariant boundary. We provide the first implementation of this approach in a formal verification context, and demonstrate its ease of use on the controller that we designed for Ardupilot.

Second, control systems are often designed under the assumption that controllers run continuously, while the actual implementation is a sampled-data system. In control theory, this process is known as emulation [43]. System designers can compensate for this (and other) approximations by adding a safety “buffer” to the system. For example, the Ardupilot controller we helped build stops the vehicle 1 meter prior to the actual safety boundary. In order to reason about whether such a buffer is sufficient for a given system, we augmented the new barrier certificate proof rule from [40, 94, 60] to bound the error introduced by a continuous time approximation of a sampled-data system. This rule allows one to perform the majority of reasoning in a purely continuous model using powerful techniques resulting from over a century of control theory research. We use this rule to show that the constant sized buffer is, in fact, sufficient to compensate for the continuous time approximation of the Ardupilot controller.

Finally, other formal verification frameworks are unable to apply differential induction and barrier certificates to invariants involving piecewise functions. We show how to remove this restriction by working in the expressive Coq proof assistant. This allows us to verify the critical piecewise invariant arising in the Ardupilot controller.

4.1 Overview

Ardupilot is a popular open source autopilot installed in over 1,000,000 vehicles, including quadcopters and other UAVs [85]. The geofence module allows one to specify a 2D polygon safe region. The pilot is free to move arbitrarily within the polygon, but the module restricts movement near the boundary and ultimately stops the vehicle within 1 meter of the boundary.

We focus on the controller logic for stopping the vehicle 1 meter before the boundary in one spatial dimension. This logic must satisfy three criteria: (1) allow unrestricted movement when the vehicle is far from the boundary, (2) take into account limitations on maximum possible braking force of the system, and (3) bring the vehicle smoothly to a stop at the 1 meter buffer.

The module does so by restricting the velocity as a function of the vehicle’s distance from the boundary. Given the maximum braking force of the systems, a natural approach is to limit the velocity as a function of the *square root* of the distance from the boundary. However, such a limitation results in a discontinuity in control at the boundary, as experienced by the controllers in Chapters 2 and 3. In control theory terminology, such a system behaves as a bang-bang controller, which allows maximum positive acceleration until the last possible moment, when maximum braking force is applied. Such controllers suffer from oscillations at the boundary, violating the third criteria (smoothness).

A solution developed by the Ardupilot engineers is to apply a piecewise limitation to velocity – far from the boundary enforce a square root relationship between position and velocity, while close to the boundary, enforce a linear relationship. Such a linear relationship removes the control discontinuity at the boundary. Section 4.4 provides the formal details of the velocity restriction.

The actual system passes the restricted velocity to an underlying velocity controller. In this chapter, we ignore the dynamics of the inner loop velocity controller controller and as in the rest of this dissertation, treat the system as a double integrator, in which the control law outputs acceleration. Such an approximation is an important first step towards developing new formal verification techniques, as it still forces us to handle the non-trivial velocity restriction law. We leave composition with the velocity controller as an interesting direction for future work.

Given the double integrator model and the velocity restriction law, we show how to derive the controller's logic to guarantee the velocity restriction and thus enforce the boundary on position. This logic comes in the form of state-dependent upper bounds on the control signal (acceleration). The actual control law implementation takes the minimum of these constraints and the pilot's desired acceleration. Crucially, we derive these constraints using our barrier certificate theorem for sampled-data systems, which allows us to derive the constraints assuming a continuous time controller, and transfer these results to the sampled-data model by proving two simple side-conditions on the intersample system behavior.

Section 4.2 presents the formal logic we use to reason about sampled-data systems and gives a double integrator model of the geofence within this logic. Section 4.3 provides formal details on barrier certificates for sampled-data systems and Section 4.4 shows how to apply them to reason about the geofence. Section 4.5 describes the Coq implementation of our results with Section 4.5.1 describing how we extend formal verification using barrier certificate to piecewise functions.

4.2 Logic

This chapter uses a different formal framework than the previous chapters. Rather than linear temporal logic, we use a logic over system trajectories, i.e. functions $\mathbb{R}_{\geq 0} \to \mathbb{R}^n$ from time to state.

Our logic is defined in terms of two relations: models (written $F \models P$) expresses that a trajectory predicate P holds on a trajectory F; and entails (written $P \vdash Q$) expresses that a trajectory predicate Q holds on all trajectories that P holds on.

Formally, entailment is defined as follows:

Definition 4.1 (Entailment)**.** *For trajectory predicates P and Q,*

$$P \vdash Q \equiv \forall F \in \mathbb{R}_{\geq 0} \to \mathbb{R}^n : F \models P \implies F \models Q$$

We now define the models relation for several types of predicates. First, we define models for a predicate that expresses whether a trajectory is a valid solution to a sampled-data system whose intersample dynamics are given by $\dot{\mathbf{z}} = f(\mathbf{z}, u(\mathbf{z_k}))$ where $\mathbf{z_k}$

is the state at the last sample and $u \in \mathbb{R}^n \to \mathbb{R}^m$ is the control law. This definition requires a notion of well-formed sample times:

Definition 4.2 (Sample times)**.** *A sampling sequence t_k with $k \in \mathbb{N}$ is well-formed with bound Δ if*

$$\begin{aligned} &t_0 = 0 && \wedge \\ &\forall k \in \mathbb{N} : 0 < t_{k+1} - t_k \leq \Delta && \wedge \\ &\forall t \in \mathbb{R}_{>0} : \exists k \in \mathbb{N} : t_k \leq t < t_{k+1} \end{aligned}$$

Using the definition of well-formed sample times, we can formally specify the models relation for a sampled-data system whose sample times (time between discrete controller executions is bounded by Δ:

Definition 4.3 (Sampled data solutions)**.**

$$\begin{aligned} F \models \{\dot{\mathbf{z}} = f(\mathbf{z}, u)\}_\Delta \equiv \exists t_k \in \{t_k : t_k \textit{ well-formed}\} : \\ \forall k \in \mathbb{N} : \forall t \in [t_k, t_{k+1}) : \dot{F}(t) = f(F(t), u(F(t_k))) \end{aligned}$$

In Definition 4.3, we enforce more structure on the system than our Sys abstraction from previous chapters (Definition 1.2). In particular, we make explicit the control law and enforce that the control input is held constant between samples. Adding this structure allows us to state a powerful barrier function theorem for sampled-data systems.

Definition 4.3 does not put any restrictions on the initial state of the trajectory, $F(0)$. For this, we use state predicates – predicates over a single state. A state predicate holds on a trajectory if it holds on the initial state of that trajectory:

Definition 4.4 (Initially)**.** *For a state predicate P and a trajectory F,*

$$F \models P \equiv F(0) \models P$$

Next, we need to express that a state predicate is an invariant of a trajectory. For this, we define an always operator, which takes a state predicate and expresses that it holds for all time along a trajectory.

Definition 4.5 (Always)**.** *For a state predicate P and a trajectory F,*

$$F \models \Box P \equiv \forall t \in \mathbb{R}_{\geq 0} : F(t) \models P$$

Finally, sampled-data systems often have an upper bound on the time between samples. Thus, we define a bounded always operator in order to reason about invariants between samples. This operator takes a state predicate and expresses that it holds until a time bound Δ along a trajectory.

Definition 4.6 (Bounded Always)**.** *For a state predicate P, a constant* $\Delta \in \mathbb{R}_{\geq 0}$*, and a trajectory F,*

$$F \models \Box_{\Delta} P \equiv \forall 0 \leq t \leq \Delta : F(t) \models P$$

4.3 Exponential barrier certificates

Deductive approaches to hybrid system verification typically involve a continuous analogue of induction. Barrier certificates are one such technique. In the typical formulation of barrier certificates, one proves the invariance of $B(\mathbf{z}) \leq 0$ by proving that the value of $B(\mathbf{z})$ does not increase along trajectories of the system. Recent work from control theory [40] has relaxed this requirement – the rate of change of $B(\mathbf{z})$ along trajectories must be proportional to its value. This allows the value of $B(\mathbf{z})$ to increase along trajectories, but requires that the rate of change slow as trajectories approach the boundary $B(\mathbf{z}) = 0$. Essentially, $B(\mathbf{z})$ can exponentially decay towards the boundary but never cross it.

In this section, we present two barrier certificate proof rules for sampled-data systems. The first (Lemma 4.1) is a trivial adaptation of the rule from [40] to sampled-data systems. We explain why this rule, by itself, is insufficient. Then we present our new version of exponential barrier certificates for sampled-data systems that decomposes verification into an exponential decay property of the continuous time system approximation and two simple properties about the intersample behavior (Theorem 4.1). In Section 4.4, we show how to use these proof rules to reason about the Ardupilot controller. All results in this section have been formalized in the Coq proof assistant. To the best of our knowledge, this is the first such formalization of exponential barrier certificates.

Barrier certificates are functions $B \in \mathbb{R}^n \to \mathbb{R}$ that assign a scalar to every state. One establishes the invariance of $B(\mathbf{z}) \leq 0$ by proving a property about the rate of change of $B(\mathbf{z})$ along trajectories of the system. Formally, $\nabla B \bullet f(\mathbf{z}, \mathbf{z_k})$ gives the rate of change or time derivative of B along trajectories of the system, also known as the Lie derivative of B. Here, ∇B denotes the gradient of B, or vector of partial derivatives $[\frac{\partial B}{\partial x_1}, \ldots, \frac{\partial B}{\partial x_n}]$.

Lemma 4.1 gives our trivial adaptation of barrier certificates from [40] to sampled-data systems.

Lemma 4.1. *Consider a continuously differentiable function $B \in \mathbb{R}^n \to \mathbb{R}$, constants*

$\lambda \in \mathbb{R}$ and $\Delta \in \mathbb{R}_{>0}$, and a state predicate P. If the following condition holds:

$$\forall \mathbf{z}, \mathbf{z_k} \in \{\mathbf{y} \in \mathbb{R}^n \mid \mathbf{y} \models P\} : \nabla B \bullet f(\mathbf{z}, \mathbf{z_k}) \leq \lambda B(\mathbf{z})$$

then

$$\begin{aligned} & B(\mathbf{z}) \leq 0 \ \wedge \ \Box P \ \wedge \ \{\dot{\mathbf{z}} = f(\mathbf{z}, u(\mathbf{z_k}))\}_\Delta \\ \vdash \ \ & \Box B(\mathbf{z}) \leq 0 \end{aligned}$$

Lemma 4.1 states that if $B(\mathbf{z}) \leq 0$ holds initially, and if, assuming a known invariant P, the time derivative of B is at most proportional to its value, then $B(\mathbf{z}) \leq 0$ is an invariant.

While this theorem is powerful for continuous time systems, it is lacking for sampled-data systems. The problem is that the premise of the theorem does not constrain the relationship between the sampled state $\mathbf{z_k}$ and the current state $\mathbf{z}$. However, as we will see, this theorem still has utility. In particular, once an invariant P has been established using Theorem 4.1, Lemma 4.1 can be used to establish a new invariant using the invariance of P.

We would like the premise of Theorem 4.1 to be relaxed so that the control input u is applied continuously. In other words, we would like to only prove $\nabla B \bullet f(\mathbf{z}, \mathbf{z}) \leq \lambda B(\mathbf{z})$ rather than $\nabla B \bullet f(\mathbf{z}, \mathbf{z_k}) \leq \lambda B(\mathbf{z})$. We can achieve this relaxation if we can establish an additional condition on the intersample behavior: the time derivative of B does not change by more than a constant C between samples. Under such a condition, the property $B(\mathbf{z}) \leq 0$ is not an invariant, but $B(\mathbf{z}) \leq C \cdot \Delta$ is. This mirrors a control design practice – treat the controller as continuous and add a constant safety buffer to account for the approximation error.

This argument is formalized in Theorem 4.1. Condition (i) is the relaxed version of the premise of Theorem 4.1. Condition (ii) characterizes the behavior of the state between samples, while condition (iii) bounds the change in time derivative of B, under the characterization established by (ii). Such a decomposition allows us to use any continuous reasoning technique to dispatch (ii) and arithmetic reasoning to establish (iii).

Theorem 4.1. *Consider a continuously differentiable function $B \in \mathbb{R}^n \to \mathbb{R}$, constants $C, \Delta \in \mathbb{R}_{>0}$, a state predicate P, and a state relation S. If the following conditions hold:*

i) $\forall \mathbf{z} \in \{\mathbf{y} \in \mathbb{R}^n \mid \mathbf{y} \models P\} : \nabla B \bullet f(\mathbf{z}, \mathbf{z}) \leq -\frac{B(\mathbf{z})}{\Delta}$

ii) $\forall \mathbf{z_k} \in \mathbb{R}^n : \mathbf{z} = \mathbf{z_k} \wedge \{\dot{\mathbf{z}} = f(\mathbf{z}, \mathbf{z_k})\}_\Delta \vdash \Box_\Delta(\mathbf{z}_k, \mathbf{z}) \in S$

iii) $\forall (\mathbf{z_k}, \mathbf{z}) \in S : \nabla B \bullet f(\mathbf{z}, \mathbf{z_k}) \leq \max\left(\nabla B \bullet f(\mathbf{z_k}, \mathbf{z_k}) + C, 0\right)$

then

$$\begin{aligned} & B(\mathbf{z}) \leq C \cdot \Delta \ \wedge \ \Box P \ \wedge \ \{\dot{\mathbf{z}} = f(\mathbf{z}, u(\mathbf{z_k}))\}_\Delta \\ \vdash \ \ & \Box B(\mathbf{z}) \leq C \cdot \Delta \end{aligned}$$

Proof. For any k and for any $t \in [t_k, t_{k+1})$,

$$\begin{aligned} B(\mathbf{z}(t)) - B(\mathbf{z}(t_k)) &= \int_{t_k}^{t} \nabla B \bullet f(\mathbf{z}(\tau), \mathbf{z}(t_k)) d\tau \\ &\leq \int_{t_k}^{t} \big(\max\left(\nabla B \bullet f(\mathbf{z}(t_k), \mathbf{z}(t_k)) + C, 0\right)\big) d\tau && \text{(by (iii) and (ii))} \\ &\leq \int_{t_k}^{t} \left(\max(-\frac{B(\mathbf{z}(t_k))}{\Delta} + C, 0)\right) d\tau && \text{(by (i))} \\ &= (t - t_k) \cdot \left(\max\left(-\frac{B(\mathbf{z}(t_k))}{\Delta} + C, 0\right)\right) \end{aligned}$$

Therefore, for any k and any $t \in [t_k, t_{k+1})$, since $t - t_k \leq Delta$,

$$B(\mathbf{z}(t_k)) \leq C \cdot \Delta \implies B(\mathbf{z}(t)) \leq C \cdot \Delta \tag{4.1}$$

By induction on k, $\forall t \geq 0 : B(\mathbf{z}(t)) \leq C \cdot \Delta$ □

Note the max in condition (iii) of Theorem 4.1. The intersample derivative of B does not need to be close to the sample time derivative of B as long as it is non-positive. This subtlety bares a resemblance to the normal formulation of barrier certificates in which the time derivative of B must always be non-positive.

4.4 Ardupilot controller

Now that we have established the general theory, we can use it to reason about a double integrator model of the Ardupilot controller, whose intersample behavior is given by:

$$\dot{x} = v, \dot{v} = u(x_k, v_k)$$

The control law u is designed to enforce $x \leq 0$. We begin this section by describing how u is designed to (almost) achieve this objective. As mentioned in Section 4.1, the control law design comes in the form of state-dependent upper bounds on the control signal u. As described in that section, the actual control law implementation takes the minimum of the pilot's desired acceleration and those constraints.

The Ardupilot control law was designed assuming that the law is applied continuously (not as a sampled-data system). This approximation simplifies design and reasoning but also introduces an error, which actually results in a violation of $x \leq 0$. We will ultimately apply Theorem 4.1 to quantify this violation.

Relevant to the controller design are the physical constraints on the system, particularly the maximum possible acceleration. Since we are interested in a controller that stops the system's position from violating some boundary, the important physical constraint is a limit on the system's braking acceleration. That is, we need to ensure that the control law does not require deceleration of a greater magnitude than is physically possible. We denote by μ this magnitude – we need to ensure that the system never reaches a state x, v in which $u(x, v) < -\mu$.

Since the stopping distance for a particle with acceleration a and initial velocity v is $\frac{v^2}{2a}$, the μ constraint implies that the system must enforce the following invariant:

$$x \leq \frac{\max(v,0)^2}{2\mu}$$

However, since the controller was designed under a continuous time approximation, the invariant needs to be strengthened by using a more conservative braking acceleration $\tilde{\mu} < \mu$. At the end of this section, we will give precise conditions on $\tilde{\mu}$ to ensure that the system never reaches a state x, v in which $u(x, v) < -\mu$. The strengthened invariant is thus:

$$x \leq \frac{\max(v,0)^2}{2\tilde{\mu}}$$

This invariant is still insufficient. The problem is that such an invariant will result in a control discontinuity at the position boundary $x = 0$. This occurs if the system is in a state in which $x = \frac{\max(v,0)^2}{2\tilde{\mu}}$. Such a state requires that the controller issue constant acceleration $-\tilde{\mu} < 0$ when $x < 0$ and permits zero acceleration when $x = 0$. As described in the introduction, such a control law produces undesirable oscillations at the boundary and is physically impossible.

To solve this problem, the Ardupilot developers designed a more conservative invariant that is a piecewise function of the distance from the boundary. Close to the boundary, the velocity limit is linear in the distance, while farther away, the velocity limit is proportional to the square root of the distance. This relationship is depicted in Figure 4.1. Notice that the piecewise transition between these two relationships occurs where they are tangent. This ensures that (a) a barrier certificate describing the region is continuously differentiable, and (b) the control constraints induced by such a barrier are

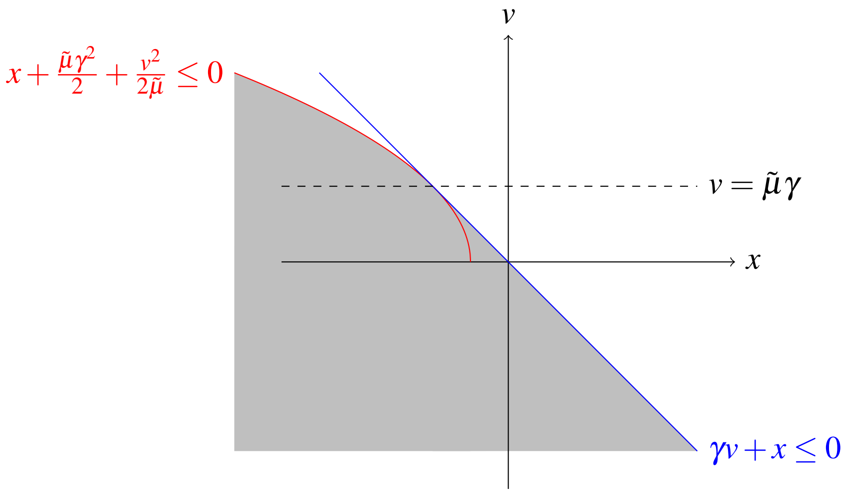

$x+\frac{\tilde{\mu}\gamma^2}{2}+\frac{v^2}{2\tilde{\mu}}\leq 0$
v
$v=\tilde{\mu}\gamma$
x
$\gamma v+x\leq 0$

Figure 4.1. A depiction of $B_1(x,v)\leq 0$ where the red curve represents the first branch of the piecewise function and blue the second.

continuous.

Formally, the region depicted in Figure 4.1 is $B_1(x,v)\leq 0$, where

$$B_1(x,v)=\begin{cases}\gamma v+x & \text{if } v\leq\tilde{\mu}\gamma\\ x+\frac{\tilde{\mu}\gamma^2}{2}+\frac{v^2}{2\tilde{\mu}} & \text{otherwise}\end{cases}\tag{4.2}$$

Our goal is to (almost) enforce the invariance of $x\leq 0$ – that is, we would like to enforce $x\leq K$ for some positive constant K. We now take a two step process to prove the invariance of $x\leq K$ for some constant K. First, we use Theorem 4.1 and constraints on u to prove the invariance of $B_1(x,v)\leq K$. Then we use Lemma 4.1 and the invariance of $B_1(x,v)\leq K$ to prove the invariance of $x\leq K$. The assumptions on u are as follows:

Assumption 4.1. *For any $x,v\in\mathbb{R}$,*

$$u(x,v)\leq\begin{cases}\frac{-(\gamma v+x)}{T\gamma}-\frac{v}{\gamma} & \textit{if } v\leq\tilde{\mu}\gamma\\ \frac{-\tilde{\mu}x}{Tv}-\frac{\tilde{\mu}^2\gamma^2}{2Tv}-\frac{v}{2T}-\tilde{\mu} & \textit{otherwise}\end{cases}$$

Note that the division by v in the second branch of assumption 4.1 is not problematic, since in that branch, $v > \tilde{\mu}\gamma > 0$.

Since the control law was designed using a continuous time approximation, we need to put a constant upper bound on the control in order to bound the intersample behavior and thus apply Theorem 4.1. Such a bound is captured in the following assumption:

Assumption 4.2. $\forall x, v \in \mathbb{R} : u(x,v) \leq \tilde{\mu}$

Theorem 4.1 requires reasoning about the time derivative of B_1, which is given by (4.3). The formal details details of reasoning about Lie derivatives of piecewise functions, like the one in (4.3) are deferred to Section 4.5.1.

$$\forall v, x_k, v_k \in \mathbb{R} : \nabla B_1 \bullet [v, u(x_k, v_k)] = \begin{cases} \gamma u(x_k, v_k) + v & \text{if } v \leq \tilde{\mu}\gamma \\ v + \frac{v \cdot u(x_k, v_k)}{\tilde{\mu}} & \text{otherwise} \end{cases} \tag{4.3}$$

The next three lemmas satisfy the three conditions of Theorem 4.1 for B_1.

Lemma 4.2 (Condition (i))**.** *For any $x, v \in \mathbb{R}$,*

$$\nabla B_1 \bullet [v, u(x,v)] \leq -\frac{B_1(x,v)}{\Delta}$$

Proof. Solving the above in equality for u results in exactly the inequality in assumption 4.1. □

Lemma 4.3 (Condition (ii))**.** *For any $x_k, v_k \in \mathbb{R}$,*

$$\begin{aligned} & x = x_k \wedge v = v_k \wedge \{\dot{x} = v, \dot{v} = u(x_k, v_k)\}_\Delta \\ \vdash \quad & \Box_\Delta v \leq v_k + \max(u(x_k, v_k), 0) \cdot \Delta \end{aligned}$$

Proof. By integration of v from 0 to Δ. □

The intersample dynamics are sufficiently simple that Lemma 4.3 can be proven by integration. However, since condition (ii) of Theorem 4.1 concerns purely continuous time dynamics, the intersample behavior of more complex systems can be verified using a number of powerful techniques from prior work, including differential induction [63]. Making such a connection formal is an interesting direction for future work.

Lemma 4.4 (Condition (iii))**.** *For any $x_k, x, v_k, v \in \mathbb{R}$ such that $v \leq v_k + \max(u(x_k, v_k), 0) \cdot \Delta$,*

$$\nabla B_1 \bullet [v, u(x_k, v_k)] \leq \nabla B_1 \bullet [v_k, u(x_k, v_k)] + 2\tilde{\mu}T$$

Proof. By assumption 4.2 and first order reasoning over real arithmetic. □

It is important to note that the above theorem only relies on the very simple intersample behavior of v and constant bounds on u. It does not rely on bounds on position nor does it depend on the more complex bounds on u used in Lemma 4.2. This serves as evidence that Theorem 4.1 allows one to prove relatively simple side conditions in order to transfer continuous time results to the sampled-data domain.

Given Lemmas 4.2, 4.3, and 4.4, we can now prove the invariance of $B_1(x, v) \leq (2\tilde{\mu}T) \cdot \Delta$.

Lemma 4.5.

$$\begin{aligned} & B_1(x,v) \leq (2\tilde{\mu}T) \cdot \Delta \qquad \land \\ & \{\dot{x}(t) = v(t), \dot{v}(t) = u(x(t_k), v(t_k))\}_\Delta \\ \vdash\ & \Box B_1(x,v) \leq (2\tilde{\mu}T) \cdot \Delta \end{aligned}$$

Proof. By Theorem 4.1. Lemmas 4.2, 4.3, and 4.4 satisfy conditions (i), (ii), and (iii) of Theorem 4.1, respectively. We instantiate the state predicate P of Theorem 4.1 with the trivial predicate True. □

Finally, given the invariance of $B_1(x,v) \leq (2\tilde{\mu}T)\cdot\Delta$ provided by Lemma 4.5, we can return to Theorem 4.2, which establishes the invariance of $x \leq (2\tilde{\mu}T)\cdot\Delta$. We restate the theorem here for convenience:

Theorem 4.2.

$$\begin{array}{ll} & B_1(x,v) \leq (2\tilde{\mu}T)\cdot\Delta \quad\quad \wedge \\ & x \leq (2\tilde{\mu}T)\cdot\Delta \quad\quad\quad\quad \wedge \\ & \{\dot{x} = v, \dot{v} = u(x_k, v_k)\}_\Delta \\ \vdash & \Box x \leq (2\tilde{\mu}T)\cdot\Delta \end{array}$$

Proof. We apply Theorem 4.1 with barrier function

$$B_2(x,v) = x - (2\tilde{\mu}T)\cdot\Delta$$

constant $\frac{-1}{\gamma}$, and state predicate

$$B_1(x,v) \leq (2\tilde{\mu}T)\cdot\Delta$$

Simple arithmetic reasoning reveals that for all $x, v \in \mathbb{R}$,

$$\nabla B_2 \bullet [v,\cdot] + \frac{B_2(x,v)}{\gamma} \leq B_1(x,v) - (2\tilde{\mu}T)\cdot\Delta \tag{4.4}$$

$$\leq 0 \tag{4.5}$$

which satisfies the premise of Theorem 4.1. □

Theorem 4.2 verifies the design procedure followed by the Ardupilot engineers – the controller can be designed assuming that it is applied continuously and a small constant buffer can compensate for the approximation error. Moreover, it shows that the constraints on u fall out naturally from the velocity bounds depicted in Figure 4.1 by a simple application of Theorem 4.1.

Finally, we provide conditions on $\tilde{\mu}$ to ensure that the system never reaches a state x, v in which $u(x,v) < -\mu$. In other words, for any x and v satisfying $B_1(x,v) \leq (2\tilde{\mu}T) \cdot \Delta$ and $x \leq (2\tilde{\mu}T) \cdot \Delta$, we need to ensure that there exists a value satisfying the upper bound from assumption 4.1 and the lower bound $-\mu \leq u(x,v)$. Arithmetic reasoning reveals that the constraints are:

$$\tilde{\mu} \leq \frac{\mu}{1 + \frac{2\Delta}{\gamma}} \tag{4.6}$$

For Ardupilot, the sample time is small relative to γ, so $\tilde{\mu}$ is close to μ. This again verifies the design procedure followed by the Ardupilot engineers – to account for error, the controller is designed with a conservative braking acceleration $\tilde{\mu}$ rather than the actual physical limit μ.

4.5 Coq implementation

We have implemented all of the results of this chapter in the Coq proof assistant. The development is available from: https://github.com/dricketts/barrier-sampled-data. The core theory required 598 lines of definitions and 620 lines of proof, while the geofence model required 99 lines of definitions and 205 lines of proof. On the other hand, the Second-Derivative Controller from Chapter 3, verified without the barrier function theorems of this chapter, required 484 lines of proof, despite simpler constraints on the control law. This provides evidence that Theorem 4.1 and Lemma 4.1 have the potential to considerably reduce the formal proof burden for safety of sampled-data systems.

4.5.1 Differentiation

In Section 4.4, we glossed over the formal details of verifying the gradient of a barrier function. Working in an expressive proof assistant like Coq allows us to verify this step as well, even for piecewise functions. That is, we can formally prove the validity of equations like (4.3) using Lemma 4.6:

Lemma 4.6. *For functions $e_1, e_2, e_3, e_4 \in \mathbb{R}^n \to \mathbb{R}$, variable x, vector $f \in \mathbb{R}^n$, and constant $c \in \mathbb{R}$, if the following conditions hold,*

i) $x = c \implies e_1 = e_2$

ii) $x = c \implies e_3 = e_4$

iii) $x \leq c \implies \nabla e_1 \bullet f = e_3$

iv) $x \geq c \implies \nabla e_2 \bullet f = e_4$

then

$$\nabla \left(\begin{cases} e_1 & \text{if } x \leq c \\ e_2 & \text{otherwise} \end{cases} \right) \bullet f = \begin{cases} e_3 & \text{if } x \leq c \\ e_4 & \text{otherwise} \end{cases}$$

To the best of our knowledge, this is the first formal verification framework for hybrid systems that can handle piecewise functions.

4.5.2 Logics in Coq

We implemented our trajectory logic using Charge! [39], a framework for easily defining and reasoning about logics within Coq. Charge! allows a us to declare that a particular type forms a logic by proving that a small set of axioms hold for that type. In our case, these axioms are proved automatically. In return, we automatically get standard logical operators such as conjunction, disjunction, and implication lifted into the logic. For example, in Section 4.3, we frequently wrote formulas like $P \wedge Q$ where P and Q are trajectory predicates. Without Charge!, we would have to define what it means to conjoin two trajectory predicates. With Charge!, we get this for free.

Along with the normal logic definitions, Charge! also automatically provides standard logical proof rules like commutativity of conjunction and tactics to discharge simple proof obligations. In short, Charge! provides much of the boiler-plate logical reasoning and definitions for free.

Most significantly, Charge! allows us to easily use the most appropriate logic for each proof obligation, all within the same formal foundation of Coq. For example, condition (ii) of Theorem 4.1 is currently phrased as a trajectory predicate entailment. However, Platzer's differential dynamic logic (dL) [65] may be a better logic for reasoning about this condition. Charge! could allow us to easily build dL as a logic in Coq and re-phrase this condition in terms of dL.

4.6 Acknowledgments

A special thanks to the Ardupilot developers, particularly Leonard Hall, who designed the piecewise velocity restriction that we modeled in this chapter.

This chapter, in part, is currently being prepared for submission for publication of the material. Ricketts, Daniel; Ghaffari, Azad; Imoleayo, Abel; Lerner, Sorin; Krstic, Miroslav. The dissertation author was the primary investigator and author of this material.

Chapter 5

Related Work

There has been a tremendous amount of work on the specification and verification of hybrid systems, both from the verification community and the control theory community. In this section, we describe some of the prior work in this area, highlighting the commonalities and difference with our own work.

5.1 Hybrid Automata

Hybrid automata [35, 52] extend traditional automata with continuous transitions. The primary reasoning method for hybrid automata is model checking [26, 36]. There have been a number of highly successful implementations such as HyTech [36], PHAVer [26], Flow* [17], and dReach [41]. While these are powerful tools, reachability is only decidable for a very restricted class of automata, for example rectangular automata [37]. In practice hybrid system model checkers often struggle with large, complex systems [59]. As discussed in Chapter 3, hybrid system model checkers cope with the complexity by either verifying weaker properties (e.g. bounded safety, concrete bounds on quantifiers) or by limiting the structure and arithmetic operators within both discrete and continuous transitions. As noted in that chapter, the modularity rules presented in that chapter can complement automated tools by providing a mechanism to decompose a system into components that are in range for full automation. Making this connection formal is an interesting direction for future work.

Alur *et al.* [4] present rules for conjoining hybrid automata with syntactically independent interface variables and for renaming variables. Our substitution rules generalize substitution in [4] to allow substituting *expressions* for variables, requiring us to separately justify progress through invertibility of substitution. Moreover, different modules often need to output to the same actuators, and our work pushes beyond the restriction of syntactically independent variables. Finally, unlike [4], we present rules for disjunctive composition.

There has also been work on hybrid automata, model checking, and other automation techniques specialized to sampled-data systems. We defer discussion of this work to Section 5.5.

5.2 Deductive logics

Hybrid systems have been formalized in proof assistants, including the HHL prover [48, 88] and using Platzer's differential dynamic logic (dL) in KeYmaera X [67]. This section focuses heavily on dL as it is the most developed logic for hybrid systems (5.2.1). We also provide a discussion of HHL and other work formalizing hybrid systems in general purpose proof assistants (5.2.2).

5.2.1 Differential Dynamic Logic

The dL logic provides a complete proof calculus [65] for hybrid systems with compositional proof rules similar to Hoare logic [38]. It has been implemented in the KeYmaera X proof assistant [67] and applied to a large number of case studies, including [51, 66, 42, 9].

The work in Chapter 2 complements the dL logic with an induction rule specific to sampled-data systems (rather than arbitrary hybrid systems) as well as an acyclic parallel composition proof rule, applied to compose sensor error/delay compensation modules with safety controllers. In addition, that work gives the first formal verification of a continuous induction rule (i.e. differential induction [62, 63]). While the work of this dissertation was done in a different logic (LTL for Chapters 2 and 3 and a trajectory logic for Chapter 4), we believe that all rules can be adapted to dL. Doing so is an interested direction for future work, as it will allow a user to reap the benefits of the generality of dL with the domain specific proof rules of this dissertation, all in a fully formal context.

The work in Chapter 3 further complements dL's general proof calculus with rules for building sampled-data systems modularly. dL is not as expressive as Coq's logic, and some of the proof rules from Chapter 3 are not expressible in dL, namely those with side conditions (*e.g.* invertible substitutions, disjointness of primed variables). The only way to add these rules to KeYmaera X would be to extend its core with soundness-critical checking of side conditions or to employ potentially slow and brittle tactics to prove soundness of composition on a case-by-case basis. By working in Coq, we are able to get higher-order modular *theorems* without compromising soundness.

Since publication of the work in Chapters 2 and 3, there has been work on embedding dL in the Coq and Isabelle proof assistants [13]. This is a first step towards extending dL with the types of domain-specific proof rules found in this dissertation, without compromising soundness. The Coq and Isabelle embeddings in [13] deep embeddings, both of the logic's syntax and of the proof theory. This was necessary in order to extract a verified version of the KeYmaera X kernel. On the other hand, the work in Chapters 2 and 3 used a deep embedding of the syntax but not the proof theory, while the work in Chapter 4 used a fully shallow embedding of the logic. Thus, all of the work in this dissertation uses Coq's kernel as the proof checking core rather than having a separate verified proof checking kernel. Having a range of embeddings within the same domain could ultimately permit an evaluation of which embedding is optimal for extending the proof calculus with higher-order domain-specific proof rules whose side conditions are not expressible within the object logic (e.g. dL).

After publication of the work in Chapters 2, there was work in dL on composition [57, 58] of hybrid systems. The authors provide a form of parallel composition of components, allowing components to interact via ports satisfying input and output contracts. Unlike the work in Chapter 3, their framework explicitly disallows components from outputting to the same variables. This makes the paper most comparable to the composition rule (Theorem 2.2) from Chapter 2. The framework from [57, 58] does remove the restriction to acyclic composition from Theorem 2.2. However, parallel composition in that framework is not an associative operation, while it is in Chapter 2. This restricts the flexibility of a user in building and composing more than two modules, as we did in Section 2.3.3. Moreover, the work from [57, 58] was not done in a higher-order proof assistant. Thus, the theorem is applied in KeYmaera X using an untrusted proof tactic. While this does not compromise soundness, it results in potentially slow and brittle proofs relative to application of a theorem in a proof assistant like Coq.

Finally, the main application result from Chapter 4 (Theorem 4.2), while provable in dL, would require a combination of proof rules including differential induction and differential auxiliaries to emulate exponential barrier certificates and a number of discrete rules to handle the discrete transitions introduced by the sampled-data model. Most significantly, there is no proof rule that allows the control to be treated as continuous while separately bounding the error introduced by this approximation. Thus, we complement the general dL calculus with powerful higher order proof rules that abstract common and natural reasoning patterns for sampled-data systems. Again, working in Coq allows us to add these rules as theorems so that we do not compromise soundness.

5.2.2 Other Logics

The HHL prover [48, 88] is an Isabelle/HOL embedding of Hybrid Hoare logic. Like KeYmaera X, the HHL prover provides a general proof calculus for hybrid systems but does not contain domain-specific proof rules like the ones in this dissertation for sampled-data systems. Moreover, the embedding does not contain any rules for verifying invariants of continuous flows. Instead, the prover relies on external decision procedures that automatically search for invariants. The prover assumes that invariants returned by these decision procedures are valid. Thus, formalizing theorems for checking invariants of continuous flows, as we have done throughout this dissertation, complements the decision procedures by preventing a bug in the decision procedures from compromising soundness. Instead, theorems like Theorem 4.1 from Chapter 4 check the output of decision procedures that produce invariants.

Other work formalizing hybrid systems in general-purpose proof assistants includes [87, 3, 61, 28, 18, 6, 8, 13]. With the exception of [13], none of this work includes formalization of *continuous* induction rules like barrier certificates and differential induction or modular proof rules like the ones from Chapters 2 and 3. In addition, the work in Chapter 4 is the first to formalize the powerful exponential condition form of barrier certificates and the first to adapt it to sampled-data systems.

5.2.3 Temporal Logic

The foundation for Chapters 2 and 3 is temporal logic, and there has been a lot of work on composition in temporal logic, most notably by Abadi and Lamport [2, 1]. Their work describes how to reason about the conjunction of LTL specifications, but they do not deal with the interplay between conjunction and progress or substitution and progress. It is important to note that the preservation/progress split is not the same as the safety/liveness split as both preservation and progress are safety properties. Abadi and

Lamport address Zeno specifications in [1], but do not address the relationship between Zenoness and conjunction. Finally, neither of these works addresses substitution in the presence of progress. It is also important to note that we use disjunction for non-deterministic choice between controllers while much of the work in temporal logic uses disjunction to represent interleaving specifications of asynchronous concurrent systems. Other work includes techniques for decomposing LTL verification into a search for suitable barrier certificates [91] and deductive rules for synthesizing controllers satisfying ATL* properties [24]. While these approaches apply to more temporal logic formulas than ours, they do not address verification using conjunction, disjunction, and substitution. There has also been work on synthesis using approximate bisimulations [82]. We focus on the complementary task of composing and reusing controllers.

5.3 Architectures for Cyber-physical Systems

In Chapters 2 and 3, we describe how to architect hybrid systems in order to ensure safety in the presence of complex, unverified controllers. There has been prior work in this area, much of it based on the simplex architecture [75]. In this architecture, there is a simple module that constantly monitors the system and takes control away from more complex modules before the system can enter an unsafe state. We follow a similar principle with our controller architecture from those two chapters. Our work complements that based on [75] by formally verifying the simple monitoring module (our controllers) rather than relying on their simplicity for correctness.

Livadas and Lynch solve a similar problem using hybrid I/O automata to model and reason about "protectors" for hybrid systems [50]. A protector is designed to ensure a safety property of a particular hybrid system. Livadas and Lynch present a series of rules for conjunctive composition of protectors. In contrast, our work also supports other forms of composition, *e.g.* disjunctive composition, and we show how these operators can be *combined* to achieve modular verification.

Our system architecture from Chapter 2 is closely related to the ModelPlex framework in [56]. This framework allows a user to produce provably correct runtime monitors for hybrid systems. These monitors check the execution of a hybrid system implementation for compliance to a model. If the implementation deviates from the model, then any safety properties with respect to that model are no longer guaranteed. In this case, the monitor provides a mechanism for switching to a fail-safe controller. ModelPlex requires that the safety of these fail-safe controllers be verified using other techniques. Thus, the work in this dissertation complements ModelPlex by providing domain-specific proof rules for verifying fail-safe controllers for sampled-data systems.

Our theory from Chapter 3 is closely related to the work on modular construction of nonblocking supervisory controllers in discrete-event systems [92]. However, our models explicitly include differential equations rather than requiring a discrete abstraction and do not require a notion of termination in a marked state. Moreover, to the best of our knowledge, none of this work makes the inductive invariant explicit and separates preservation from progress.

Finally, the area of switching control [47] from control theory focuses on (in our terminology) disjunctive composition of controllers. Our inclusion of invariants in the discrete controller corresponds to expressing the switching boundary in a switching controller. However, the focus of switching control theory is optimality, stability, and transitionability, whereas we focus on complementary properties like bounding the state space.

5.3.1 Geofences

There has been research on keeping UAVs in a designated safe area, and more generally on obstacle avoidance strategies for UAVs. The most related work is the recent geofencing strategy developed by Gurriet *et al.* [33]. This work places constraints on a pilot's desired velocity vector to avoid leaving a specified safe region. The strategy is complementary to the one described in this paper as it leaves the maximum velocity in a given direction unspecified. Future work can explore composing their strategy with the concrete velocity constraints described in Chapter 4.

5.4 Inductive Methods for Continuous and Hybrid Systems

There is a long history of techniques for establishing forward invariance of systems of differential equations.[1] One of the earliest such techniques is known as Nagumo's viability theorem [10]. This technique relies on a notion called the tangent cone of a set at a point. Informally, the tangent cone of a set S at a point x is the set of all vectors starting from x that point inward towards S. Nagumo's viability theorem states that a closed set S is forward invariance with respect to a system of differential equations $\dot{\mathbf{x}} = f(\mathbf{x})$, if at all points $\mathbf{x}$ on the boundary of S, $f(\mathbf{x})$ is contained in the tangent cone of $\mathbf{x}$ at S. Such a semantic condition for invariance is powerful, as it applies to arbitrary closed sets. However, it is difficult to apply in practice without an effective way of computing the tangent cone of a particular closed set at a point.

Since Nagumo's theorem, a number of techniques have been developed that establish forward invariance of particular classes of sets. In [69], Prajna *et al.* coined the term barrier certificates and showed that for a differentiable function B, $B \leq 0$ is forward invariant with respect to a system of differential equations if the Lie derivative of B along the vector field of that system is negative at $B = 0$.[2]. The premise $B = 0$ in

[1] A set S is forward invariant with respect to a system of differential equations if every solution starting in S remains in S.

[2] The Lie derivative of B along the vector field $f(\mathbf{x})$ is defined by $\nabla B \bullet f$ and gives the rate of change along solutions of the system $\dot{\mathbf{x}} = f(\mathbf{x})$

this rule comes with a number of drawbacks. First, as the authors note in [69], it renders the set of barrier certificates non-convex, making it inefficient to automatically search for barrier certificates for a given system. Second, as is explored in [29], this premise makes it challenging to generalize the rule to sets built using finite boolean combinations of polynomial inequalities. Finally, only considering the Lie derivative of B exactly at the boundary of the set prevents the rule from being extended to sampled-data systems using the approach we employ in Chapter 4. Because of the first reason, the authors of [69] also give a formulation of barrier certificates without the premise $B = 0$.

Platzer's differential induction [62, 63] provides a generalization of this weaker formulation of barrier certificates (without the premise $B = 0$) to finite boolean combinations of polynomial inequalities. The total differential operator from [62, 63] and the more recent rewrite rule-based formulation from [65] provide an automatic mechanism for computing Lie-derivatives of finite boolean combinations of polynomial inequalities. This automation is very useful but fails to handle piecewise functions as we do in Chapter 4. In addition, differential induction does not generalize the more powerful exponential condition-based barrier certificates from [40] that we use in Chapter 4. Instead, differential induction must be augmented with differential auxiliaries [64] to handle systems that exponentially decay towards the invariant boundary. Most significantly, the exponential condition-based barrier certificates provide a natural mechanism for quantifying the invariant violation from a continuous-time approximation of a sampled-data system, as we do in Chapter 4. It is not clear how to do this with differential induction, as the condition for this proof rule does not imply that trajectories of the system move towards the invariant region when they start outside (i.e. after a violation).

Recent work from the control theory community has produced rules that are less conservative than differential induction for inequalities (not boolean combinations). In [40], Kong *et al.* present exponential condition-based barrier certificates that we adapt in Chapter 4. In [5] Ames *et al.* present what they term reciprocal barrier functions. Rather than going to zero at the invariant boundary, these functions become unbounded at the invariant boundary. As noted in [94], this makes reciprocal barrier functions unsuited for verifying robust safety properties, as modeling error can result in a trajectory reaching a state in which the barrier function is undefined. Barrier Lyapunov functions [84] suffer from the same robustness issue as they too grow to infinity at the invariant boundary. Our adaptation of barrier certificates to sampled-data systems in Chapter 4 can be viewed as a robustness property – modeling error is introduced by a continuous time approximation of the controller. However, neither of the robustness notions covered in [94] are sufficient for sampled-data systems as they do not address additive control disturbances.

Taly and Tiwari [83] present a number of induction rules based on the Lie derivative that are complete for various classes of invariants. They focus on rules that are effectively computable, in contrast with Nagumo's theorem. However, all of their rules consider the Lie derivative only at the boundary of the invariant set. As in [69], this makes them unsuited for reasoning about robust safety since disturbances can push the system outside the boundary of the invariant set. The premises of the rules in [83] express nothing about the behavior of the system after such violations. Similarly, [49] presents a necessary and complete rule that only considers points at the boundary of the invariant set. For our setting, this suffers from the same drawback as [83].

In [29], Ghorbal, Sogokon, and Platzer explore the relative deductive power of a number of the above rules for verifying positive invariance. They also present a mechanism for reasoning about Lie derivatives of piecewise functions. In contrast with our handling of piecewise functions in Chapter 4, the mechanism from [29] can handle piecewise functions that are not everywhere differentiable. However, the mechanism does not apply to exponential condition-based barrier certificates, and thus would be unsuitable for sampled-data systems. Moreover, this mechanism has not been implemented in a formal verification framework like Coq.

Very recently, Dai *et al.* [20] have proposed a generalization of exponential condition-based barrier certificates from [40]. In particular, rather than showing that the Lie derivative of the barrier function is at most proportional to its value, one shows that the Lie derivative is at most a polynomial function of its value for an appropriately chosen polynomial. The authors present a family of such polynomials, for which the condition from [40] is a special case. In related work, Zeng *et al.* [96] present a formulation of barrier functions in which the barrier function must be a Darboux polynomial.[3]. These papers raise an interesting direction for future work: can such techniques be adapted to sampled-data systems to produce a more powerful proof rule than that in Chapter 4?

Some of the work in this section [20, 40, 96] also presents automation for generating polynomial inequalities satisfying their respective barrier function conditions. Since it is designed for continuous time systems, the automation from [40] could be applied directly to satisfy condition (i) of Theorem 4.1 in Chapter 4. We did not need this automation as we were able to design a barrier function manually. However, this connection provides evidence for the value of Theorem 4.1. By decomposing the problem into an intersample property and a property of the continuous time approximation, we are able to take advantage of control theory techniques for continuous time. Also, our work in Chapter 3 is complementary to this automation – the theorems in that chapter provide a mechanism for decomposing problems into more manageable components to address scalability issues.

The work in [5, 94] also gives the notion of a control barrier function, a technique for synthesizing a controller from a barrier function. In this technique, the controller is initially left unspecified and later synthesized from the barrier certificate condition, e.g.

[3] B is a Darboux polynomial with respect to the vector field $f(\mathbf{x})$ if $\nabla B \bullet f = c(\mathbf{x}) \cdot B(\mathbf{x})$ for some polynomial c

the exponential condition from [40]. This is essentially the technique we used in building assumption 4.1 from Chapter 4.

Related to the modularity rules in Chapter 3, a recent paper [93] addresses the challenge of composing two control barrier functions in a way that ensures a controller exists satisfying both conditions. This is analogous to our conjunctive composition. The key property is called control sharing and is related to progress from Chapter 3. The technique from that paper only applies to two control barrier functions and continuous time controllers. However, generalizing it to an arbitrary number of controllers and combining it with our Theorem 4.1 from Chapter 4 could provide an alternate technique for conjunctive composition of sampled-data systems.

Sloth *et al.*[80] also present a mechanism for compositional verification of continuous time systems using barrier certificates. In addition to focusing on continuous time systems rather than sampled-data systems, this work differs from parallel composition in Chapter 3 by considering systems whose components interact via interconnection inputs rather than by sharing a single control input that must satisfy a composition of inequalities.

It is important to note that the only formal implementation of any of the inductive conditions in this section is Platzer's differential induction in the KeYmaera X proof assistant [67]. Implementing these conditions formally is important. As noted in [83], unsound continuous induction rules have been published and used before their unsoundness was discovered.

Finally, some of the above approaches [69, 40] also have generalizations to arbitrary hybrid systems to ensure that the barrier certificate changes value appropriately after discrete transitions of the system. While powerful, these rules do not allow one to verify the controller with respect to a continuous time approximation and separately reason about the error introduced by this approximation, as we do in Chapter 4. Instead, the controller must explicitly take into account intersample behavior in order to ensure that the barrier certificate satisfies the appropriate condition between samples.

5.5 Sampled-Data Systems

Barrier functions are part of a general class of functions called storage functions [90] that are used to reason about dynamical systems by assigning a scalar value to each state and analyzing how the value of the functions dissipates along trajectories of the system. There has been a variety of work on storage function approaches for reasoning about sampled-data systems. Much of this work, including [15, 74], focuses on stability of sampled-data systems using Lyapunov or Lyapunov-like functions, which is an important but orthogonal concern. The most relevant to this dissertation is the work by Laila, Nešić, and Teel [43] on a general framework for analyzing the behavior of dissipation inequalities for sampled-data systems. The exponential condition-based barrier functions from [40], which we adapt in Chapter 4, are a particular type of dissipation inequality. The framework in [43] applies to controllers that are designed using emulation: the controller is designed with respect to a continuous time approximation of the system and the implemented digitally in a sampled-data system. The authors then provide a series of theorems that quantify the effect of emulation on dissipation inequalities. This is essentially the same approach as Theorem 4.1 in Chapter 4. The framework in [43] is far more general than our model in Chapter 4, as it handles disturbances, controllers that maintain state, etc. However, the theorems in [43], when specialized to exponential condition-based barrier certificates and our less general model of sampled-data systems, apply only to states bounded by some constant magnitude. This restriction is not required by Theorem 4.1. Moreover, none of the work in [43] has been formalized in a proof assistant like Coq. Finally, the theorems in [43] only characterize dissipation inequalities over a single sampling period, while Theorem 4.1 bounds the invariant violation over an infinite time horizon. Nevertheless, real systems often permit conservative constant bounds on the magnitude of the state, and it would be interesting to formalize the framework of [43] in Coq and extend it to infinite time horizons to determine how the violation bounds compare to those of Theorem 4.1.

Some work analyzes sampled-data systems by treating them as time-delay systems [27]. Prajna *et al.* adapted their barrier certificate formulation to time delay systems [68]. However, the model of time delay systems in [68] is not as general as [27] and crucially cannot express systems with discontinuous delay, which is necessary for sampled-data systems.

There has also been work on algorithms for computing the discriminating kernel[4] of a sampled-data systems for an invariant set [54, 55, 31]. The discriminating kernel characterizes the set of initial states of the system for which there exists a control law that keeps all trajectories inside the invariant set. The algorithms in [54, 31] only compute the discriminating kernel for a finite time horizon and thus provides a weaker guarantee than the infinite horizon safety guarantees in this dissertation. The algorithm in [55] can, in certain cases reach a fixed point thus guaranteeing safety over an infinite time horizon. However, there are no characterization of the class of systems for which such a fixed point will be reached. In addition, formalizing soundness of the algorithms from these papers in a proof assistant would be challenging, as they do not produce a witness that can be efficiently checked for correctness, as in the case of algorithms for generating barrier certificates [20, 40, 96].

Model checking has also been employed specifically for sampled-data systems. In [76], Silva and Krogh present sampled-data hybrid automata – the specialized model permits a more efficient model checking algorithm using the *CheckMate* model checker [77] than general purpose hybrid system model checkers. However, the model

[4] The discriminating kernel is sometimes known as the viability kernel.

checker can only guarantee safety over a finite time horizon. Similarly, in [78], Simko and Jackson present a model checking algorithm for sampled-data systems that is based on Taylor models of the continuous dynamics and SMT solvers rather than automata. Their algorithm also only guarantees safety over a finite time horizon.

An alternate approach for verifying safety of sampled-data systems was explored in [97] based on timed relational abstractions. In this approach, automation constructs relations that map the state of the system at a sample time to the state of the system at the next sample time. However, this model assumes that the inter sample time is fixed rather than bounded and more importantly, only verifies properties of states exactly at the sample times. An additional mechanism would be needed to verify properties of states between samples.

5.6 Acknowledgments

Thanks to André Platzer, Nathan Fulton, and Andrew Sogokon for answering numerous questions to help put this work in context.

This chapter includes and expands upon material as it appears in International Conference on Formal Methods and Models for System Design 2015. Ricketts, Daniel; Malecha, Gregory; Alvarez, Mario M.; Gowda, Vignesh; Lerner, Sorin, ACM Press, 2015. International Conference on Embedded Software 2016. Ricketts, Daniel; Malecha, Gregory; Lerner, Sorin, ACM Press, 2016. The dissertation author was the primary investigator and author of these papers.

Chapter 6

Conclusions and Future Work

We have presented a series of proof rules for safety verification of sampled-data systems and evaluated them on various versions of the sampled-data double integrator, an important benchmark in control theory. Chapter 2 gave rules for discrete induction specialized to sampled-data systems and acyclic composition of components. Chapter 3 described a general framework for building sampled-data systems using conjunction (parallel composition), disjunction (alternative composition), and substitution (reuse). Finally Chapter 4 revisited an atomic building block from previous chapters, using a better performing but more complex position bounding controller to motivate a stronger continuous induction theorem for sampled-data systems.

All of the work described in this dissertation was done in the expressive and foundational context of the Coq proof assistant. This expressiveness was essential, as it allowed us to add domain-specific reasoning techniques as theorems and dispatch side conditions using formal proofs, rather than using axioms or tactics and dispatching side conditions using unverified decision procedures. In summary, the work described in this document was in support of the following:

> **Thesis:** Domain-specific proof rules implemented in a higher-order proof assistant simplify reasoning without compromising soundness for sampled-data systems.

There are a number of exciting future research directions for this work. In the rest of this section, we discuss some of these directions.

Horizontal composition Chapter 3 explored composition of sampled-data systems, particularly those controlling overlapping sets of actuators. However, the proof rules of Chapter 3 were only evaluated on the simpler but poorer performing atomic position (height) controller from Chapter 2, not on the more complex but better performing position controller from Chapter 4. It would be valuable to explore composition and re-use of the more complex controller from Chapter 4, as this would be an important step towards verification of the full Ardupilot geofencing module. A promising approach is to combined ideas from [93], which explores composition of barrier certificates for continuous time systems, with theorems from Chapter 4, which explore barrier certificates for sampled-data systems. This might address a particular shortcoming in Chapter 3: the composition proof rules did not leverage any of the differential invariants/barrier certificates used to verify subcomponents.

Vertical composition In Section 4.1, we described how the real implementation of the Ardupilot controller does not directly set acceleration but instead outputs desired velocity commands to an inner loop velocity controller. To simplify reasoning the outer loop controller was designed under the approximation that the inner loop controller instantaneously achieves any desired velocity command. The informal justification for this approximation is that, as long as the velocity commands do not change too quickly, the velocity dynamics are sufficiently fast relative to position dynamics that the approximation error will not cause significant violations of position constraints. It would be valuable to formalize this argument: under what conditions can the position constraint-enforcing outer loop controller be vertically composed with the inner loop velocity controller, and how much constraint violation does this cause?

Answering this question will require determining what properties the inner loop velocity controller must satisfy. The informal argument is based on the velocity dynamics being sufficiently fast – this suggests that the inner loop controller must satisfy a fast convergence property, such as local exponential stability. It would be valuable to verify such a property in the same formal framework as this dissertation, suggesting that the Coq framework needs to be extended with rules for reasoning about stability properties from control theory, particularly for sampled-data systems. Another promising approach is to explore the use of reference governors [11], a concept from control theory in an inner loop controller is augmented with a module (the reference governor) that modifies commands (references) to the controller in order to guarantee desired state constraints.

Verified implementations All of the systems verified in this dissertation have contained high-level models of controllers rather than executable code. Our own experience has demonstrated that bugs can occur at all levels of the development process, not just in the high-level design. Based on the ideas in Chapter 4, we implemented a geofence module for Ardupilot that prevents a vehicle from leaving a specified safe area, regardless of what the pilot does. This important safety feature required extending the one-spatial dimension constraints from Chapter 4 to an arbitrary 2D polygon and to a circular fence. At the core of this implementation is a constraint solver that chooses a safe velocity as close as possible to the pilot's command while satisfying the constraints of each edge of the fence (or circle). Our original C++ implementation passed the review process of the Ardupilot developers, was merged into the development, and run in the current public release of Ardupilot. Unfortunately, we later discovered that the constraint solving implementation is incorrect.[1]

[1] The bug is documented in the following github issues: https://github.com/ArduPilot/ardupilot/issues/4429 and https://github.com/ArduPilot/ardupilot/issues/4807.

This experience suggests that it is valuable to have end-to-end formal verification results, from the high level design all the way to the low level implementation. Coq is well suited for this task, as it is expressive enough for both the high level models (as demonstrated in this dissertation) as well as executable code verification [46, 7], even including floating point arithmetic [14]. Such an end-to-end verification result would be challenging for a standalone hybrid system verification tool as it would need to interface with another tool for formalizing and reasoning about code. We expect that a shallow encoding of a hybrid system logic in Coq will facilitate this task, as it will ease the process of integrating with other Coq developments for reasoning about low level code, such as [7, 14].

Probability All controllers in this dissertation require an estimate of the state of the system. Since sensors are not perfect, these estimates are inherently probabilistic. In Chapter 2, we presented an approach to reason about systems in the presence of sensor error when there is a concrete bound on that error. However, this can be overly pessimistic in certain cases and inaccurate in others. The core sensor fusion algorithms used to estimate a system's state based on sensor readings (e.g. Kalman filters) are based on probability [79]. It would be valuable to formalize these approaches and then reason about state constraint enforcing controllers in the presence of probability. Again, the expressiveness of Coq provides a promising framework for this direction.

More examples and other domains We evaluated the sampled-data proof rules in this dissertation on various instantiations of a single important application: the double integrator. While this is a benchmark application, it would be valuable to evaluate the proof rules on other sampled-data systems. A promising starting point would be the large collection of sampled-data systems that have been verified in the KeYmaera and

KeYmaera X proof assistants, e.g. [51, 42]. This will help build more evidence and understanding of when domain-specific proof rules simplify reasoning and reduce the proof burden for sampled-data systems.

In addition, while sampled-data can be used to model many real world systems, there are many systems that fall outside the domain. One interesting example is the domain of event-triggered systems.[2] There is potential for domain-specific proof rules to simplify reasoning for this and other classes of systems. Developing these rules could be an important step towards making formal verification a practical technique for real cyber-physical systems.

[2] Sampled-data systems are sometimes called time-triggered systems.

Bibliography

[1] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Trans. Program. Lang. Syst.*, 16(5):1543–1571, September 1994.

[2] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, May 1995.

[3] E. Abraham-Mumm, U. Hannemann, and M. Steffen. Verification of hybrid systems: formalization and proof rules in PVS. In *ECCS '01*, pages 48–57, 2001.

[4] Rajeev Alur and Thomas A. Henzinger. Modularity for timed and hybrid systems. In *CONCUR '97*, volume 1243, pages 74–88. Springer Berlin Heidelberg, 1997.

[5] A. D. Ames, X. Xu, J. W. Grizzle, and P. Tabuada. Control barrier function based quadratic programs for safety critical systems. *IEEE Transactions on Automatic Control*, PP(99):1–1, 2016.

[6] Abhishek Anand and Ross Knepper. ROSCoq: Robots powered by constructive reals. *ITP'15*, 15:2015, 2015.

[7] Andrew W. Appel and Robert Dockins and Aquinas Hobor and Josiah Dodds and Xavier Leroy and Sandrine Blazy and Gordon Stewart and Lennart Beringer. *Program Logics for Certified Compilers*. April 2014.

[8] Nikos Arechiga, James Kapinski, Jyotirmoy V. Deshmukh, André Platzer, and Bruce H. Krogh. Forward invariant cuts to simplify proofs of safety. In Alain Girault and Nan Guan, editors, *EMSOFT*, pages 227–236. IEEE, 2015.

[9] Nikos Aréchiga, Sarah M. Loos, André Platzer, and Bruce H. Krogh. Using theorem provers to guarantee closed-loop system properties. In Dawn Tilbury, editor, *ACC*, pages 3573–3580, 2012.

[10] Jean-Pierre Aubin. *Viability theory*. Springer Science & Business Media, 2009.

[11] A. Bemporad. Reference governor for constrained nonlinear systems. *IEEE Transactions on Automatic Control*, 43(3):415–419, Mar 1998.

[12] Frederic Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *TYPES*, volume 4502 of *LNCS*, pages 48–62. Springer Berlin Heidelberg, 2007.

[13] Brandon Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völp, and André Platzer. Formally verified differential dynamic logic. In Yves Bertot and Viktor Vafeiadis, editors, *Certified Programs and Proofs - 6th ACM SIGPLAN Conference, CPP 2017, Paris, France, January 16-17, 2017*, pages 208–221. ACM, 2017.

[14] S. Boldo and G. Melquiond. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pages 243–252, July 2011.

[15] Tongwen Chen and Bruce A Francis. Input-output stability of sampled-data systems. *IEEE Transactions on Automatic Control*, 36(1):50–58, 1991.

[16] Tongwen Chen and Bruce A. Francis. *Optimal Sampled-Data Control Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[17] Xin Chen, Sriram Sankaranarayanan, and Erika Abraham. Flow* 1.2: More effective to play with hybrid systems. In *ARCH14-15*, volume 34 of *EPiC Series in Computing*, pages 152–159, 2015.

[18] Pieter Collins, Milad Niqui, and Nathalie Revol. A Taylor Function Calculus for Hybrid System Analysis: Validation in Coq (Extended Abstract), 2010.

[19] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA+ proofs. In *International Symposium on Formal Methods*, pages 147–154. Springer Berlin Heidelberg, 2012.

[20] Liyun Dai, Ting Gan, Bican Xia, and Naijun Zhan. Barrier certificates revisited. *Journal of Symbolic Computation*, 80, Part 1:62 – 86, 2017. SI: Program Verification.

[21] Nuh Aygun Dalkiran, Moshe Hoffman, Ramamohan Paturi, Daniel Ricketts, and Andrea Vattani. Common knowledge and state-dependent equilibria. In *Algorithmic game theory*, pages 84–95. Springer Berlin Heidelberg, 2012.

[22] Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 235–248, New York, NY, USA, 2014. ACM.

[23] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] Rayna Dimitrova and Rupak Majumdar. Deductive control synthesis for alternating-time logics. EMSOFT '14, pages 14:1–14:10, New York, NY, USA, 2014. ACM.

[25] 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. CAV'11, pages 379–395. Springer-Verlag, 2011.

[26] Goran Frehse. PHAVer: algorithmic verification of hybrid systems past HyTech. *STTT*, 10(3):263–279, 2008.

[27] Emilia Fridman, Alexandre Seuret, and Jean-Pierre Richard. Robust sampled-data stabilization of linear systems: an input delay approach. *Automatica*, 40(8):1441 – 1446, 2004.

[28] Herman Geuvers, Adam Koprowski, Dan Synek, and Eelis van der Weegen. Automated machine-checked hybrid system safety proofs. In *ITP '10*, pages 259–274, 2010.

[29] Khalil Ghorbal, Andrew Sogokon, and André Platzer. A hierarchy of proof rules for checking positive invariance of algebraic and semi-algebraic sets. *Computer Languages, Systems and Structures*, 47(1):19–43, 2017.

[30] Jeremy H. Gillula, Gabriel M. Hoffmann, Huang Haomiao, Michael P. Vitus, and Claire J. Tomlin. Applications of hybrid reachability analysis to robotic aerial vehicles. *The International Journal of Robotics Research*, 30(3):335–354, 2011.

[31] Jeremy H. Gillula, Shahab Kaynama, and Claire J. Tomlin. Sampling-based approximation of the viability kernel for high-dimensional linear sampled-data systems. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC '14, pages 173–182, New York, NY, USA, 2014. ACM.

[32] L. Grune. Input-to-state dynamical stability and its lyapunov function characterization. *Automatic Control, IEEE Transactions on*, 47(9):1499–1504, Sep 2002.

[33] T. Gurriet and L. Ciarletta. Towards a generic and modular geofencing strategy for civilian uavs. In *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 540–549, June 2016.

[34] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[35] Thomas A. Henzinger. The theory of hybrid automata. In *LICS '96*, pages 278–292, 1996.

[36] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. In *CAV '97*, 1997.

[37] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *J. Comput. Syst. Sci.*, 57(1):94–124, 1998.

[38] C. A. R. Hoare. Proof of a program: Find. *Commun. ACM*, 14(1):39–45, January 1971.

[39] Jesper Bengtson. ChargeCore. https://github.com/jesper-bengtson/ChargeCore. Accessed: 2017-02-12.

[40] Hui Kong, Fei He, Xiaoyu Song, William N. N. Hung, and Ming Gu. *Exponential-Condition-Based Barrier Certificate Generation for Safety Verification of Hybrid Systems*, pages 242–257. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[41] Soonho Kong, Sicun Gao, Wei Chen, and Edmund Clarke. dReach: δ-Reachability Analysis for Hybrid Systems. pages 200–205. Springer Berlin Heidelberg, 2015.

[42] Yanni Kouskoulas, David Renshaw, André Platzer, and Peter Kazanzides. Certifying the safe design of a virtual fixture control algorithm for a surgical robot. HSCC '13, pages 263–272, New York, NY, USA, 2013. ACM.

[43] Dina S. Laila, Dragan Nei, and Andrew R. Teel. Open- and closed-loop dissipation inequalities under sampling and controller emulation. *European Journal of Control*, 8(2):109 – 125, 2002.

[44] Leslie Lamport. The temporal logic of actions. *TOPLAS*, 16(3):872–923, 1994.

[45] Leslie Lamport. Real time is really simple. Technical report, MSR-TR-2005-30, Microsoft Research, 2005.

[46] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[47] Daniel Liberzon. *Switching in systems and control*. Springer Science & Business Media, 2012.

[48] Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. A Calculus for Hybrid CSP. APLAS'10, pages 1–15, Berlin, Heidelberg, 2010. Springer-Verlag.

[49] Jiang Liu, Naijun Zhan, and Hengjun Zhao. Computing semi-algebraic invariants for polynomial dynamical systems. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 97–106, New York, NY, USA, 2011. ACM.

[50] Carolos Livadas and Nancy A. Lynch. Formal verification of safety-critical hybrid systems. In *HSCC '98*, 1998.

[51] Sarah M. Loos, André Platzer, and Ligia Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. FM'11, pages 42–56, Berlin, Heidelberg, 2011. Springer-Verlag.

[52] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O Automata. *Inf. Comput.*, 185(1):105–157, August 2003.

[53] André Oliveira Maroneze. *Certified Compilation and Worst-Case Execution Time Estimation*. PhD thesis, Rennes 1, 2014.

[54] Ian M Mitchell and Shahab Kaynama. An improved algorithm for robust safety analysis of sampled data systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 21–30. ACM, 2015.

[55] Ian M Mitchell, Shahab Kaynama, Mo Chen, and Meeko Oishi. Safety preserving control synthesis for sampled data systems. *Nonlinear Analysis: Hybrid Systems*, 10:63–82, 2013.

[56] Stefan Mitsch and Andr Platzer. ModelPlex: Verified Runtime Validation of Verified Cyber-Physical System Models. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 199–214. Springer International Publishing, 2014.

[57] Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger, and André Platzer. A component-based approach to hybrid systems safety verification. In Erika Abraham and Marieke Huisman, editors, *IFM*, volume 9681 of *LNCS*, pages 441–456. Springer, 2016.

[58] Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger, and André Platzer. Change and delay contracts for hybrid system component verification. In Marieke Huisman and Julia Rubin, editors, *FASE*, LNCS. Springer, 2017.

[59] Olaf Mller and Thomas Stauner. Modelling and verification using linear hybrid automata – a case study. *Mathematical and Computer Modelling of Dynamical Systems*, 6(1):71–89, 2000.

[60] Q. Nguyen and K. Sreenath. Exponential control barrier functions for enforcing high relative-degree safety-critical constraints. In *2016 American Control Conference (ACC)*, pages 322–328, July 2016.

[61] Milad Niqui and Olga Tveretina. Modular Development of Hybrid Systems for Verification in Coq. In *HSCC '08*, HSCC '08, pages 638–641. Springer-Verlag, 2008.

[62] Andr Platzer. *Logical analysis of hybrid systems: proving theorems for complex dynamics*. Springer Publishing Company, Incorporated, 2010.

[63] André Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *J. Log. Comput.*, 20(1):309–352, 2010. Advance Access published on November 18, 2008.

[64] André Platzer. The structure of differential invariants and differential cut elimination. *Logical Methods in Computer Science*, 8(4):1–38, 2012.

[65] André Platzer. A uniform substitution calculus for differential dynamic logic. In Amy Felty and Aart Middeldorp, editors, *CADE*, LNCS. Springer, 2015.

[66] André Platzer and Edmund M Clarke. *Formal verification of curved flight collision avoidance maneuvers: A case study*. Springer, 2009.

[67] Andr Platzer. KeYmaera X: A Hybrid Systems Theorem Prover. http://www.ls.cs.cmu.edu/KeYmaeraX/. Accessed: 2015-04-28.

[68] S. Prajna and A. Jadbabaie. Methods for safety verification of time-delay systems. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 4348–4353, Dec 2005.

[69] Stephen Prajna and Ali Jadbabaie. *Safety Verification of Hybrid Systems Using Barrier Certificates*, pages 477–492. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[70] Venkatesh G Rao and Dennis S Bernstein. Naive control of the double integrator. *IEEE Control Systems*, 21(5):86–97, 2001.

[71] D. Ricketts, G. Malecha, M. M. Alvarez, V. Gowda, and S. Lerner. Towards verification of hybrid systems in a foundational proof assistant. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 248–257, Sept 2015.

[72] Daniel Ricketts, Gregory Malecha, and Sorin Lerner. Modular deductive verification of sampled-data systems. In *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, pages 17:1–17:10, New York, NY, USA, 2016. ACM.

[73] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Automating Formal Proofs for Reactive Systems. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 452–462, New York, NY, USA, 2014. ACM.

[74] Alexandre Seuret and Matthew M. Peet. Stability analysis of sampled-data systems using Sum of Squares. *IEEE Transactions on Automatic Control*, 58(6):1620–1625, January 2013.

[75] Lui Sha, Ragunathan Rajkumar, and Michael Gagliardi. Evolving dependable real-time systems. In *Aerospace Applications Conference*, pages 335–346. IEEE, 1996.

[76] B. I. Silva and B. H. Krogh. Modeling and verification of hybrid systems with clocked and unclocked events. In *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No.01CH37228)*, volume 1, pages 762–767 vol.1, 2001.

[77] B Izaias Silva, Keith Richeson, Bruce Krogh, and Alongkrit Chutinan. Modeling and verifying hybrid dynamic systems using checkmate. In *Proceedings of 4th International Conference on Automation of Mixed Processes*, volume 4, pages 1–7, 2000.

[78] Gabor Simko and Ethan K. Jackson. A bounded model checking tool for periodic sample-hold systems. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC '14, pages 157–162, New York, NY, USA, 2014. ACM.

[79] Dan Simon. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley-Interscience, 2006.

[80] Christoffer Sloth, George J. Pappas, and Rafael Wisniewski. Compositional safety analysis using barrier certificates. In *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '12, pages 15–24, New York, NY, USA, 2012. ACM.

[81] Eduardo D. Sontag. Input to state stability: Basic concepts and results. In *Nonlinear and Optimal Control Theory*, pages 163–220. Springer, 2006.

[82] P. Tabuada. An approximate simulation approach to symbolic control. *IEEE Transactions on Automatic Control*, 53(6):1406–1418, July 2008.

[83] Ankur Taly and Ashish Tiwari. Deductive verification of continuous dynamical systems. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[84] Keng Peng Tee, Shuzhi Sam Ge, and Eng Hock Tay. Barrier lyapunov functions for the control of output-constrained nonlinear systems. *Automatica*, 45(4):918 – 927, 2009.

[85] The ArduPilot Team. ArduPilot. http://ardupilot.com. Accessed: 2017-01-25.

[86] Vadim Utkin and Hoon Lee. {CHATTERING} {PROBLEM} {IN} {SLIDING} {MODE} {CONTROL} {SYSTEMS}. *{IFAC} Proceedings Volumes*, 39(5):1 –, 2006. 2nd {IFAC} Conference on Analysis and Design of Hybrid Systems.

[87] Norbert Vlker. Towards a HOL Framework for the Deductive Analysis of Hybrid Control Systems, 2000.

[88] Shuling Wang, Naijun Zhan, and Liang Zou. *An Improved HHL Prover: An Interactive Theorem Prover for Hybrid Systems*, pages 382–399. Springer International Publishing, Cham, 2015.

[89] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem-overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[90] Jan C. Willems. Dissipative dynamical systems part i: General theory. *Archive for Rational Mechanics and Analysis*, 45(5):321–351, 1972.

[91] T. Wongpiromsarn, U. Topcu, and A. Lamperski. Automata theory meets barrier certificates: Temporal logic verification of nonlinear systems. *IEEE Transactions on Automatic Control*, PP(99):1–1, 2015.

[92] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals and Systems*, 1(1):13–30, 1988.

[93] X. Xu. Control sharing barrier functions with application to constrained control. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 4880–4885, Dec 2016.

[94] Xiangru Xu, Paulo Tabuada, Jessy W. Grizzle, and Aaron D. Ames. Robustness of control barrier functions for safety critical control. *IFAC-PapersOnLine*, 48(27):54 – 61, 2015.

[95] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[96] Xia Zeng, Wang Lin, Zhengfeng Yang, Xin Chen, and Lilei Wang. Darboux-type barrier certificates for safety verification of nonlinear hybrid systems. In *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, pages 11:1–11:10, New York, NY, USA, 2016. ACM.

[97] Aditya Zutshi, Sriram Sankaranarayanan, and Ashish Tiwari. Timed relational abstractions for sampled data control systems. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 343–361, Berlin, Heidelberg, 2012. Springer-Verlag.