Verifiably Safe Autonomy for Cyber-Physical Systems

Nathan Fulton

CMU-CS-18-125

November 2018

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
André Platzer, Chair
Jeremy Avigad
Goran Frehse, ENSTA ParisTech
Zico Kolter
Stefan Mitsch

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers CNS-1035800, CNS-1054246, and CNS-1446712, by the U.S. Army Research Office under grant number W911NF0910273, by the Department of Transportation under grant number DTRT12GUTC11, by the Silicon Valley Community Foundation under grant numbers 20151438675388, 2016158691, and 2017174874, by the Air Force Office of Scientific Research under grant number FA95501610288, and by the Defense Advanced Research Projects Agency under grant number FA875018C0092.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Cyber-Physical Systems, Hybrid Systems, Autonomous Systems, Formal Verification, Differential Dynamic Logic, Automated Theorem Proving, Reinforcement Learning

For Abby

Abstract

This thesis demonstrates that autonomous cyber-physical systems that use machine learning for control are amenable to formal verification.

Cyber-physical systems, such as autonomous vehicles and medical devices, are increasingly common and increasingly autonomous. Designing safe cyber-physical systems is difficult because of the interaction between the discrete dynamics of control software and the continuous dynamics of the vehicle's physical movement. Designing safe autonomous cyber-physical systems is even more difficult because of the interaction between classical controls software and machine learning components.

Formal methods capable of reasoning about these hybrid discrete-continuous dynamics can help engineers obtain strong safety guarantees about safety-critical control systems. Several recent successes in applying formal methods to hybrid dynamical systems demonstrate that these tools provide a promising foundation for establishing safety properties about planes, trains, and cars. However, existing theory and tooling does not explain how to obtain formal safety guarantees for systems that use reinforcement learning to discover efficient control policies from data. This gap in existing knowledge is important because modern approaches toward building cyber-physical systems combine machine learning with classical controls engineering to navigate in open environments.

This thesis introduces KeYmaera X, a theorem prover for hybrid systems, and uses KeYmaera X to obtain verified safety guarantees for control policies generated by reinforcement learning algorithms. These contributions enable strong safety guarantees for optimized control policies when the underlying environment matches a first-principles model.

This thesis also introduces an approach toward providing safety guarantees for learned control policies even when reality deviates from modeling assumptions. The core technical contribution is a new class of algorithms that blend learning and reasoning to update models in response to newly observed dynamics in the environment. When models are updated online, we leverage verification results about the original but incorrect model to ensure that there is a systematic relationship between the optimization objective and desired safety properties. When models are updated offline, formal verification results are preserved and explainable environmental models are synthesized. These contributions provide verifiable safety guarantees for systems that are controlled by policies obtained through reinforcement learning, justifying the use of reinforcement learning in safety-critical settings.

Acknowledgments

My Ph.D. would not have been possible without the gracious support of many colleagues, family, and friends.

First and foremost, I would like to thank André Platzer for his support throughout my graduate education. André is an extraordinary advisor who has provided invaluable guidance that has shaped me as a researcher, as an educator, and as a person. Stefan Mitsch has also played an incredibly important role in my research and in the research of everyone else who builds upon KeYmaera X. I also thank the rest of my committee for their helpful feedback and thoughts.

My colleagues past and present have been an invaluable source of support. The Logical System Lab is a wonderful environment for doing research. My best days of graduate school were often due to the many wonderful people who have passed through the lab: Sarah Loos, Stefan Mitsch, Jean-Baptiste Jeannin, Stefan Mitsch, João Martins, Jan-David Quesel, Andrew Sogokon, Jonathan Laurent, Fabian Immler, Khalil Ghorbal, Luis Garcia, Lorenz Sahlmann, Andreas Müller, Eric Zawadzki, Ran Ji, Brandon Bohrer, Yong Kiam Tan, Jessica Packer, and Markus Völp. I also want to thank the students of 15-424 and, in particular, David Bayani and Viren Bajaj.

Many other graduate students at Carnegie Mellon played an important role in my graduate education: Anna Gommerstadt, Ellis Hershkowitz, Cyrus Omar, Vittorio Perera, Michael and Erin Maass, Ivan Ruchkin, and Roykrong Sukkerd.

Many people outside of Carnegie Mellon have made my time in Pittsburgh enjoyable and have provided emotional support throughout the past several years. Among these many people, Marcello Codianni, Dan and Samantha Ehrmann, Ben Letson, Youngmin Park, and Michael Lindsey played a special role in helping me through the highs and lows of graduate school. Thank you.

I thank my undergraduate mentors for the special role they played in shaping my development as a researcher and as a person: Mark Mahoney, Charlotte Chell, Erlan Wheeler, Jonathan Aldrich, and Cyrus Omar.

My family has supported me in too many ways to list. Thank you, Mike, Katy, Heather, Melissa, and Laura.

Finally, this document would not have been possible without Abby's support. Abby very likely knows many sections of this document better than I do. Her pragmatic support throughout my graduate education was invaluable. But more importantly, she has never stopped supporting me. From across the country, from across the table, and from across the world, Abby has always been there in the smallest ways and in the biggest ways. I could never ask for a better partner or friend. Thank you.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The automotive and aeronautical industries have continually improved the energy-efficiency, safety, comfort and automation of vehicles. Achieving these improvements required substantially increasing the size and complexity of vehicle software. The growing use of software in these safety-critical settings inspired the development of mathematical models – called *hybrid systems* – that model the interaction between discrete software systems and the continuous systems under control [7, 77]. Hybrid systems provide a fruitful formalism for stating and proving safety properties about systems that combine discrete computation with continuous control.

Over the past decade, designers of vehicles have moved on from low-level control problems. Tomorrow's software systems not only help control the engine and the brakes, but also make high level decisions about where and how a vehicle should move. Advanced driver-assist systems are already deployed. Most major automobile manufacturers are experimenting with fully autonomous vehicles. Designers of planes and trains are also deploying partially autonomous vehicles and experimenting with fully autonomous systems. The future of mobility is autonomous. These autonomous systems make extensive use of machine learning, such as reinforcement learning, to control in open environments. Therefore, designing safe autonomous cyber-physical systems requires establishing safety properties about systems that use reinforcement learning and other optimization techniques for control.

Unlike traditional software engineering domains where light-weight quality assurance mechanisms (e.g., testing) often suffice, best practices for safety-critical systems suggest the use of formal verification. Ideally, developers of safety-critical systems should construct a model of the system under control and then write a formal, computer-checked proof that their control software satisfies key safety properties with respect to the underlying model. For example, a developer might construct a system of differential equations describing how a car behaves and then prove that a piece of control software prevents the car from entering an unsafe state. Formal proofs of relevant safety properties ensure that a system is *verifiably safe*.

This dissertation demonstrates that **autonomous cyber-physical systems that use reinforcement learning for control are amenable to formal verification**.

Fully autonomous systems that make use of both offline and online learning will need to come with strong safety guarantees. Therefore, developing formal methods that are capable of providing safety guarantees for modern learning algorithms is an important challenge.

Despite recent successes in modeling and verifying safety properties about cyber-physical

systems [101, 128], existing hybrid systems verification approaches are not directly applicable to modern reinforcement learning algorithms. Classical software verification tools (e.g., model checkers [36], deductive program verification tools such as KeY and Dafny [3, 123], and general-purpose theorem provers such as Coq, Isabelle, and Lean [47, 132, 141]) are capable of verifying properties about reinforcement learning algorithms. The mode of use for each of these tools is different, so the exact properties that can be verified and the difficulty of the verification task vary greatly between these tools and paradigms. However, none of these systems currently provide a productive environment for hybrid systems verification because they lack proof search algorithms and proof constructive primitives specialized to the task of verifying properties about control algorithms and especially differential equations.

Conversely, existing verification tools for cyber-physical systems (such as KeYmaera X [64, 154]) are specialized to hybrid systems analysis and therefore do not provide a reasonable setting for describing algorithms such as those found in the reinforcement learning literature. This dissertation demonstrates how to use hybrid systems analysis tools to provide safety guarantees for systems that use reinforcement learning without resorting to encoding the entire reinforcement learning algorithm as an explicit part of the hybrid system.

Verifiably safe autonomy is an epistemically challenging goal. Verification results for cyber-physical systems are always stated with respect to a model of the world. Obtaining strong safety guarantees in situations that are anticipated by system designers are certainly required for safe autonomy, but truly autonomous systems must provide safety guarantees even when there are modeling gaps between design-time assumptions and observed reality.

Part I of this dissertation introduces the KeYmaera X theorem prover for hybrid systems and explains how theorem proving may be used to obtain highly trustworthy safety proofs for on-model control; i.e., how to guarantee system safety whenever system designers can provide a sufficiently accurate model of the world. Chapter 3 introduces KeYmaera X and Chapter 4 discusses the design of its interactive proof development language, including a discussion of several large proof automation developments. Chapter 5 discusses an extension to the theorem prover's base logic that has since been extended to enable inter-operation with other provers, allowing for end-to-end verification of control software by leveraging formlizations of compilers and arithmetic found in these other systems.

Unfortunately, behaving well in anticipated scenarios is not enough. Autonomy implies the ability to act safely even in situations that were not explicitly modeled by system designers. Therefore, safe autonomous systems must be able to act well *off-model*; i.e., when environmental modeling assumptions are violated. Achieving this goal requires closing these modeling gaps via a combination of offline and online reasoning. Part II of this dissertation explores three ways in which these *modeling gaps* may be closed.

The first modeling gap closed by this dissertation is the gap between verified descriptions of control policies and actually implementable control policies. The verification results obtained in Part I are stated with respect to highly nondeterministic descriptions of the controller. These control descriptions will concisely characterize *all* of the safe actions available in each state, but will not generally explain which of these actions should be taken in order to achieve a high-level goal. Chapter 7 explains how to close this gap between safe control and efficient control by leveraging verification results to obtain safety guarantees for reinforcement learning algorithms.

The second modeling gap considered by this dissertation is the gap between a system of dif-

ferential equations and a physical system's actual behavior. Chapter 8 considers the special case in which the system designer identifies many possible models of the world and the control system must safely choose between the available models. Chapter 9 moves beyond the assumption that an accurate environmental model is provided, and instead considers how a combination of learning and reasoning can be used to generate both a model and a controller that satisfy a safety property starting from only input/output examples and a global safety specification.

By explaining how to obtain trustworthy correctness proofs for on-model control and how to leverage these proofs during off-model control, this thesis demonstrates that autonomous cyber-physical systems that use reinforcement learning for control are amenable to formal verification.

Part I

Trustworthy Verification for Hybrid Dynamics

Chapter 2

Background on Hybrid Systems Verification

Safety-critical systems should not be deployed without high confidence in safe system operation. Deploying systems without strong safety assurances has significant ethical ramifications; an inadequate understanding of safe autonomy poses a serious risk for near-term and mid-term deployments of autonomous systems in safety-critical domains [142]. For this reason, deploying systems without certain safety assurances runs afoul of industry standards in both automotive [55] and aeronautics [160] domains.

Formal verification is an approach toward building this confidence by way of mathematical proof. In a typical verification effort, the scientist or engineer builds a mathematical model of the safety-critical system, identifies mathematical characterizations of key safety properties, and then proves that control software enforces these safety properties with respect to the model. Modern systems are far too large and complex for hand-written proofs to suffice, so designers interested in formal proofs use analysis software such as model checkers or theorem provers to generate computer-checked proofs.

This verificationist approach toward ensuring safety immediately raises two important questions: how can we know that the analysis software itself is trustworthy, and how do we know that our mathematical models of reality are accurate?

Part I of this thesis focuses on the first question: how can we build highly trustworthy analysis software for hybrid systems? We introduce the KeYmaera X theorem prover, which demonstrates how to analyze cyber-physical systems. Unlike existing hybrid systems analysis tools, KeYmaera X is built on a small soundness-critical core with a defined logical foundations. This ensures that analyzing a model with KeYmaera X strictly increases our confidence in a system's design, instead of merely shifting uncertainty about a model to uncertainty about the correctness of the model analysis tool. We begin by recalling the logical foundations upon which KeYmaera X is built.

2.1 Hybrid Programs

Cyber-physical systems are characterized by the interaction between discrete control software and a physical system under control. Hybrid dynamical systems [7, 149] mix discrete and continuous dynamics, providing a compelling mathematical formalism for describing the mixture of discrete control and continuous movement that characterizes cyber-physical systems.

Hybrid programs [147, 148, 149] are a programming language model of hybrid dynamics. Hybrid programs extend nondeterministic imperative programs (i.e., regular programs) with differential equations. Typically the discrete portion of a hybrid program describes the behavior of a software controller and the continuous portion of a hybrid program describes the smooth movement of the physical system under control. In verification tasks, the controller is often stated as a nondeterministic set of safe control actions; nondeterministic controllers are simpler to verify because large numbers of possible control inputs – even uncountably infinite numbers of possible control inputs – can be concisely grouped together as a guarded nondeterministic assignment.

Hybrid systems are an expressive mathematical tool capable of accurately modeling a broad range of physical and sociological phenomena. This thesis focuses on the use of hybrid systems to model control systems, such as those found in partially and fully autonomous vehicles. An informal description of hybrid programs is given in Table 2.1.

Program Statement	Meaning
$\alpha;\beta$	Sequentially composes α and β.
$\alpha \cup \beta$	Executes either α or β.
α^*	Repeats α zero or more times.
$x := \theta$	Evaluates θ and assigns result to x.
$x := *$	Assigns an arbitrary real value to x.
$\{x_1' = \theta_1, ..., x_n' = \theta_n \& F\}$	Continuous evolution[1].
$?F$	Aborts if F is not true.

Table 2.1: Hybrid Programs.

Example 1 (The Linear Car Hybrid Program)**.** *One of the simplest hybrid programs is a model of a car moving along a straight line, choosing a new acceleration* $a \in \{-B, A\}$ *at least once every* T *seconds.*

Listing 2.1: The Linear Car Program.

```
{
  //Choose a new acceleration accel ∈ {−B, A}
  { accel := -B ∪ accel := A }
  //Reset the timer.
  c := 0;
  //System dynamics describing linear motion for at most T time.
  { pos'=vel, vel'=accel, c'=1 & c ≤ T }
}* //loop 0 or more times
```

[1] A continuous evolution along the differential equation system $x_i' = \theta_i$ for an arbitrary duration within the region described by formula F.

Line 3 of Example 1 allows only two choices of acceleration: maximum acceleration A or maximum braking $-B$. A more realistic model allows the choice of any acceleration $accel \in [-B, A]$. This behavior is expressible in dL by changing line 3 of Example 1 to read:

$$\texttt{accel := *; ?(-B} \leq \texttt{accel} \land \texttt{accel} \leq \texttt{A).}$$

This program first allows `accel` to take on *any* value, and then immediately asserts that this new value of `accel` must be between $-B$ and A. In this way, a nondeterministic controller can characterize an infinite number of control choices without succumbing to the curse of dimensionality (and/or requiring discretization to enable safety and other reachability analyses).

After the controller is executed, a timer is reset to 0 ($c := 0$). This timer will tick during the continuous fragment of the hybrid program ($c' = 1$ on line 7). The next run of the controller will happen, at the latest, before $c = T$, because the flow of the differential equations are restricted to the domain where $c \leq T$.

Line 7 of Example 1 demonstrates how a system's physical behavior can be characterized using differential equations subject to evolution domain constraints. The derivative of the car's position is `vel` and the velocity `vel` changes according to the choice of acceleration `accel`. The system is constrained to the domain $c \leq T$ so that the controller will run again before T time units elapse, at which point a new acceleration will be chosen and the timer will be reset to 0. Finally, the star on line 8 sequences the controller, the timer reset, and the differential equations zero or more times.

2.2 Differential Dynamic Logic

Differential dynamic logic (dL) [147, 148, 149, 152] is a first-order multimodal logic for specifying and proving properties of hybrid programs. Each hybrid program α has modal operators $[\alpha]$ and $\langle\alpha\rangle$, which express reachability properties of program α. The formula $[\alpha]\phi$ expresses that the formula ϕ is true in all states reachable by the hybrid program α. Similarly, $\langle\alpha\rangle\phi$ expresses that the formula ϕ is true after some execution of α.

Definition 1 (Formulas)**.** *The formulas of* dL *are defined as follows (with θ, η as terms, p as predicates, C as quantifier symbols, and ϕ, ψ ranging over* dL *formulas):*

$$\begin{array}{lll} \langle \phi, \psi \rangle \ ::= \ \theta \geq \eta & & \textit{Comparisons} \\ \quad | \quad p(\theta_1, \ldots, \theta_k) & & \textit{Predicates} \\ \quad | \quad C(\phi) & & \textit{Symbols} \\ \quad | \quad \neg\phi \mid \phi \land \psi \mid \forall x \ \phi \mid \exists x \ \psi & & \textit{First-order Logic} \\ \quad | \quad [\alpha]\phi \mid \langle\alpha\rangle\phi & & \textit{Modalities} \end{array}$$

The grammar given by Def. 1 is the minimal. In practice, hybrid systems reachability properties formalized in dL also make use of disjunction, implication, and equivalences:

$$\theta, \psi ::= \ldots \mid \phi \lor \psi \mid \phi \rightarrow \psi \mid \phi \leftrightarrow \psi$$

Each of these are definable, but the system and technology discussed throughout the rest of this dissertation is designed in terms of the extended base grammar.

The denotational semantics of dL, discussed in the next section, will assign to each formula the set of states in which the formula evaluates to true. For example,

$$[\![x > 0 \land y > 0]\!] = \{s \mid s(x) > 0 \land s(y) > 0\}$$

. The intuitive meaning of formulas that do not contain modalities matches that of classical first order logic. Before defining the semantics of dℒ, we consider another example that will recur throughout this document as a simple canonical example of a hybrid systems control task.

Example 2. *Consider a car, moving in a straight line, that must stop before arriving at a stop sign. This property is expressible in dℒ by the formula in the following listing, where* `stopSignPos` *is the location of the stop sign and* `safe`: $\mathbb{R}^3 \to$ `Bool` *is a to-be-defined formula describing the set of positions and velocities in which it is safe to accelerate.*

Listing 2.2: A Safety Specification for the Linear Car.

```
A > 0 ∧ B > 0 ∧ pos < stopSignPos ∧ safe(pos, vel, -B) →
[
  {
    //Choose a new acceleration accel ∈ [−B, A],
    //accelerating only when safe.
    { accel := -B ∪ accel := *; ?safe(pos, vel, accel) }
    //Reset the timer.
    c := 0;
    //System dynamics describing linear motion for at most T time.
    { pos'=vel, vel'=accel, c'=1 & c ≤ T }
  }* //loop 0 or more times
]pos < stopSignPos
```

Example 2 is a reachability property of a simple hybrid dynamical system with the canonical form `init`→`[{ctrl; plant}*]safe`. Here, the system begins within some initial set `init`, repeatedly executes a discrete controller `ctrl` followed by some continuous physical dynamics described by the system of differential equations `plant`, and always ends in a state that satisfies `safe`.

The premise (or precondition) of Example 2 describes a set of initial states for the hybrid dynamical system. Aside from bounds on constants, we must also assume that the car begins in a configuration such that continuously braking will bring the car to a complete stop before reaching the stop sign. The precondition `safe(pos, vel, -B)` expresses this assumption. The conclusion of the implication states that *every execution* of Example 1 ends in a state where $pos < stopSignPos$.

Nondeterminism in Hybrid Systems: A Prelude To Safe Learning

Notice that the model is nondeterministic and is specifically designed to capture a safety property. The controller model does not choose a single action; instead, the model describes an entire set of possible safe actions. The physical model does not choose an exact amount of time to follow the flow of the ODEs; instead, any flow along the ODE up to time $0 \leq c \leq T$ is possible. Furthermore, the verification condition does not mention fuel efficiency, passenger comfort, or other important fitness criteria.

Verification conditions such as Example 2 capture only the critical safety property of the system. Both of these modeling choices are crucial for ensuring the tractability of hybrid systems analyses; a fully deterministic description of the system would prove intractable to verify automatically. Even interactive verification might be too labor intensive in many cases.

The reinforcement learning algorithms developed in this thesis provide a way to choose efficient resolutions to this nondeterminism that satisfy safety constraints while also optimizing for other objectives. Reinforcement learning may be viewed as an optimizing compiler for nondeterministic and underspecified models. The safety constraints specified in $\mathsf{d}\mathcal{L}$ may be viewed as a constraint this optimization process.

2.3 Semantics of $\mathsf{d}\mathcal{L}$

The semantics of differential(-form) dynamic logic is given by the semantics of terms θ, the semantics of formulas φ, and the semantics of programs α. Formulas may occur in programs (within tests) and programs may occur in formulas (within modalities). Therefore, the definitions of formulas and programs are mutually recursive. We recall these definitions from [152].

Definition 2 (Semantics of Terms)**.** *The semantics of a term θ with interpretation I in state s is given by the following inductive definition.*

1. $I[\![x]\!](s) = s(x)$ *for variables* $x \in V$
2. $I[\![f(\theta_1, \ldots, \theta_n)]\!](s) = I(f)(I[\![\theta_1]\!](s), \ldots, I[\![\theta_n]\!](s))$ *for function symbols* $f \in I$
3. $I[\![\theta + \nu]\!] = I[\![\theta]\!] + I[\![\nu]\!]$
4. $I[\![\theta \cdot \nu]\!] = I[\![\theta]\!] \cdot I[\![\nu]\!]$
5. $I[\![(\theta)']\!](s) = \sum_{x \in V} s(x') \frac{\partial I[\![\theta]\!]}{\partial x}(s)$

Variables are given meaning by the state s and function symbols are given meaning by the interpretation I. The semantics of addition and multiplication are straight-forward; division and subtraction are definable in terms of addition and multiplication. Restricting division by zero is trivial in theory. However, actually implementing a theorem prover that avoids unsoundness resulting from division by zero – without over-burdening the user or hobbling automation – is challenging in practice. We discuss our approach toward this problem in Chapter 3.

The semantics of primed terms ($(\theta)'$) is defined as the differential of θ; i.e., as a sum over the spatial partial derivatives of the (finitely many) variables occurring in the term. The fact that this differential form corresponds to the time-derivative of θ along the solution to any differential equation is proven in [152, Lemma 35].

Semantics of Formulas The semantics of a $\mathsf{d}\mathcal{L}$ formula φ, denoted $[\![\varphi]\!]$, defines for each interpretation I the set of all states in which φ is true under I. The interpretation I may contain a finite set of n-ary predicate symbols $I(p) \subseteq \mathbb{R}^n$. The interpretation I defines the meaning of quantifier symbols C as functionals $I(C) : \mathscr{P}(S) \to \mathscr{P}(S)$ mapping sets of states where C is true to sets of states where the application of $I(C)$ is true. We denote by $s_{x \to r}$ the state that agrees with s on the value of every variable except x, which takes on the values r in $s_{x \to r}$.

The semantics of formulas are defined simultaneously with the semantics of hybrid programs because formulas may contain programs (in modalities) and programs may contain formulas (in tests). The semantics of each program $I[\![\alpha]\!]$ is a state transition relation that characterizes the reachable states of a program. The interpretation may define state transitions encoded as program constants $a : \mathscr{P}(S \times S)$.

Definition 3 (Semantics of Formulas)**.** *The semantics of a formula φ within interpretation I is the set of states in which φ is true, and is inductively defined as follows:*

1. $I[\![\theta \geq \nu]\!] = \{s \in S : I[\![\theta]\!](s) \geq I[\![\theta]\!](s)\}$
2. $I[\![p(\theta_1, \ldots, \theta_n)]\!] = \{s \in S : (I[\![\theta_1]\!](s), \ldots, I[\![\theta_n]\!](s)) \in I(p)\}$
3. $I[\![C(\varphi)]\!] = I(C)(I[\![\varphi]\!])$ *for quantifier symbols* C
4. $I[\![\neg\varphi]\!] = S \backslash (I[\![\varphi]\!])$
5. $I[\![\varphi \wedge \psi]\!] = I[\![\varphi]\!] \cap I[\![\psi]\!]$
6. $I[\![\exists x \varphi]\!] = \{s \in S : s_{x \to r} \in I[\![\varphi]\!]$ *for some* $r \in \mathbb{R}\}$
7. $I[\![\forall x \varphi]\!] = \{s \in S : s_{x \to r} \in I[\![\varphi]\!]$ *for all* $r \in \mathbb{R}\}$
8. $I[\![\langle\alpha\rangle\varphi]\!] = I[\![\alpha]\!] \circ I[\![\varphi]\!] = \{s : q \in I[\![\varphi]\!]$ *for some* q *such that* $(s, q) \in I[\![\alpha]\!]\}$
9. $I[\![[\alpha]\varphi]\!] = I[\![\alpha]\!] \circ I[\![\varphi]\!] = \{s : q \in I[\![\varphi]\!]$ *for all* q *such that* $(s, q) \in I[\![\alpha]\!]\}$

Definition 4. *The semantics of a program α in interpretation I is defined inductively as follows:*

1. $I[\![a]\!] = I(a)$ *for program constants* a
2. $I[\![x := \theta]\!] = \{(s, s_{x \to r}) : r = I[\![\theta]\!](s)\}$
3. $I[\![?\psi]\!] = \{(s, s) : s \in I[\![\psi]\!]\}$
4. $I[\![\alpha \cup \beta]\!] = I[\![\alpha]\!] \cup I[\![\beta]\!]$
5. $I[\![\alpha; \beta]\!] = I[\![\alpha]\!] \circ I[\![\beta]\!]$
6. $I[\![\alpha^*]\!] = \cup_{n \in \mathbb{N}} I[\![\alpha^n]\!]$ *with* $\alpha^{n+1} \equiv \alpha^n; \alpha$ *and* $\alpha^0 \equiv ?true$
7. $I[\![x' = \theta \& \psi]\!] = (s, q)$ *such that:*
   - $s = \varphi(0)$ *on* $V \backslash \{x'\}$ *and*
   - $q = F(r)$ *for some function* $F : [0, r] \to V$ *of duration* r *satisfying* $I, F \models x' = \theta \& \psi$

   where $I, F \models x' = \theta \& \psi$ *iff:*
   - $F\zeta \in I[\![x' = \theta \& \psi]\!]$,
   - $F(0) = F(\zeta)$ *on* $V \backslash \{x, x'\}$ *for all* $0 \leq \zeta \leq r$, *and*
   - $\frac{dF(t)(x)}{dt}(\zeta)$ *exists and is equal to* $F(\zeta)(x')$ *for all* $0 \leq \zeta \leq r$.

Assignment and nondeterministic choice are illustrative examples of how the semantics of hybrid programs are defined. Assignment maps each state s to a new state q that is identical to s except for the new value of x.

The semantics of nondeterministic choice $\alpha \cup \beta$ maps each state s to *two* new states: one which results from executing α and the other which results from executing β. The semantics of a system of differential equations within a domain is defined by the solution to the system; $\mathsf{d}\mathcal{L}$ only allows ordinary differential equations and the interpretation I of function symbols is assumed to contain only smooth functions; therefore, the solution F is guaranteed to exist [125]. The assumption in $\mathsf{d}\mathcal{L}$ that $\frac{dF(t)(x)}{dt}(\zeta)$ exists is analogous to the way in which many theorem provers and logics assume that division by zero is undefined. These design choices are analogous in the following sense. In theory, these are unproblematic assumptions that the reader is asked to verify whenever writing down a differential equation (or analogously, a division operation). However, in practice, checking this property locally is challenging and requires syntactic constraints to ensure that the theorem prover's implementation is sound and complete. We discuss this issue at greater length in Chapter 3.

2.4 The dℒ Hilbert Calculus

Proving specifications such as Example 2 requires a sound set of axioms and proof rules for dℒ. The KeYmaera X theorem prover implements a Hilbert-style [92, 93] proof calculus with three components: a set of axioms, as well as proof rules for performing uniform substitutions and contextual rewriting proof rule.

Axioms The axioms and proof rules of dℒ from [151] are enumerated in Figures 2.1 and 2.2. These axioms are designed to support compositional proofs of dℒ formulas by decomposing formulas and programs into their constituent parts. For example, the axiom of nondeterministic choice $[a \cup b]p(\bar{x}) \leftrightarrow [a]p(\bar{x}) \wedge [b]p(\bar{x})$ decomposes a reachability property for a nondeterministic hybrid system into two reachability properties, one for each of the constituent programs in the nondeterministic choice.

In typical verification tasks, the axioms in Fig. 2.1 are used to symbolically decompose regular programs and the axioms in Fig. 2.2 enable various reasoning techniques for handling ordinary differential equations. For example, we used the axioms in Fig. 2.2 to implement an Ordinary Differential Equation solver based on logical deductions and have also implemented reasoning techniques based on differential invariants [64]. The CE proof rule allows for equational rewriting of equivalent subformulas, whereas CQ and CT allow for equational rewriting of equal terms.

Uniform Substitutions Typical axiom systems contain a countably infinite number of axioms generated from a finite set of axiom schemata. The Hilbert axiomatization of dℒ does not have axiom schemata; rather, it has a finite number of axioms, a finite number of proof rules (represented as sets of formulas), and a proof rule called Uniform Substitution (US) for performing soundness-preserving substitutions on these axioms. For example, consider the axiom of nondeterministic choice

$$[a \cup b]p(\bar{x}) \leftrightarrow [a]p(\bar{x}) \wedge [b]p(\bar{x})$$

Notice that a and b are concrete atomic programs, and that $p(\bar{x})$ is a concrete predicate. The choice axiom alone is not enough to prove

$$[x := 0 \cup x := 1]x \geq 0 \leftrightarrow [x := 0]x \geq 0 \wedge [x := 1]x \geq 1$$

because this axiom is not an schematic meta-formula; rather, it is a concrete formula and substitutions are defined separately via a substitution proof rule.

Uniform substitutions [35, 152] provide a mechanism for using axioms that mention generic programs and predicates to prove theorems that contain concrete programs and formulas.

Uniform substitutions are mappings from functions $f(\bar{x})$ to terms, predicate symbols $p(\bar{x})$ to formulas, quantifier symbols $C(\_)$ to formulas, and program constants a to programs where $\bar{x}$ is a set of variables that may be bound and $\_$ a reserved quantifier symbol of arity zero. We may also use $p(\cdot)$ where $\cdot$ means that p may mention *any* variable. The substitution $a \rightsquigarrow x := 0$ substitutes any occurrence of the program variable a with program $x := 0$. And $p(\cdot) \rightsquigarrow x \geq 0$

$\langle\cdot\rangle$ $\langle a\rangle p(\bar{x}) \leftrightarrow \neg[a]\neg p(\bar{x})$

$[:=]$ $[x := f]p(x) \leftrightarrow p(f)$

$[?]$ $[?q]p \leftrightarrow (q \rightarrow p)$

$[\cup]$ $[a \cup b]p(\bar{x}) \leftrightarrow [a]p(\bar{x}) \wedge [b]p(\bar{x})$

$[;]$ $[a; b]p(\bar{x}) \leftrightarrow [a][b]p(\bar{x})$

$[^*]$ $[a^*]p(\bar{x}) \leftrightarrow p(\bar{x}) \wedge [a][a^*]p(\bar{x})$

K $[a](p(\bar{x}) \rightarrow q(\bar{x})) \rightarrow ([a]p(\bar{x}) \rightarrow [a]q(\bar{x}))$

I $[a^*](p(\bar{x}) \rightarrow [a]p(\bar{x})) \rightarrow (p(\bar{x}) \rightarrow [a^*]p(\bar{x}))$

V $p \rightarrow [a]p$

G $\dfrac{p(\bar{x})}{[a]p(\bar{x})}$

$\forall$ $\dfrac{p(x)}{\forall x\, p(x)}$

MP $\dfrac{p \rightarrow q \quad p}{q}$

CT $\dfrac{f(\bar{x}) = g(\bar{x})}{c(f(\bar{x})) = c(g(\bar{x}))}$

CQ $\dfrac{f(\bar{x}) = g(\bar{x})}{p(f(\bar{x})) \leftrightarrow p(g(\bar{x}))}$

CE $\dfrac{p(\bar{x}) \leftrightarrow q(\bar{x})}{C(p(\bar{x})) \leftrightarrow C(q(\bar{x}))}$

US $\dfrac{\varphi}{\sigma(\varphi)}$

Figure 2.1: Axioms and Proof Rules of Differential Dynamic Logic; C is a quantifier symbol, p, q are predicate symbols, c, f, g are function symbols, and σ is an admissible uniform substitution. Admissibility conditions on uniform substitutions are defined in [152].

substitutes a predicate $p(\theta)$ with a formula $\theta \geq 0$ for any argument term θ. For example, the substitution

$$a \rightsquigarrow x := 0$$
$$b \rightsquigarrow x := 1$$
$$p(\bar{x}) \rightsquigarrow x \geq 0$$

with $\bar{x} = \{x\}$ applied to the choice axiom $[a \cup b]p(\bar{x}) \leftrightarrow [a]p(\bar{x}) \wedge [b]p(\bar{x})$ produces the formula

$$[x := 0 \cup x := 1]x \geq 0 \leftrightarrow [x := 0]x \geq 0 \wedge [x := 1]x \geq 1$$

A substitution is *uniform* if it satisfies the constraints on the occurrences of free and bound variables given by [152, Definition 19, Fig. 1]. The uniformity constraint is required to ensure the soundness of the uniform substitution proof rule. Logical deductions in dL may appeal to the truth-preserving nature of substitutions via the US proof rule (Fig. 2.1).

Example 3 (Admissible and Clashing Substitutions)**.** *Restricting the US proof rule to admissible uniform substitutions is necessary for preserving the soundness of the calculus. Consider the substitution and formula*

$$\sigma = \{a \rightsquigarrow x := x - 1, p \rightsquigarrow x \geq 0\}$$
$$\phi \equiv p \rightarrow [a]p.$$

If σ were admissible for ϕ (it is not!), then the US proof rule would allow a proof of $x \geq 0 \rightarrow [x := x - 1]x \geq 0$

$$\dfrac{\dfrac{*}{p \rightarrow [a]p}}{x \geq 0 \rightarrow [x := x - 1]x \geq 0}$$

$$\begin{array}{rl}
\text{DW} & [x' = f(x)\,\&\,q(x)]q(x) \\
\text{DC} & \big([x' = f(x)\,\&\,q(x)]p(x) \leftrightarrow [x' = f(x)\,\&\,q(x) \wedge r(x)]p(x)\big) \leftarrow [x' = f(x)\,\&\,q(x)]r(x) \\
\text{DE} & [x' = f(x)\,\&\,q(x)]p(x,x') \leftrightarrow [x' = f(x)\,\&\,q(x)][x' := f(x)]p(x,x') \\
\text{DI} & [x' = f(x)\,\&\,q(x)]p(x) \leftarrow \big(q(x) \rightarrow p(x) \wedge [x' = f(x)\,\&\,q(x)](p(x))'\big) \\
\text{DG} & [x' = f(x)\,\&\,q(x)]p(x) \leftrightarrow \exists y\,[x' = f(x), y' = a(x)y + b(x)\,\&\,q(x)]p(x) \\
\text{DS} & [x' = f\,\&\,q(x)]p(x) \leftrightarrow \forall t{\geq}0\,\big((\forall 0{\leq}s{\leq}t\,q(x+fs)) \rightarrow [x := x + ft]p(x)\big) \\
[':=] & [x' := f]p(x') \leftrightarrow p(f) \\
+' & (f(\bar{x}) + g(\bar{x}))' \;=\; (f(\bar{x}))' + (g(\bar{x}))' \\
\cdot' & (f(\bar{x}) \cdot g(\bar{x}))' \;=\; (f(\bar{x}))' \cdot g(\bar{x}) + f(\bar{x}) \cdot (g(\bar{x}))' \\
\circ' & [y := g(x)][y' := 1]\big((f(g(x)))' = (f(y))' \cdot (g(x))'\big)
\end{array}$$

Figure 2.2: Differential Equation Axioms and Differential Axioms.

but this formula is clearly not valid. Conversely, consider the very similar substitution σ' and the formula φ:

$$\sigma' = \{a \rightsquigarrow x := x - 1, p(\bar{x}) \rightsquigarrow x \geq 0\}$$
$$\varphi \equiv [a]p(\bar{x})$$

for $\bar{x} = (x)$. Because σ' is φ-admissible, the US proof rule allows the deduction following

$$\frac{x \geq 0}{[x := x - 1]x \geq 0}$$

via a uniform substitution on the G proof rule.

Example 3 demonstrates that the US rule is not sound for naïve substitutions. A sound calculus must restrict uniform substitutions so that substitutions which introduce unsound deductions are not permitted. For this purpose, dℒ defines when a given substitution is *admissible* for a formula and restricts the US proof rule so that the rule is only applicable when the substitution σ is ϕ-admissible. The two cases in Example 3 demonstrate why admissibility of a substitution depends upon the formula to which a substitution is applied – a substitution may be sound for one formula and unsound for another.

The slight difference between the substitutions σ and σ' in Example 3 demonstrates the significance of the difference between p, $p(x)$, and $p(\bar{x})$. These three predicate symbols have different static semantics. The first symbol (p) has a nullary predicate symbol. The second ($p(x)$) has a predicate symbol where the variable x may occur freely, and the third ($p(\bar{x})$) has a predicate symbol where any $x \in \bar{x}$ may occur freely. These free variables of p continue to be permitted in its replacement. Additional free variables are allowed by the US rule under certain admissibility conditions (see [151, Fig. 1]).

The definition of admissibility depends upon the static semantics of dℒ formulas, so this difference in the static semantics of p, $p(x)$, and $p(\bar{x})$ is crucial when determining whether a substitution is admissible.

The explication of admissibility for uniform substitutions in $\mathsf{d}\mathcal{L}$ is critical for soundness. In this thesis we use $\mathsf{d}\mathcal{L}$ and its axiomatization [151] as implemented by KeYmaera X for verifying hybrid systems models of cyber-physical systems.

Chapter 3

The KeYmaera X Tool

Formal verification requires the use of analysis software, such as a model checker [36] or a theorem prover. This requirement is due to the fact that formal proofs about software grow very large – far too large to construct, check, or maintain by hand. Analysis software must be trustworthy; an untrustworthy tool merely translates uncertainty about the system under analysis into uncertainty about the analysis tool. Analysis software must also be useful; a maximally trustworthy analysis tool that cannot even establish simple properties about simple systems is not useful in practice.

This chapter introduces the KeYmaera X theorem prover for differential dynamic logic[1]. The distinguishing features of KeYmaera X are a small soundness-critical core that ensures the correctness of the system, and a tactics language called Bellerophon for implementing custom hybrid systems proof construction and proof search procedures on top of the small core. This combination of a small core with a hybrid systems proof programming environment make KeYmaera X the most trustworthy and extensible hybrid systems analysis tool available today. This chapter introduces the core and compares KeYmaera X to existing hybrid systems verification tools; the challenge of taming the complexity of the core to provide a productive theorem proving environment is taken up in Chapter 4.

3.1 The KeYmaera X Core

KeYmaera X [64] is structured to maintain a trustworthy core. The KeYmaera X core is trustworthy because it is a small and simple piece of software with a defined logical foundation.

The KeYmaera X implementation of the $\mathsf{d}\mathcal{L}$ Hilbert calculus is attractive from a soundness perspective because it is simple and small. The implementation contains two main components: a text file containing verbatim copies of axioms, and a small amount of Scala code implementing the proof rules (including Uniform Substitution). All reasoning executed by the KeYmaera X theorem prover runs through this small, soundness-critical core. Unlike existing foundations of hybrid systems, the Hilbert calculus of $\mathsf{d}\mathcal{L}$ easily enables sound theorem prover implementations. Unlike existing hybrid systems analysis tools, KeYmaera X isolates all soundness-critical rea-

[1] This chapter is based on the paper *KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems* by Fulton et al. [64].

soning in a small core consisting of simple axioms and proofs rules. An untrusted code base built around this core implements common proof construction and proof search techniques.

The KeYmaera X core is implemented in Scala, an ML-inspired language for the Java Virtual Machine. The core implementation has three major components: a set of axioms (see Fig. 2.1 and Fig. 2.2), a set of proof rules that explain how to use these axioms (e.g., the uniform substitution rule), and a definition of the static semantics of the logic that determine when these proofs rules are applicable (see Example 3). The remainder of this section explains how each of these components of dL are implemented in the KeYmaera X core.

Representing Proofs KeYmaera X represents proof states using a `Provable` object. `Provable` objects have a conclusion and a set of subgoals. Each conclusion and subgoal is a sequent of the form $\varphi_{-1}, \ldots, \varphi_{-n} \vdash \varphi_1, \ldots, \varphi_m$ where the subscripts are positional addresses that identify whether a formula is an assumption (negative) or a conclusion (positive).

When a `Provable` contains no subgoals, the conclusion sequent is a fact about dL. When there is a single conclusion of the form $\vdash \varphi$ and no subgoals, φ is a theorem of dL.

For example, a complete proof that $\vdash_{\mathsf{dL}} \forall x(x > 0 \vee x \leq 0)$ is represented by the Scala object

```
Provable("\forall x (x>0||x <= 0)".asFormula, Nil)
```

Only the core may create or modify `Provables`. The core creates provables – without proof – for facts that are verified by external real arithmetic decision procedures implemented in Mathematica and Z3. Therefore, although the KeYmaera X core is small, the total size of the trusted computing base includes the implementation of Z3 and/or the `Reduce` command of Mathematica [99, 171]. This reliance on external solvers is possible to eliminate. For example, Platzer, Quesel and Rümmer leverage a combination of Gröbner bases, the Positivstellensatz, and semi-definite programming to general witnesses for real arithmetic [156].

Conspicuously missing from the `Provable` representation of a proven fact is any record of the actual proof that justifies the result. Chapter 5 introduces an extension to dL, the Logic of Proofs for Differential Dynamic Logic ($\mathsf{LP}_{\mathsf{dL}}$), that makes such justifications explicit. Section 5.5 discusses how to implement $\mathsf{LP}_{\mathsf{dL}}$ outside of the soundness-critical core of KeYmaera X.

Implementation of Axioms The implementation of axioms in the KeYmaera X core is simple: a single string contains a list of all axioms within the KeYmaera X core. For example, the axiom for nondeterministic choice is

```
1 [a;++b;]p(||) ↔ ([a;]p(||) & [b;]p(||))
```

where `a,b` are program constants and `p(||)` is a predicate that may mention any variable.

Proof rules are implemented as mappings from formulas to (lists of) formulas. The most significant proof rule is the Uniform Substitution rule, which includes an implementation of the static semantics and substitution admissibility conditions of dL.

Unfortunately, getting real work done in this raw Hilbert calculus is nontrivial. Even simple properties have verbose proofs. For example, a proof of $[x := 0 \cup x := 1]x \geq 0$ is far from concise [61]:

$$\text{MP}\,\dfrac{\text{US}\,\dfrac{[\cup]\,\dfrac{*}{[a\cup b]p(\bar{x})\leftrightarrow[a]p(\bar{x})\wedge[b]p(\bar{x})}}{[x:=0\cup x:=1]x\geq 0\leftrightarrow[x:=0]x\geq 0\wedge[x:=1]x\geq 0} \qquad \Delta}{[x:=0\cup x:=1]x\geq 0}$$

with $\bar{x}=\{x\}$ where Δ is:

$$\text{Prop}\,\dfrac{\text{Prop}\,\dfrac{\Delta_1 \qquad \Delta_2}{[x:=0]x\geq 0\wedge[x:=1]x\geq 0}}{([x:=0\cup x:=1]x\geq 0\leftrightarrow[x:=0]x\geq 0\wedge[x:=1]x\geq 0)\rightarrow[x:=0\cup x:=1]x\geq 0}$$

where Δ_1 is

$$\text{MP}\,\dfrac{\text{US}\,\dfrac{[:=]\,\dfrac{*}{[x:=t]p(t)\leftrightarrow p(x)}}{[x:=0]x\geq 0\leftrightarrow 0\geq 0} \qquad \text{Prop}\,\dfrac{\mathbb{R}\,\dfrac{*}{0\geq 0}}{[x:=0]x\geq 0\leftrightarrow 0\geq 0\rightarrow[x:=0]x\geq 0}}{[x:=0]x\geq 0}$$

and Δ_2 is

$$\text{MP, Prop, US}\,\dfrac{[:=]\,\dfrac{*}{[x:=t]p(t)\leftrightarrow p(x)} \qquad \mathbb{R}\,\dfrac{*}{1\geq 0}}{[x:=1]x\geq 0}$$

This lengthy derivation (which *still* elides some details!) demonstrates that although the Hilbert calculus provides a compelling target for sound theorem prover implementations, even trivial hybrid systems reachability properties require extremely verbose proofs.

3.2 Bellerophon

Practical use of the dℒ Hilbert calculus requires careful design of interactive and automated theorem proving environments that leverage the simplicity and soundness of the Hilbert calculus while also enabling verification of complex reachability properties for realistic models of cyber-physical systems.

Enabling efficient automated theorem proving and productive interactive proving requires a mechanism for defining and composing common proof construction techniques. Bellerophon, introduced by Fulton et al. [65] and discussed at greater length in Chapter 4, is a programming language and standard library for constructing dℒ proofs in KeYmaera X. Bellerophon implements a high-level sequent calculus on top of the simpler dℒ Hilbert calculus, enabling human-readable proofs for realistic hybrid systems. The Bellerophon standard library also contains several automatic proof search procedures, called tactics, that can automatically prove properties about common subclasses of hybrid systems. Bellerophon combinators provide a mechanism for composing these building blocks in to proof construction procedures and proof search algorithms. As a result, the verbose proof presented above reduces to the very small Bellerophon program invoking tactics in the Bellerophon standard library:

Listing 3.1: Branching in Bellerophon proofs.

```
choiceb(1); andR(1); <( //Split proof: 1 subgoal per control choice
  //[x:=0]x>=0 case
  assignb(1); QE
  ,
  //[x:=1]x>=0 case
  assignb(1); QE
)
```

The above tactic splits the proof into one case for each program on either side of the nondeterministic choice operator $\cup$. In each of these cases, the assignment is symbolically executed to produce a purely arithmetic subgoal, which is then discharged using a decision procedure for real arithmetic. KeYmaera X interfaces with implementations of arithmetic solvers in Z3 [45] and Mathematica [99]. The primary automated theorem prover implemented in KeYmaera X, `master`, will automatically construct the above tactic.

KeYmaera X distinguishes between built-in tactics and composite tactics. A Built-in tactic is a piece of Scala code that directly manipulates `Provable` objects using the KeYmaera X core. A composite tactic is a tactic built by applying combinators to built-in tactics. For example, the above tactic is a composite tactic while `choiceb` and `assignb` are built-in tactics.

The major challenge addressed by KeYmaera X is retaining a productive theorem proving environment without expanding the theorem prover's core any more than necessary. The way in which Bellerophon answers this challenge is discussed further in Chapter 4.

3.3 Related Work on Hybrid Systems Verification

Trustworthy and productive hybrid systems theorem proving requires a small soundness-critical core, automation specific to hybrid systems, and a mechanism for composing this automation. Even though these ingredients can be found scattered across a multitude of theorem provers, their combination provides a novel tactical theorem proving technique for hybrid systems. Table 3.1 compares several tools along the dimensions that we identify as crucial to productive hybrid systems verification (SC indicates a soundness-critical dependency on user-defined tactics or on an external implementation of a more scalable arithmetic decision procedure).

Table 3.1: Comparison of KeYmaera X to Related Verification Tools and Provers.

Tool	Small Core	HS Library	HS Auto	Scriptable	External Tools
KeYmaera X	Yes	Yes	Yes	Yes	SC
Hybrid Systems Tools	No	No	Yes	No	SC
Theorem Provers[2]	Yes	No	No	Yes	No
$d\mathcal{L}$ in Isabelle,Coq[25]	Yes	Yes[3]	No[4]	Yes	No
KeYmaera 3	No	Yes	Yes	SC	SC

[3] E.g., Coq [132], Isabelle [140], HOL [82, 166], and Lean [46, 47].
[4] via encodings in $d\mathcal{L}$
[5] via KeYmaera X proof term extraction discussed in Chapter 5

Logics of Hybrid Systems Theorem provers for hybrid dynamics use deduction in a formal calculus to establish reachability properties about a programming languages model of hybrid dynamics. Differential Dynamic Logic and Hybrid CSP [126] both fit this paradigm. The first theorem prover for $d\mathcal{L}$ was KeYmaera[6] [154]. KeYmaera [3] was built as an extension to the KeY system [3].

Unlike KeYmaera X, KeY and therefore KeYmaera were not built on a small core and their tactics proving mechanisms are soundness-critical. Furthermore, the KeYmaera tool includes more soundness-critical external tooling dependencies than KeYmaera X. These factors significantly limit the trustworthiness of proofs obtained using KeYmaera. Perhaps more importantly, the lack of separation between prover automation and soundness-critical proof checking limits the extensibility of of KeYmaera relative to KeYmaera X because new proof search procedures implemented in KeYmaera are soundness-critical, whereas proof developers using KeYmaera X are free to speculate. The ability to speculatively modify a proof and later check the resulting proof for correctness plays an important enabling role in our work presented in Chapter 8.

Automation implemented in KeYmaera is not only soundness-critical, but is also expressed in a way that is not easy to interrogate. Unlike KeYmaera, KeYmaera X exposes a tactical proof programming language (discussed further in Section 4.2. The fact that tactics in KeYmaera X are themselves implemented in a tactical programming language means that these programs are amenable to manipulation. This is a key enabling feature for our work in Chapter 8.

General-Purpose Theorem Proving Differential dynamic logic is a specialized logic for stating and proving reachability properties of hybrid dynamical systems. Several researchers have proposed building a theory of hybrid systems within existing theorem provers. The most complete approach is that of Bohrer et al. [25], whose work is compared with KeYmaera X in Table 3.1. The approach of Bohrer et al. leverages purpose-built systems such as KeYmaera X while building an interface with the larger theories available in tools like Coq and Isabelle. Unlike embeddings of $d\mathcal{L}$ in other theorem provers, KeYmaera X is designed from the ground up with productive hybrid systems theorem proving in mind. For this reason, Bohrer et al. [25] currently export proofs from KeYmaera X into other systems and independently check the proofs there, instead of manually constructing proofs within their $d\mathcal{L}$ embeddings.

Some libraries implemented within general-purpose theorem provers are relevant to the analysis of continuous systems. For example, Immler and Hölzl formalized Picard-Lindelöf in Isabelle [96, 97]. Analyzing solvable differential equations is only a small part of hybrid systems verification; verifying cyber-physical systems requires reasoning about the interaction between control software and differential equations; Example 4 in Chapter 4 demonstrate that doing so is not always trivial.

Foster et al. [56] recently proposed a new approach toward verifying hybrid systems using a Hoare logic implemented in Isabelle. Foster et al. focus on unifying theories. Therefore, their current exploration of interactive and automated hybrid systems verification focuses on demonstrating that these properties are encodable within their framework, rather than on the

[6]Not to be confused with KeYmaera X, a clean-slate implementation that shares no code with KeYmaera, implements a different core calculus, and takes a different approach toward both interactive and automated theorem proving.

verification effort itself.

Delta Decidability The δ-decidability framework, introduced by Gao et al. [66], relaxes the decision problem for real arithmetic extended with special functions such as sine and cosine by introducing numerical perturbations. Decision procedures determine, for an arbitrary formula φ, whether the formula is true or false. The δ-decision procedure relaxes this problem by allowing numerical perturbations to φ. Instead of returning true or false, a δ-decision procedure will, for any formula φ containing only bounded quantifiers and for arbitrarily small δ, return either that φ is true or else will return that a small perturbation to the formula will make the formula false. The dReal tool [110] implements a δ-decision procedure for real arithmetic extended with special functions by using interval constraint propagation [22] as the base theory for a DPLL(t) solver [30]. Gao et al. extend this δ-decidability framework to the problem of bounded reachability analysis for hybrid systems [68], and the dReach tool [67] leverages dReal to implement this bounded reachability analysis.

The approach taken by dReal and dReach is similar to the earlier of Herde, Eggers, and Fränzle whose HySAT solver is also DPLL-based [57, 89].

Unlike dReach and the work of Gao et al. and Herde et al. on bounded reachability analysis for hybrid systems [68, 89], KeYmaera X is capable of verifying systems with unbounded quantifiers and can analyze systems with unbounded time horizons. Both of these are important features because many salient safety properties about hybrid systems cannot be stated in terms of bounded quantifiers and/or bounded time.

Unlike dReal and dReach, KeYmaera X isolates all soundness-critical reasoning to a small soundness-critical core and enables interactive analysis of hybrid systems. Also unlike dReal and dReach, KeYmaera X is capable of checking both safety and liveness properties of hybrid systems. This distinction is important to our work on reinforcement learning discussed in Part II because the ModelPlex algorithm that plays a central role in that approach relies on a hybrid systems liveness analysis.

Timed Automata Timed automata express a restricted subset of hybrid dynamics in which continuous dynamics are restricted to a finite set of real-valued resettable clocks. Tools such as UPPAAL [20, 21, 42, 119] and KRONOS [43] are able to automatically model-check properties of (networks of) timed automata. Unfortunately, timed automata are not expressive enough to capture the rich continuous dynamics typically found in cyber-physical systems in which both positional and time variables may have continuous dynamics. Hybrid systems include the full expressiveness of ordinary differential equations, which is necessary to capture many interesting controls problems including those which often occur when designing safety-critical control systems for automobiles and aircraft. Unlike tools based on timed automata, KeYmaera X is capable of checking both safety and liveness properties of both timed systems and hybrid systems.

Hybrid Automata and Reachability Hybrid automata [85] are a representation of hybrid dynamics based on automata theory. Several approaches toward model checking for hybrid automata have been proposed and implemented; all of these approaches attempt to compute a set

of a states that are reachable from an initial configuration. The reachable set is often over-approximated to make automated analysis tractable. Over-representations might be represented as ellipsoids [115, 116], convex polytopes, zonotopes [74], taylor models [31], and/or support functions [78]. Many of these approaches consider subsets of hybrid dynamics; e.g., PHAver [58] considers linear hybrid automata and SpaceEx [59] considers piecewise affine dynamics. KeYmaera X currently provides less automation[7] than these tools but is also not restricted to a subclass of hybrid systems. KeYmaera X also differs from these tools because it provides a robust interactive theorem proving environment; see Fulton et al. [65] Mitsch and Platzer [134]. A third and important distinction between KeYmaera X and both PHAver and SpaceEx is the ability of the latter tools to provide visual insights into a system's dynamics.

Several other tools also implement reachability analyses for hybrid automata. FLOW* [32, 33] is a reachability tool for hybrid automata that uses Taylor models to over-approximate reachable sets. The Compare Execute Check Engine (C2E2) [50] is an automatic verification tool that combines numerical simulation with over-approximation to check bounded-time properties of hybrid systems. Both of these tools have been used to verify large case studies (e.g., on large genetic networks and helicopter dynamics). The HyTech [8, 86] system is capable of automatically verifying temporal properties for some classes of hybrid systems.

Among tools based on hybrid automata, Ariadne [23] is notable because – like KeYmaera X– Ariadne aspires to provide both an analysis tool and a development environment for constructing new hybrid systems analyses. Unlike KeYmaera X, Ariadne does not isolate soundness-critical reasoning from user-defined verification algorithms.

A number of tools provide support for analysis of hybrid systems within languages or software packages. The MATLAB Hybrid Toolbox [19] contains a myriad of analysis and simulation tools, including a technique for designing model-predictive controllers for hybrid systems. The HyReach [130] tool is a MATLAB toolbox that approaches verification of linear hybrid systems by using support functions to compute reachable sets. Checkmate [137] implements a MATLAB toolbox for verifying, exploring, and simulating hybrid systems. The verification approach implemented in Checkmate is based on flow pipe approximations of over-approximated hybrid systems. S-Taliro [11] is a MATLAB toolbox for robustness analysis that uses Monte Carlo methods for stochastic testing of hybrid systems. HyPro [95, 164] is a C++ library for implementing hybrid systems analyses. HyPro implements state set representations that are useful for implemented reachability analyses for hybrid automata.

The Charon [10] system enables modular specification of hybrid systems by including primitives for composing agents and building hierarchies of behavior. Ptolemy [29, 52], focuses on modeling and simulation of cyber-physical systems with concurrent components. Unlike KeYmaera X, Charon and Ptolemy do not enable verification of hybrid systems.

Compared to tools for analyzing hybrid systems, KeYmaera X has several characteristics that are crucial to our work on safe reinforcement learning.

- Unlike any other tools except KeYmaera, KeYmaera X focuses on interactive deductive verification allowing users to tackle verification problems that are currently outside the scope of automatic analyses. To date, this includes most industrially relevant hybrid sys-

[7] This limitation is one of the implementation rather than the theory. Section 4.5.3 discusses how automation can be built on top of KeYmaera X using a combination of schematic dℒ formulas and Bellerophon tactics.

tems.

- Unlike any other tools except general purpose theorem provers, KeYmaera X contains a small soundness-critical core enabling complex proof automation without decreasing the trustworthiness of the prover. However, unlike general purpose theorem provers, KeYmaera X provides a robust tactics library for verifying hybrid systems and for building automated hybrid systems analysis tooling.
- Unlike all existing hybrid systems tools, KeYmaera X provides extensible automation expressed in a tactical programming language allowing us to systematically modify proof scripts. This will play a crucial role in our work on model update learning in Chapter 8.
- KeYmaera X is capable of both safety and liveness checking, enabling the ModelPlex algorithm for generating monitoring conditions that plays a crucial role in Chapter 7 and Chapter 8.
- Unlike [8, 32, 33, 43, 50, 86], KeYmaera X enables tactical and deductive theorem proving of both safety and liveness properties for hybrid systems. The tactical theorem proving paradigm is a particularly important aspect of KeYmaera X that we leverage to great effect in Chapter 8 and Chapter 9 to provide safety guarantees for hybrid systems.

3.4 Conclusion

KeYmaera X is a novel hybrid systems theorem prover that provides: *1)* a small foundational core; *2)* a library of high-level primitives automating common deductions (e.g., computing Lie Derivatives, computing and proving solutions of ODEs, propagating quantities across dynamics in which they do not change, automated application of invariant candidates, and conservation/symmetry arguments); and *3)* scriptable heuristic search automation. KeYmaera X can also automatically generate ODE invariants [155] and generate code via a verified toolchain [26]. This combination of features enables our approach toward safe reinforcement learning in cyber-physical systems.

Chapter 4

Bellerophon

The previous chapter introduced the KeYmaera X system and discussed the structure of the prover core at length. Although we acknowledged the difficulty of constructing a productive theorem prover based upon a small core, few details were given about how KeYmaera X overcomes this difficulty. This chapter[1] explains how the Bellerophon proof programming environment helps turn the soundness-preserving core of KeYmaera X into a productive theorem proving environment for establishing safety properties about hybrid dynamical systems.

Our presentation of Bellerophon begins with the syntax and semantics for the Bellerophon tactic combinator language, passes through an example verification effort exploiting Bellerophon's support for invariant and arithmetic reasoning for a non-solvable system, and culminates with a discussion of substantial proof engineering efforts undertaken using the Bellerophon proof engineering environment.

4.1 Introduction

Theorem proving is an attractive technique for verifying correctness properties of hybrid systems because it is applicable to a large class of hybrid systems [148]. Verification for hybrid systems is not semi-decidable, thus requiring human assistance along two major dimensions. First, general-case hybrid systems proving requires identifying invariants of loops and differential equations, which is undecidable in both theory and practice. Second, the remaining verification tasks consist of first-order logic over the reals with polynomial terms. Decision procedures exist which are complete in theory [37], but are only complete in practice if a human provides additional guidance. Because both these dimensions are essential to hybrid systems proving, innovating along these dimensions benefits a wide array of hybrid systems verification tasks.

We argue that trustworthy and productive hybrid systems theorem proving requires: *1)* a small foundational core; *2)* a library of high-level primitives automating common deductions (e.g., computing Lie Derivatives, computing and proving solutions of ODEs, propagating quantities across dynamics in which they do not change, automated application of invariant candidates, and conservation/symmetry arguments); and *3)* scriptable heuristic search automation.

[1] This chapter is based on *Bellerophon: Tactical Theorem Proving for Hybrid Systems* by Fulton et al. [65]

As Chapter 3 demonstrated, KeYmaera X [64] is structured from the very beginning to maintain a small and trustworthy core. Bellerophon is built upon this foundation. Using the logical foundations of dℒ [152], Bellerophon implements a set of automated deduction procedures. These procedures manifest themselves as a *library of hybrid systems primitives* in which complex hybrid systems can be interactively verified. Finally, heuristic automation tactics written in Bellerophon automatically apply these primitives to provide automation of hybrid systems reachability analysis.

Even though these ingredients can be found scattered across a multitude of theorem provers, their combination into a tactical theorem proving technique for hybrid systems is new. This combination enables more ambitious verification efforts and, as we will see in Part II, enables several novel approaches toward obtaining safety guarantees for reinforcement learning algorithms.

General purpose theorem provers, such as Coq [132] and Isabelle [141], have small foundational cores and tactic languages, but their tactic languages and automation are not tailored to the needs of hybrid systems. Even when these provers have formalizations of the classical theory of differential equations, those theories do not provide substantial automation and are not tightly incorporated into a holistic theorem proving environment. This chapter addresses the problem of getting from a strong mathematical foundation of hybrid systems [152] to a productive hybrid systems theorem proving tool. Reachability analysis tools, e.g. SpaceEx [59], provide automated hybrid systems verification for *linear* hybrid systems, but at the expense of a large trusted codebase and limited ways of helping when automation fails, which is inevitable due to the undecidability of the problem. KeYmaera's [154] user-defined rules are no adequate solution because they enlarge the trusted codebase and are difficult to get right.

Contributions. This chapter demonstrates how to combine a small foundational core [152], reusable automated deductions, and problem-specific proof-search tactics into a tactical theorem prover for hybrid systems. It presents Bellerophon, a hybrid systems tactics language and library implemented in the theorem prover KeYmaera X [64]. Bellerophon includes a tactics library which provides the decision procedures and heuristics necessary for a productive interactive hybrid systems proving environment. We first demonstrate the interactive verification benefits of Bellerophon through interactive verification of a simple hybrid system, which is designed to showcase a maximum of features in a minimal example. In the process, we also discuss significant components of the Bellerophon standard library that enable such tactical theorem proving. We then present two examples of proof search procedures implemented in Bellerophon, demonstrating Bellerophon's suitability for implementing reusable proof search heuristics for hybrid systems. Along the way, we demonstrate how the language features of Bellerophon support manual proofs and proof search automation.

4.2 The Bellerophon Tactic Language

Bellerophon is a programming language and standard library for automating proof constructions and proof search operations of the KeYmaera X core. As in other LCF-style provers [165], Bellerophon is not soundness-critical. This frees us to provide courageous reasoning strategies that enable users to perform high-level proofs about hybrid systems while still benefiting

from the high degree of trustworthiness that comes from a small soundness-critical core and the cross-verification of dℒ in Isabelle and Coq [25]. A basic use of Bellerophon is to recover a convenient sequent calculus for dℒ [147] from the simpler Hilbert calculus-based core [152] of KeYmaera X. This demonstrates that Bellerophon is expressive enough to implement the automation capabilities of the predecessor prover KeYmaera [154] from a smaller set of primitives. Beyond that, Bellerophon is used, e.g. for programming both individual proofs and custom proof search procedures.

This section presents the basic constructs of the Bellerophon language. Readers familiar with tactic languages for interactive theorem provers (e.g., [132]) will find many constructs familiar, but should pay particular attention to the discussion of Bellerophon's standard library. For usability, traceability and educational purposes, Bellerophon tactics can be written in a hierarchical structure that maps to the graphical tree structure of the resulting dℒ sequent proof [134].

The following dℒ formula Example 4 will be used as a running example throughout this chapter to demonstrate tactical theorem proving in Bellerophon. The example models a skydiver that may open his parachute or leave the chute closed. The model guarantees that the skydiver will land at a safe speed by ensuring that the parachute opens early enough.

Example 4 (Safety specification for the skydiver model)**.**

$$x \geq 0 \land g > 0 \land 0 < a = r < p \land -\sqrt{\frac{g}{p}} < v < 0 \land m < -\sqrt{\frac{g}{p}} \land T \geq 0 \qquad \textit{(init)}$$

$$\rightarrow [\{\ (?\underbrace{\left(r = a \land v - g \cdot T > -\sqrt{\frac{g}{p}}\right)}_{\textit{Dive}}\ \cup\ r := p);} \qquad \textit{(ctrl)}$$

$$t := 0;\ \{x' = v,\ v' = r \cdot v^2 - g\ \&\ x \geq 0 \land v < 0 \land t \leq T\} \qquad \textit{(plant)}$$

$$\}^*](x = 0 \rightarrow |v| \leq |m|) \qquad \textit{(post cond.)}$$

Opening the parachute is a discrete control decision. The diver's physics are modeled as an ODE, accounting for both gravity and drag, which changes when the parachute opens. This example is carefully crafted to demonstrate many of the challenges in hybrid systems reasoning while retaining relatively simple dynamics. Qualitative changes happen to the continuous dynamics after a discrete state transition, the dynamics are non-linear, and the property of interest is not directly inductive.

We model a gravitational force ($g > 0$), a drag coefficient (r) which depends on whether the parachute is closed (air a) vs. open (parachute p), the skydiver's altitude $x \geq 0$ and velocity $v < 0$. The time between control decisions is bounded by the skydiver's reaction time T. We also assume that the diver does not pass through the earth $x \geq 0$ and (to streamline this presentation) that $v < 0$.

The controller contains two options for our skydiver. The left choice lets a closed parachute ($r = a$) stay closed if the speed after one control cycle is definitely safe, computed by over-approximating as if gravity were the only force ($v - g \cdot T > -\sqrt{\frac{g}{p}}$). The right control choice opens the parachute, after which it stays open (as $r \neq a$). For simplicity, we say the parachute opens instantly.

The differential equations allow model the drag on the skydiver caused a combination of gravity and the parachute's drag.

Figure 4.1: Outline of a Sequent-Style Proof for Example 4.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Gamma \vdash \mathit{Dive} \to [\mathit{ode}](x{=}0 \to |v|{\leq}|m|)}}{\Gamma \vdash [\mathbf{?\mathit{Dive}}][\cdots](x{=}0 \to |v|{\leq}|m|)}\,\text{testb} \quad \dfrac{\dots}{\Gamma \vdash [\mathit{ode}(p)](x{=}0 \to |v|{\leq}|m|)}\,\text{assignb}}{\Gamma \vdash [?\mathit{Dive}][\cdots](x{=}0 \to |v|{\leq}|m|) \mathbf{\wedge} [r := p][\cdots](x{=}0 \to |v|{\leq}|m|)}\,\wedge\text{R}}{\Gamma \vdash [\{?\mathit{Dive} \mathbf{\cup} r := p\}][\{p' = v, v' = f(v,g,r)\}](x{=}0 \to |v|{\leq}|m|)}\,\text{choiceb}}{x \geq 0, \dots \vdash [\{?\mathit{Dive} \cup r := p\}\mathbf{;}\{x' = v, v' = f(v,g,r)\}](x{=}0 \to |v|{\leq}|m|)}\,\text{composeb}}{\vdash x \geq 0 \mathbf{\wedge} \dots \mathbf{\to} [\{?\mathit{Dive} \cup r := p\}; \{x' = v, v' = f(v,g,r)\}](x{=}0 \to |v|{\leq}|m|)}\,\text{prop}$$

The safety theorem says when the skydiver hits the ground, the velocity is at most a specified safe landing speed $|v|{\leq}|m|, v < 0$. We assume the parachute is initially closed ($r = a$), the speed initially safe ($v > -\sqrt{\frac{g}{p}}$), and the safe landing speed faster than the limit speed of the parachute ($m < -\sqrt{\frac{g}{p}}$).

The proof of the skydiver example motivates the constructs of our language and standard library. A sketch of the proof follows.

Proof 1 (Skydiver sequent proof sketch)**.** The proof starts from the initial conjecture (Example 4) at the bottom, phrased as a *sequent*. Each sequent has the shape *assumptions* $\vdash$ *obligations*, which means from the assumptions left of the turnstile $\vdash$, we have to prove any formula on the right. Horizontal lines indicate that the sequent below the horizontal line is proved when the sequent above the horizontal line is proved, justified by the tactic that is annotated left of the horizontal bar (the corresponding operator is highlighted in boldface and red). For example, the first step `prop` makes all conjuncts left of an implication available as assumptions, so the goal $x \geq 0 \wedge B \to C$ below the line becomes $x \geq 0, B \vdash C$ above the line. When proof rules (e.g., `andR`) result in multiple subgoals, each subgoal continues in a separate branch and all need to be proved.

Each step in the sequent proof above is a built-in tactic:

`prop` Exhaustively applies propositional proof rules in the sequent calculus.

`composeb` Splits sequential composition $[\alpha;\beta]P$ into nested modalities $[\alpha][\beta]P$.

`choiceb` Splits choice $[\alpha \cup \beta]P$ into a conjunction of subsystems $[\alpha]P \wedge [\beta]P$.

`andR`, `implyR`, `existsL`, `...` are the right conjunction rule ($\wedge$R), the right implication rule ($\to$R) and left existential rule ($\exists$L) as usual in sequent calculus. Throughout the chapter, we will make use of standard propositional sequent calculus tactics that follow this naming convention.

`testb` Makes test condition $[?Q]P$ available as assumption $Q \to P$.

`assignb` Makes effect of assignment $[x := t]P(x)$ available as update to $P(t)$ or as assumption $x = t$ with proper renaming of other occurrences of x.

Bellerophon programs, called tactics, are functions mapping lists of sequents to (lists of[2])

[2]Tactics may map a single sequent to a list of sequents; the simplest example of such a tactic `andR` corresponds to the proof rule $\wedge$R, which maps a single sequent $\Gamma \vdash A \wedge B, \Delta$ to the list of subgoals $\Gamma \vdash A, \Delta$ and $\Gamma \vdash B, \Delta$.

sequents. Built-in tactics (ranged over by τ) are implemented in Scala. Proof developers can combine existing tactics using the constructs described in Table 4.1. Built-in programs are implemented as a sequence of operations on a data structure that can only be created or modified by the soundness-critical core of KeYmaera X, thereby ensuring soundness of built-in tactics.

Table 4.1: Meaning of Bellerophon Tactic Combinators.

Language Primitive			Operational Meaning
e	::=	τ	Built-in tactic
	|	$e(\overline{v})$	Applies a tactic e to a (list of) positions or formulas
	|	e_1 `;` e_2	Sequential Composition: Applies e_2 on the output of e_1
	|	$e_1 \mid e_2$	Either Composition: Applies e_2 if applying e_1 fails
	|	e^*	Saturating Repetition: Repeatedly applies e as long as it is applicable (diverging if it stays applicable indefinitely)
	|	$?(e)$	Optional: Applies e if e does not result in an error
	|	$<(e_1, e_2, \ldots e_n)$	Applies e_1 to the first of n subgoals, e_2 to the second, etc.

Built-in Tactics. Bellerophon is both a stand-alone language and a domain-specific language embedded in the Scala programming language. Built-in tactics directly manipulate the KeYmaera X core to transform formulas in a validity-preserving manner. Bellerophon programmers can construct new tactics either by writing new built-in tactics in Scala, or else by combining pre-existing tactics using the combinators described in Table 4.1. KeYmaera X ships with a large library of tactics for proof construction and proof search. Some built-in tactics – the propositional rules and `choiceb` for example – are straight-forward applications of the axioms in [152]. Others provide a significant amount of automation on top of the axiomatic foundations. For example, `prop` combines propositional sequent calculus rules to an automated proof search procedure that often performs numerous simpler proof steps automatically.

Parameters. Most tactics are parameterized by formulas, locators, or both. Formula parameters are provided whenever the behavior of a tactic is dependent upon a particular formula; for example, the loop and differential induction tactics take an invariant formula as parameter. Locators specify where in a sequent a tactic should be applied. The simplest form of locator is an explicit position. Negative positions refer to formulas to the left of the turnstile ($\vdash$) and positive positions refer to formulas to the right of the turnstile,[3] e.g., $-1 : A,\ -2 : B,\ -3 : C \vdash 1 : D,\ 2 : E$ with annotated formula positions. In addition to explicit positions, Bellerophon provides indirect locators: *(i)* $e(\texttt{R})$ applies e to the first applicable position[4] in the succedent; *(ii)* $e(\texttt{Rlast})$ applies e to the last position in the succedent. $e(\texttt{L})$ and $e(\texttt{Llast})$ behave accordingly in the antecedent.

Basic Combinators. Tactics are executed sequentially using the ; combinator. In $e; f$, the left tactic e is executed on the current subgoal and then the right tactic f is executed on the result of the left tactic's execution. The | combinator attempts multiple tactics – moving from left to right through a list of alternatives. The $*$ combinator in e^* repeats the tactic e as long as it is applicable. Many proof search procedures are expressible as a repetition of choices.

[3]The addressing scheme extends to subformulas and subterms in a straight-forward way. Interested readers may refer to the Bellerophon documentation for details.

[4]Tactic e is applicable at a position pos if $e(pos)$ does not result in an error.

Branching. Proof search often results in branching. For example, a canonical proof of the induction step of Example 1 decomposes into two cases: a diving case corresponding to the control decision ?*Dive* and a deployed parachute case corresponding to the control decision $r := p$. Fig. 4.1 visually emphasizes the branching structure of this proof, which can be helpful for structuring tactics too. The $<$ combinator expresses how a proof decomposes into cases. An explicit tactic directly performing the derivation illustrated in Fig. 4.1 without any search is:

Listing 4.1: A Structured Bellerophon Tactic for a Branching Proof.

```
prop ; composeb(1) ; choiceb(1) ; andR(1) ; <(
  testb(1) ; ...,   /* tactic for left  branch of andR */
  assignb(1) ; ...  /* tactic for right branch of andR */
)
```

Equivalently, the proof search tactic `unfold` automates proofs such as Listing 4.1 by applying all propositional and dynamical axioms until encountering a loop program or a differential equation, where cleverness might be needed.

4.3 Formal Semantics

This section provides a formal semantic model for the intuitions conveyed in previous sections. This formalism is merely a model of the underlying implementation; in particular, the Bellerophon error reporting and handling mechanisms are much more sophisticated than the semantics defined here.

The semantics presented below contain an embedding into Scala, where built-in tactics (ranged over by β throughout) are interpreted. We also assume, implicitly, an evaluation context for Scala tactics that includes all of the tactics named in this thesis. This context mapping is implemented explicitly in the Bellerophon parser; see `DerivationInfo.scala` in the KeYmaera X source code[5].

4.3.1 Evaluation of Tactics

Recall from Chapter 3 that tactics operate over `Provable`s, which contain a `conclusion` sequent and a list of sequents called `subgoals` that imply the conclusion. Provables represent the proof state of a sequent derivation:

$$\frac{\Gamma_1 \vdash \Delta_1 \qquad \cdots \qquad \Gamma_n \vdash \Delta_n}{\Gamma_{\text{conclusion}} \vdash \Delta_{\text{conclusion}}}$$

An alternative representation for Provables is $(s; s_1, \ldots, s_n)$ where each s_i is a subgoal and s is the conclusion. Notice that there is a correspondence between each sequent s and the provable $(s; s)$ and also note that $(s; \text{nil})$ is the witness that s is proved. We denote by sugoals(p) the list of subgoals of a provable p.

[5] http://github.com/KeYmaeraX-release

Each tactic `t` maps a Provable to one or more new Provables. Tactics may also take arguments. For example, in the next section we will introduce a `loop` tactic that allows the user to specify a formula J that acts as a loop invariant as well as a position p indicated where in the sequent to loop program occurs. We refer to the type signature of the argument list that a tactic takes as the tactic's signature.

Definition 5 (Signature of a Bellerophon Tactic)**.** *The signature* $\mathsf{sig}(\texttt{e})$ *of a Bellerophon tactic* `e` *is generated by the grammar*

$$\tau ::= \mathit{String} \mid \mathit{Formula} \mid \mathit{Term} \mid \mathit{Position} \mid \tau, \ldots, \tau$$

where each Formula *is a* dℒ *formula, each* Term *is a* dℒ *term, and each* Position *is a sequence of dot-delimited integers indicating a position in a sequent or formula following the numbering scheme introduced in Chapter 3.*

For example, the signature of the `loop` tactic is $\mathsf{Formula}, \mathsf{Position}$; i.e.,

$$\mathsf{sig}(\texttt{loop}) = \mathsf{Formula}, \mathsf{Position}$$

The formula and position passed to `loop` are referred to as arguments or, collectively, the argument list.

The remainder of this section describes the evaluation semantics presented in Fig. 4.2, in which v ranges over argument lists, e over (unapplied or partially applied) tactics, and p over provables. When reading the formal semantics, it is important to pay attention to whether a tactic is being applied to an argument list or to a provable.

We denote by $e(p)$ the application of a tactic to a provable p, which may contain several subgoals. The i^th subgoal of p is denoted by $p[i]$ with $i \geq 1$ (the zeroth "subgoal" is the conclusion of the provable). Note that each subgoal is a sequent, not a provable. However, by reasonable abuse of notation, we can refer to each $p[i]$ as the `Provable` $(p[i]; p[i])$.

E-PRIMITIVE-ARG When constructing tactics, it is often useful to escape into the host language (Scala) to perform some computation, call an external tool (such as a computer algebra system), or manipulate formulas. The suitability of Scala for writing code that manipulates formulas is unsurprising due to the influence of ML on Scala's design [143] and the fact that ML was first developed as a language for developing tactics for the LCF system [76, 165]. To facilitate a blend of Scala programming and Bellerophon programming, we allow the definition of primitive tactics that are fully implemented in the Scala language. A primitive tactic name β is simply a string that is mapped (implicitly via reflection or explicitly via a key-value mapping) to a Scala class containing a method that maps proof states to proof states by performing operations in the KeYmaera X core. The premise of the rule states that applying the tactic associated with β to the argument list v evaluates (in Scala) to e'. The premise also contains an assumption that the signature of β in Bellerophon and the type of $[\![\beta]\!]_{\mathsf{Scala}}$ in Scala correspond. Under these conditions, $\beta(\beta(v)) \Downarrow e'$. We intentionally leave the exact implementation of the mapping between tactic names and provables loosely defined because the implementation details are not essential to the definition of the meaning of Bellerophon programs. Our design choices on this question have changed over time without any causing changes to other aspects of the Bellerophon evaluation semantics.

E-PRIMITIVE The E-PRIMITIVE rule is similar to the E-PRIMITIVE-ARG rule except for provables instead of for argument lists.

E-APP The E-APP rule allows a tactic to be reduced before it is applied to an argument list.

E-SEQ and E-EITHER the sequence and either rules implement the intuitive meaning of the sequential composition and either combinators; the sequential combinator composes the effect of two tactics sequentially, and the either tactic allows either the left or the right tactic to be applied. Note that the semantics implemented in KeYmaera X make the effect of the either combinator deterministic; i.e., the EITHER-RIGHT rule as implemented in KeYmaera X is:

$$\text{E-EITHER-RIGHT-KYX}\quad \frac{e_1(p) \Downarrow p' \qquad e_2(p) \Downarrow \mathsf{err}}{(e_1 \mid e_2) \Downarrow p'}$$

Note that in sequential implementation of this rule, $e_2(p)$ may remain unevaluated after first evaluating $e_1(p)$ to a non-error value p'. Alternatively, in a parallelized implementation, the evaluation of $e_2(p)$ may be terminated prematurely once the evaluation of $e_1(p)$ is complete.

Looping Tactics A tactic containing the repeat combinator can terminate in two ways. The first option is for the tactic to eventually reach an error (e.g., by becoming inapplicable to the current provable). The second option is for the tactic to reach a fixed point. These two alternatives are described by the E-*-ERR and E-*-FP rules respectively.

Tests Wrapping a tactic in the test combinator allows the user to specify that a possibly inapplicable tactic should be attempted. If the child tactic results in an error, then the proof state is unmodified and the tactic does not return an error state and the original proof state is propagated. However, if the child tactic does not result in an error, then the effect of the child tactic is propagated.

Branching Tactics Finally, the E-BRANCH-MAP allows a provable with several subgoals to be decomposed into several sub-provables to which tactics may be applied. Recall that by abuse of notation, each $p[1]$ in the premise is really the provable $(p[i]; p[i])$ where $p[i]$ is a sequent corresponding to the i^th subgoal. The meaning flatten(sugoals(p'_1) :: sugoals(p'_n) :: nil) in the conclusion of this rule is simply that each sugoals(p'_i) is a (possibly empty!) list of subgoals, so the list of lists (p'_1 :: p'_n :: nil) should be flattened into a single list. Thus, the final expression

$$(p; \text{flatten}(\text{sugoals}(p'_1) :: \text{sugoals}(p'_n) :: \text{nil}))$$

denotes a provable for which p is the conclusion and in which the subgoals are a (flattened) list of the subgoals resulting from the proof attempts on each branch.

Error Semantics Several of the above-mentioned rules refer to tactics evaluating to an error state. Error states are important in tactical theorem proving because many tactics are necessarily heuristic and may fail in unexpected ways. The E-PRIMITIVE propagates exceptions and throwables out of Scala and into Bellerophon. The exact implementation of this tactic is extremely important to usability, but is here left abstract. The sequential and either composition error propagation rules are straight-forward. Branching may fail in two ways. This first is a size mismatch between the arity of the tactic list passed to the branch combinator and the arity of the subgoal list of the provable to which the branching tactic is applied (E-BRANCH-SIZE-ERR). The second is an error occurring in a proof of one of the subgoals (E-BRANCH-SIZE-ERR).

E-PRIMITIVE-ARG
$$\frac{[\![\beta(v)]\!]_{\text{Scala}} \equiv e' \qquad \mathsf{sig}(\beta) = \text{type}_{\text{Scala}}(\beta)_a}{\beta(\beta(v)) \Downarrow e'}$$

E-PRIMITIVE
$$\frac{[\![\beta(p)]\!]_{\text{Scala}} \equiv p'}{\beta(\beta(p)) \Downarrow p'}$$

E-APP
$$\frac{e \Downarrow e'}{e(v) \Downarrow e'(v)}$$

E-SEQ
$$\frac{e_1(p) \Downarrow p' \qquad e_2(p') \Downarrow p''}{(e_1 \texttt{ ; } e_2)(p) \Downarrow p''}$$

E-EITHER-LEFT
$$\frac{e_1(p) \Downarrow p'}{(e_1 \mid e_2)(p) \Downarrow p'}$$

E-EITHER-RIGHT
$$\frac{e_2(p) \Downarrow p'}{(e_1 \mid e_2)(p) \Downarrow p'}$$

E-*-ERR
$$\frac{e(p_0) \Downarrow p_1 \qquad \cdots \qquad e(p_i) \Downarrow p_{i+1} \qquad e(p_{i+1}) \Downarrow \mathsf{err}}{e^*(p_0) \Downarrow p_{i+1}}$$

E-*-FP
$$\frac{e(p_0) \Downarrow p_1 \qquad \cdots \qquad e(p_i) \Downarrow p_{i+1} \qquad e(p_{i+1}) \Downarrow p_{i+1}}{e^*(p_0) \Downarrow p_{i+1}}$$

E-TEST-ERR
$$\frac{e(p) \Downarrow \mathsf{err}}{?e(p) \Downarrow p}$$

E-TEST
$$\frac{e(p) \Downarrow p'}{?e(p) \Downarrow p'}$$

E-BRANCH-MAP
$$\frac{\mathsf{length}(\mathsf{subgoals}(p)) = n \qquad e_1(p[1]) \Downarrow p'_1 \qquad \ldots \qquad e_n(p[n]) \Downarrow p'_n}{<(e_1, \ldots, e_n)(p) \Downarrow (p; \mathsf{flatten}(\mathsf{sugoals}(p'_1) :: \mathsf{sugoals}(p'_n) :: \mathsf{nil}))}$$

Figure 4.2: The evaluation dynamics of Bellerophon.

[a] $[\![\cdot]\!]_{\text{Scala}}$ identifies and executes the tactic named β in the KeYmaera X built-in tactic library, then executes that tactic on the object obtained by running the parser on v.

$\boxed{e \Downarrow \mathsf{err}}$

$$\frac{[\![\beta(v)]\!]_{\mathsf{Scala}} \equiv \mathsf{err}}{\beta(v) \Downarrow \mathsf{err}}\ \text{E-PRIMITIVE-ARG} \qquad \frac{[\![\beta(p)]\!]_{\mathsf{Scala}} \equiv \mathsf{err}}{\beta(p) \Downarrow \mathsf{err}}\ \text{E-PRIMITIVE} \qquad \frac{e \Downarrow \mathsf{err}}{e(v) \Downarrow \mathsf{err}}\ \text{E-APP} \qquad \frac{e_1(p) \Downarrow \mathsf{err}}{(e_1\ ;\ e_2)(p) \Downarrow \mathsf{err}}\ \text{E-SEQ-ERRLEFT}$$

$$\frac{e_1(p) \Downarrow p' \qquad e_2(p') \Downarrow \mathsf{err}}{(e_1\ ;\ e_2)(p) \Downarrow \mathsf{err}}\ \text{E-SEQ-ERRRIGHT} \qquad \frac{e_1(p) \Downarrow \mathsf{err} \qquad e_2(p) \Downarrow \mathsf{err}}{(e_1 \mid e_2)(p) \Downarrow \mathsf{err}}\ \text{E-EITHER-ERR} \qquad \frac{\text{length}(\text{subgoals}(p)) \neq n}{<(e_1, \ldots, e_n)(p) \Downarrow \mathsf{err}}\ \text{E-BRANCH-SIZE-ERR}$$

$$\frac{\exists i.1 \leq i \leq n \wedge e_i(p[i]) \Downarrow \mathsf{err}}{<(e_1, \ldots, e_n)(p) \Downarrow \mathsf{err}}\ \text{E-BRANCH-EVAL-ERR}$$

Figure 4.3: Error Propagation Semantics of Bellerophon (includes all evaluation rules of the form $e \Downarrow \mathsf{err}$ that do not introduce new errors).

4.4 Demonstration of Tactical Hybrid Systems Proving

In this section, we demonstrate that the Bellerophon standard library's techniques for invariance properties, conservation properties, and real arithmetic simplifications, as implemented in KeYmaera X, make it a convenient mechanism for interactively verifying hybrid systems. We do so by considering the proof of Example 4 in detail.

Loop Invariants

Verifying a system loop begins with identifying a loop invariant J that is true initially, implies the post-condition and is preserved by the controller. Each formula of the initial condition in the parachute model is invariant except $r = a$; therefore, we will proceed with the following invariant J:

$$\underbrace{(x \geq 0 \wedge v < 0)}_{\textit{ev.dom.}} \wedge \underbrace{\left(g > 0 \wedge 0 < a < p \wedge T \geq 0 \wedge m < -\sqrt{\frac{g}{p}}\right)}_{\textit{diff. inductive}} \wedge \underbrace{v > -\sqrt{\frac{g}{p}}}_{\textit{hard}} \tag{4.1}$$

Note that J holds initially and implies formula $|v| \leq |m|$ because $v > -\sqrt{\frac{g}{p}} > m$. These facts prove automatically. Therefore, the core proof needs to prove $J \rightarrow [\textit{ctrl}; \textit{plant}]J$. We express the proof thus far with the following tactic:

Loop Induction Tactic.

```
1 implyR(1); loop(J, 1); <(QE,QE,nil)
```

The `implyR` tactic corresponds to the right implication rule ($\rightarrow$R) in sequent calculus; the first argument states that we should apply this proof rule at the first position in the succedent. The

`loop` tactic uses the dℒ axioms about loops to derive three new subgoals: (1) the loop invariant holds initially (*init* $\rightarrow J$); (2) the loop invariant implies the post condition ($J \rightarrow$ *post cond.*); and (3) the loop invariant is preserved throughout a single iteration of the loop ($J \rightarrow [\textit{ctrl};\textit{plant}]J$). The `loop` rule in KeYmaera X is derived in Bellerophon from axioms and automatically retains assumptions about constants that do not change in the system. The `nil` tactic has no effect and is used in `<()` to keep subgoal (3) unchanged.

The branching combinator `<()` allows us to isolate each of these three subtasks from one another. Subgoals (1) and (2) are proven using a Real Arithmetic solver (`QE`, for `Q`uantifier `E`limination), since the arithmetic is easy enough here.

Decomposing Control Programs

The skydiver model's control program is simple. It checks if it is safe to keep the parachute closed, or sets r to open the parachute (at any time, but at the latest when it is no longer safe to keep it closed). Therefore, we will immediately symbolically execute the control program and consider the two resulting subgoals, both of which are reachability conditions on purely continuous dynamical systems. This splitting could be done manually, as in Listing 4.1. But we decide to split it automatically using the `unfold` tactic.

Decomposing Control Programs.

```
1 implyR(1); loop(J, 1); <(QE,QE,unfold)
```

ODE Tactics in the Standard Library

The rest of the proof will make use of several tactics in the Bellerophon standard library:

`boxAnd` Splits $[\alpha](P \wedge Q)$ into separate postconditions $[\alpha]P$ and $[\alpha]Q$.

`dC(R)` Proves a new property R of an ODE and then restricts the differential equation to remain within the evolution domain R (differential cut).

`dW` Proves $[x' = f(x)\&Q]P$ by proving that domain Q implies postcondition P.

`dI` Proves $[\{x' = f(x)\}]P$ by proving P and its differential P' along $x' = f(x)$.

`dG(y'=ay+b,R)` Adds new differential equation $y' = ay + b$ to $[x' = f(x)\&Q]P$, and replaces the post condition by equivalent formula R (possibly mentioning the fresh *differential ghost variable* y).

These tactics perform significant automation on top of the dℒ axioms. For example, `dI` performs automatic differentiation via exhaustive left-to-right rewriting of our axiomatization of differentials (e.g., $(s \cdot t)' = s't + st'$) and propagates the local effect of the differential equation. The `dI` tactic preserves initial value constraints for variables that are not changed by the differential equation. It often performs hundreds of axiom applications automatically. The difference between the sound Differential Induction axiom [152] and the automation provided by the `dI` tactic is an exemplary demonstration of the difference between a theoretically complete mathematical/logical foundation, and a pragmatically useful tactical library.

We are now ready to consider two purely continuous subgoals of the form $J \rightarrow [\textit{plant}(r)]J$: one where $r = a$ (the parachute is closed) and one where $r = p$ (the parachute is open), which are both valid for different reasons.

Closed Parachute: Chaining Inequalities

We first consider the $r = a$ case, in which the parachute is closed. Symbolically executing the control program results in a remaining subgoal that requires us to prove:

$$J \wedge v - g \cdot T > -\sqrt{g/p} \rightarrow [\{x' = v,\ v' = a \cdot v^2 - g \& x \geq 0 \wedge v < 0 \wedge t \leq T\}]J$$

We use `boxAnd` to work on the conjuncts of the loop invariant J separately, since each are preserved for different reasons. The proofs for the first two sets of loop invariants in J (labeled *ev. domain* and *diff. induction*) are identical to the $r = p$ case and will be discussed later. Here, we focus on the formula $J \wedge v - g \cdot T > -\sqrt{\frac{g}{p}} \rightarrow [\{x' = v, v' = a \cdot v^2 - g \& x \geq 0 \wedge v < 0 \wedge t \leq T\}]v > -\sqrt{\frac{g}{p}}$, which handles the third conjunct of J (labeled *hard*).

Compute that $v \geq v_0 - g \cdot t \geq v_0 - g \cdot T > -\sqrt{\frac{g}{p}}$, where v_0 is the value of v before the ODE. In Bellerophon proofs for differential equations, we use $old(v)$ to refer to the initial value of a variable that will change throughout a loop or differential equation. For example, when executing $\{x := x - 1\}^*$ from a state where $x = 5$, the value of `old(x)` is always 5 whereas the value of x will change on each iteration of the loop.

Each of the subformulas in the postcondition above is a differentially inductive invariant, or else is valid after the domain constraint is automatically augmented with constants $g > 0 \wedge p > 0$. Therefore, we use a chain of `dC` justified either by `dI` or by `dW` for each inequality in this tactic:

A Chain of Inductive Inequalities.

```
1 /* Key lemmas                          proofs of lemmas */
2 dC(v>=old(v)-g()*t,1);              <(nil , dI(1));
3 dC(old(v)-g()*t>=old(v)-g()*T,1); <(nil , dW(1);QE);
4 dC(old(v)-g()*T>-c,1);              <(nil , dI(1));
5 dW(1) ; QE
```

The argument is a sequence of differential cuts, each of which has a simple proof, and whose conjunction implies the post-condition. Each of the `nil` tactics in the `<()` passes along a single subgoal to the next tactic, so that at the end we have a long conjunction in our domain constraint containing each of the cuts. This style of proof is pervasive in hybrid systems verification, and easily expressed in Bellerophon. One key feature that makes this proof so concise is the use of `old(v)`, which introduces a variable v_0 that remembers the initial value of v. Tactic `dW(1)`; QE on line 3 proves the cut from the evolution domain constraint.[6]

The inequalities in the evolution domain of the differential equation system are now sufficiently strong to guarantee the postcondition, so we use `dW` to obtain a final arithmetic subgoal: $\Gamma \vdash v \geq v_0 - g \cdot t \geq v_0 - g \cdot T > -\sqrt{\frac{g}{p}} \rightarrow v > -\sqrt{\frac{g}{p}}$, where Γ contains constants propagated by the rule `dW` (unlike the DW axiom).

Although this arithmetic fact is obvious to us, `QE` will take a substantial amount of time to prove this property (at least 15 minutes on a 32 core machine running version 10 Mathematica

[6]The attentive reader will notice we use `g()` instead of g. This is to indicate that the model has an arity 0 function symbol `g()`, rather than an assignable variable whose value may change from state to state.

and version 4.3.7 of KeYmaera X). This is a fundamental limitation of Real Arithmetic decision procedures, which have extremely high theoretical and practical complexity [41].

The simplest way to help `QE` is to introduce a simpler formula that captures the essential arithmetic argument: e.g., cut in $\forall a, b, c, d\,(a \geq b \geq c > d \rightarrow a > d)$ and then instantiate this formula with our chain of inequalities. We take this approach for demonstration (see the implementation). As an alternative, transforming and abbreviating formulas in Bellerophon achieves a similar effect.

Open Parachute: Differential Ghosts

We now consider case 2, where the parachute is already open ($r = p$). After executing the discrete program the remaining subgoal is:

$$J \rightarrow [\{x' = v,\ v' = p \cdot v^2 - g \ \&\ \underbrace{x \geq 0 \wedge v < 0 \wedge t \leq T}_{\text{evolution domain constraint}}\}]J$$

The proof proceeds by decomposing the post-condition J into three separate subgoals, one for each conjunct in J. In Listing 4.5, the `boxAnd` tactic uses axiom $[\alpha](P \wedge Q) \leftrightarrow [\alpha]P \wedge [\alpha]Q$ from left to right, to rewrite the instance of $[\alpha](P \wedge Q)$ to separate corresponding conjuncts $[\alpha]P \wedge [\alpha]Q$. The first set of formulas in J (labeled *ev. domain*) are *not* differentially inductive, but are trivially invariant because the evolution domain constraint of the system already contains these properties. Differential weakening by `dW` is the appropriate proof technique for these formulas, see line 1 in Listing 4.5. The second set of formulas (labeled *diff. inductive*) are *not* implied by the domain constraint, but are inductive along the ODE because the left and right sides of each inequality have the same time-derivative (0). Differential induction by `dI` is the appropriate proof technique for establishing the invariance of these formulas, see line 2 in Listing 4.5.

Listing 4.5: Differential Weakening and Differential Induction.

```
1 boxAnd(1); andR(1); <(dW(1);QE , nil);
2 boxAnd(1); andR(1); <(dI(1) , nil)
```

The third conjunct (labeled *hard*) requires serious effort: we have to show that $v > -\sqrt{\frac{g}{p}}$ is an invariant of the differential equation. This formula is *not* a differentially inductive invariant because it is getting less true over time. To become inductive, we require additional dynamics to describe energy conservation. The Bellerophon library provides a tactic to introduce additional dynamics as *differential ghosts* into a differential equation system. Often, differential ghosts can be constructed systematically. Here, we want to show $v > -c$ where $c = \sqrt{\frac{g}{p}}$, so we need a property with a fresh differential ghost y that entails $v + c > 0$, e.g., $y^2(v + c) = 1$. The formula $y^2(v + c) = 1$ becomes inductively invariant when $y' = -\frac{1}{2}p(v - c)$. In summary, tactic `dG` in Listing 4.6 introduces $y' = -\frac{1}{2}p(v - c)$ into the system and rewrites the post-condition to $y^2(v + c) = 1$ with the additional assumptions that y does not contain any singularities ($p > 0 \wedge g > 0$).

Listing 4.6: Finishing the Parachute Open Case with a Ghost.

```
1 dG(y'=-1/2*p*(v-(g()/p)^(1/2)), p>0&g()>0&y^2*(v+c)=1, 1) ;
```

```
2 dI(1.0); QE
```

Tactic `dG` results in a goal of the form $\exists y[\cdots](p > 0 \land g > 0 \land y^2(v + c) = 1)$, so in line 2 of Listing 4.6 we apply `dI` at the first child position 1.0 of succedent 1 *in context* of the existential quantifier to show that the new property $y^2(v + c) = 1$ is differentially invariant with the differential ghost y.

If a system avoids possible singularities, the `ODE` tactic in the Bellerophon standard library automatically computes the differential ghost dynamics (here $y' = -\frac{1}{2}p(v-c)$) and postcondition (here $y^2(v+c) = 1$) with the resulting proof. Additionally, notice that `dG` conveniently constructs the axiom instance of DG [152], saving the proof developer from manually constructing such instances.

The proof in Listing 4.6 completes the invariant preservation proof for $r = p$. The full proof artifact for the skydiver demonstrates how Bellerophon addresses each of the major reasoning challenges in a typical hybrid systems verification effort.

4.5 The Bellerophon Standard Library

The Bellerophon standard library contains two major components: an implementation of various easy-to-use proof calculi for dL, and a suite of general-purpose tactics that encode general results about dynamical systems.

4.5.1 Proof Calculi

The simplest component of the Bellerophon standard library is an implementation of the dL sequent calculus in terms of the dL Hilbert axiom system. Most of this implementation is a straightforward translation of axioms into sequent-style proof rules. For example, the proof rule for box assignment:

$$\frac{\Gamma \vdash \varphi_{x\to\theta}, \Delta}{\Gamma \vdash [x := \theta]\varphi, \Delta}$$

where $\varphi_{x\to\theta}$ is short-hand for the replacement of all free occurrences of x by θ in φ, is implemented by:

1. cutting the assignment axiom `[x_:=f();]p(x_)<-> p(f())` into the proof,
2. proving the branch that contains the cut-in axiom in the succedent by hiding all formulas except the axiom and making a direct appeal to that axiom.
3. performing a uniform substitution on this axiom so that $x\_$ is substituted by x, $f()$ by θ, and $p(x)$ and $p(f())$ by φ and $\varphi_{x\to\theta}$,
4. performing equivalence rewriting on the formula $[x := \theta]\varphi$ using the newly instantiated equivalence, and finally
5. hiding the now-applied equivalence.

This general pattern – transforming a formula using an axiomatic equivalence or implication – is particularly common in the Bellerophon implementation of the dL sequence calculus. For

this reason, the Bellerophon implementation includes a specialized `useAt` tactic will perform this style of reasoning automatically.

Although most of the dL sequent calculus more-or-less follows from this pattern of reasoning, the implementation of the solve rule is not so straightforward.

4.5.2 Solving Differential Equations

Many systems of differential equations have explicit, closed form solutions residing in decidable fragments of Real arithmetic. When these solutions exist, they are powerful tools for reachability analysis.

For example, the purely continuous system $x'(t) = v(t), v'(t) = a(t)$ where $t' = 1$ is often a useful model of fragments of hybrid systems. The solution to this system is $x(t) = a_0\frac{t^2}{2} + v_0t + x_0, v(t) = a_0t + v_0, a(t) = a_0$ where x_0, v_0, a_0 are symbolic initial values for the system. For hybrid systems, constraints on these variables are determined by the discrete fragment of the system and any initial conditions on the reachability theorem under consideration.

The following axiom schema can be used to verify systems with explicit closed-form solutions that reside within a decidable fragment of arithmetic, where both x and f are vectors, and $y(t)$ is the solution to the differential equations for x evaluated at time t:

$$[x' = f]\phi \leftrightarrow \forall t \geq 0[x := y(t)]\phi$$

We choose to disclude rules and axioms such as the one above from the core of KeYmaera X because of the complex side condition that $y'(t) = f$, which is difficult and subtle to check. Such side-conditions interact in subtle ways with the rest of the logic, and should be avoided when possible. Instead of introducing a new and fragile mechanism to check this one side condition, KeYmaera X leverages Bellerophon tactics to construct an untrusted solver using the more general axiomatization of Lie Derivatives already present in the prover's core.

An Axiomatic ODE Solver. The intuition behind our tactic for an axiomatic ODE solver is simple. For linear homogeneous systems, all partial solutions of the system are inductive invariants. We use the Differential Cut (DC) rule to cut in each partial solution, and then prove that the solution is an invariant using Differential Induction (DI). We then rewrite the post-condition P in terms of time alone, and remove all differential equations except $t' = 1$ from the system using the Differential Ghost (DG) axiom in the right-to-left direction. Finally, we use the Differential Solve (DS) axiom when we have only $t' = 1$ left in the ODE.

A formal derivation for an example system $x' = v, v' = a$ is given in Fig. 4.4. Some of these steps require auxiliary goals; a full discussion of this derivation is presented [152].

A formal derivation corresponding to the proof constructed by the axiomatic solver for the system $x' = v, v' = a$ follows. Some of these steps require auxiliary goals; a full discussion of this derivation is presented in [152], which introduced this proof technique.

The corresponding top-level Bellerophon tactic decomposes this proof into a number of steps:

The `addTime` tactic introduces a time variable to an autonomous system of equations using the DG axiom (corresponding to the bottom two steps in the sequent derivation).

The `cutInSoln` tactic is a built-in tactic that computes partial solutions to the ODE and cuts in these solutions following the variable dependency ordering. For example, the first partial

$$\phi \vdash \forall s(x_0 + \frac{a}{2}s^2 + v_0 s \geq 0)$$

$$\phi \vdash \forall t \geq 0 \forall 0 \leq s \leq t [t := 0 + 1s]x_0 + \frac{a}{2}t^2 + v_0 t \geq 0 \quad \text{(}^{\text{DS}}\text{)}$$

$$\phi \vdash [t' = 1]x_0 + \frac{a}{2}t^2 + v_0 t \geq 0 \quad \text{(}^{\text{DG}}\text{)}$$

$$\phi \vdash [v' = a, t' = 1]x_0 + \frac{a}{2}t^2 + v_0 t \geq 0 \quad \text{(}^{\text{DG}}\text{)}$$

$$\phi \vdash [x' = v, v' = a, t' = 1]x_0 + \frac{a}{2}t^2 + v_0 t \geq 0 \quad \text{(}^{\text{DC}}\text{)}$$

$$\phi \vdash [x' = v, v' = a, t' = 1 \& v = v_0 + at]x_0 + \frac{a}{2}t^2 + v_0 t \geq 0 \quad \text{(}^{\text{DC}}\text{)}$$

$$\phi \vdash [x' = v, v' = a, t' = 1 \& v = v_0 + at \wedge x = x_0 + \frac{a}{2}t^2 + v_0 t]x_0 + \frac{a}{2}t^2 + v_0 t \geq 0 \quad \text{(}^{\text{G}}\text{)}$$

$$\phi \vdash ([x' = v, v' = a, t' = 1 \& v = v_0 + at \wedge x = x_0 + \frac{a}{2}t^2 + v_0 t]x = x_0 + \frac{a}{2}t^2 + v_0 t) \rightarrow x \geq 0 \quad \text{(}^{\text{DW}}\text{)}$$

$$\phi \vdash [x' = v, v' = a, t' = 1 \& v = v_0 + at \wedge x = x_0 + \frac{a}{2}t^2 + v_0 t]x \geq 0 \quad \text{(}^{\text{DC}}\text{)}$$

$$\phi \vdash [x' = v, v' = a, t' = 1 \& v = v_0 + at]x \geq 0 \quad \text{(}^{\text{DC}}\text{)}$$

$$\phi \vdash [x' = v, v' = a, t' = 1]x \geq 0 \quad \text{(}^{\text{prop}}\text{)}$$

$$\phi \vdash \exists t [x' = v, v' = a, t' = 1]x \geq 0 \quad \text{(}^{\text{DG}}\text{)}$$

$$\phi \vdash [x' = v, v' = a]x \geq 0$$

Figure 4.4: An Axiomatic Proof Using Solutions

Listing 4.7: Top-Level Axiomatic Solve Tactic.

```
1 solve(pos) := addTime(pos);
2               cutInSoln(pos)*;
3               simplifyPostCondition(pos);
4               inverseDiffCut(pos)*;
5               inverseDiffGhost(pos)*;
6               useAt("DS")(pos)
```

solution for the example in Fig. 4.4 is $v = v_0 + at$. The $\star$ operator repeats this operation until it is no longer applicable.

The purpose of lines 3–5 of `solve` is to remove all occurrences of primed variables (except t) from both the evolution domain constraint and the post-condition of the formula. Following the $x' = v, v' = a$ example, our goal is to produce a formula $[x' = v, v' = a, t' = 1 \& Q]P$ in which Q and P do not mention x or v, but may mention a or t. These steps are necessary so that the DG axiom can be used to remove $x' = v, v' = a$ from the system; doing so makes the univariate DS axiom applicable. This axiom is the only one which characterizes the notion of a solution, and is only applicable to the equation $t' = 1$; applying the DS axiom to larger systems of differential equations would (rightly) result in a substitution clash. It is in fact important that such a clash should occur because allowing the use of the DS axiom for arbitrary systems of differential equations would certainly be unsound.

The `simplifyPostCondition` tactic rewrites the post-condition so that it does not mention any primed variables in the autonomous system. This step ensures that t is the only variable bound by the system c, allowing all other variables to be removed from the system via use of the DG axiom in right-to-left order.

The `inverseDiffCut` tactic rewrites $[c \& H \wedge s]P$ to $[c \& H]P$ in the evolution domain constraint where c is the system of differential equations. This step removes all mentions of one of the ODE's primed variables in the evolution domain constraint. This step incurs significant overhead because removing each component of the domain constraint requires at least one call to a real arithmetic solver. An alternative, more efficient, way of performing these proof steps is to use the differential refine axiom.

After the `inverseDiffCut` steps, the domain constraint and post-condition are both stated in terms of the time variable and do not mention any other variables that occur primed in the ODE. The `diffGhost` tactic is now clear to remove all equations of the system of differential equations except $t' = 1$. Each of the tactics discussed so far are themselves built-in tactics that construct Bellerophon expressions.

Finally, the DS axiom is used to solve the now univariate system, producing a final arithmetic subgoal corresponding to the solution to the system of differential equations.

The tactic presented here is a slight simplification of the full top-level tactic implemented in KeYmaera X. The KeYmaera X tactic performs a significant amount of preprocessing to make the base tactic applicable in a wider range of settings. If initial values for primed variables (e.g. x_0, v_0) are not included in the initial antecedent, then symbolic initial conditions are added by discrete ghosts. Differential equations are commuted so that variables occur in dependency ordering (i.e., $x' = v, v' = a$ and $v' = a, x' = v$ are both supported by commuting across the comma). Real arithmetic subgoals are optimized. The tactic also includes a custom integrator that handles a subset of linear homogeneous differential equations. KeYmaera X only supports automatic solving of systems whose solutions exist in a decidable fragment of real arithmetic because the ODE solver tactic makes repeated calls to an arithmetic decision procedure.

The solver supports both safety properties ($[x' = v, v' = a]P$) as well as liveness properties (e.g., $\langle x' = v, v' = a \rangle P$). The diamond component of the ODE solver is a critical component of the ModelPlex algorithm [135]. The full implementation in KeYmaera X is roughly 1000 lines of Scala (excluding substantial supporting code that is reused in other parts of the codebase; e.g., the implementation of useAt, the core Bellerophon data structures and interpreter, etc.). These 1000

lines construct Bellerophon programs using a Scala domain-specific language whose semantics are equivalent to those presented in this chapter.

4.5.3 Reasoning about Bifurcations

Despite the relative simplicity and small size of the KeYmaera X core, Bellerophon and KeYmaera X are capable of expressing many properties about continuous dynamical systems.

One significant tactic is the axiomatic ODE solver, which uses the axioms and proof rules of dL to prove the existence of solutions to a subset of linear ordinary differential equations. The ODE solver implemented by Fulton and others provides validated solutions to differential equations and is one of the largest tactics in the Bellerophon standard library. The ModelPlex algorithm [135], which generates runtime monitors for hybrid systems, is implemented as a Bellerophon tactic and makes essential use the axiomatic ODE solver.

KeYmaera X and dL are designed for hybrid program verification, not general-purpose mathematics. Therefore, the variety of results about dynamical systems that are encodable using a combination of dL and Bellerophon tactics is often surprising. The Axiomatic ODE Solver is a great example of this surprising expressiveness. The existence of closed-form solutions for linear systems is not directly expressible in dL. However, a schema of dL formulas describing all such systems combined with a Bellerophon tactic that can prove the invariance of a solution for any system in this schema produces the same result. The dL formulaic schema is a theorem about a general class of dynamical systems, and the Bellerophon tactic generates a proof for each formula in the schema.

Bifurcation theory provides another nice example of the often surprising expressiveness of KeYmaera X and Bellerophon. A bifurcation point is a point at which a dynamical system's qualitative dynamics changes [24, 157]. From a theorem proving perspective, bifurcation points are interesting because the proof of a property about an ODE is often different on either side of a bifurcation. Perhaps the simplest example of a bifurcation point arises in the following system:

$$x' = r + x^2$$

The existence of a bifurcation point in this 1D saddle-node system [118] – and the location of the fixed points of the system on either side of this bifurcation – can be encoded in dL and Bellerophon. The following dL formula encodes the fact that some fixed point f exists for any choice of a parameter $r \leq 0$: $r \leq 0 \rightarrow \exists f(x = f \rightarrow [x' = r + x^2]x = f)$

The proof of this property must identify the bifurcation point at $r = 0$ and discover the corresponding fixed points ($f = -\sqrt{-r}$ when $r < 0$ and $f = 0$ when $r = 0$):

The Equilibrium Points of the 1D Saddle-Node Bifurcation[7].

```
/* Move r <= 0 to the antecedent */
implyR(1);
/* Introduce and prove r = 0 ∨ r < 0 so that we can split the
 * rest of our analysis along this bifurcation. */
cut(r = 0 ∨ r < 0); <(hideL(-1), hideR(1) ; QE);
/* Split the proof. */
orL(-1); <(
  /* CASE 1: r=0 */
```

```
  existsR({0}, 1);                          /* choose f=0 */
  implyR(1);
  dG({y' = -xy}, yx = 0 ∧ y > 0, 1); /* x = 0 ↔ ∃y(yx = 0 ∧ y > 0) */
  existsR({1}, 1);                          /* choose y ≠ 0; e.g., y = 1 */
  /* Consider y*x=0 and y>0 differential invariants separately */
  boxAnd(1); andR(1); <(
    /* y*x=0 is differentially inductive because:
       (yx)' = 0 ↔ y'x + x'y = 0 and y'x + x'y = 0 ↔ -xy^2 + r + x^2 = 0
       (recall: r=0)
    */
    dI(1)
    ,
    /* y>0 case needs an extra cut to use differential ghost for open set */
    dG(z'=x/(2z), z^2y = 1, 1);
    existsR({1}, 1);
    dI(1)
  )
  ,
  /* CASE 2: r < 0 */
  /* introduce new variable s = sqrt(-r) and prove s exists. */
  cut(∃s.r = -s^2); <(nil, hideR(1) ; QE);
  /* Some cleanup work about s */
  existsL(-2) ; existsR(-s, 1) ; implyR(1) ;
  dG({y' = (s - x)y}, y(x + s) = 0 ∧ y > 0, 1) ; existsR({1}, 1) ;
  boxAnd(1) ; andR(1); <(
    dI(1),
    dG({z' = (x-s)/(2z)}, z^2y = 1, 1) ; existsR({1}, 1) ; dI(1)
  )
)
```

The main idea is that many results about dynamical systems which are not traditionally thought of as reachability properties are none-the-less expressible using a combination of dℒ and Bellerophon even though KeYmaera X is not explicitly designed to support general-purpose mathematics. I.e., a *combination* of Bellerophon tactics and schematic dℒ formulas can express general results about dynamical systems. This fact enables our proof-producing axiomatic ODE solver without any direct appeal to Picard–Lindelöf. With a bit of ingenuity, KeYmaera X's approach toward hybrid systems analysis is surprisingly powerful.

4.5.4 Tactical Automation for ODEs

Automated reasoning for ODEs is critical to scalable analysis of hybrid systems. Even when human interaction is required, automation for simple reachability problems – such as reachability for solvable or univariate subsystems – streamlines analysis and reduces requisite human effort.

The skydiver example above illustrated the interplay between finding differential invariants and proving with differential induction and differential ghosts. The tactic `ODE` in the Bellerophon standard library automates this interplay for solvable systems and some unsolvable, nonlinear

[7] Disambiguating `{ · }` markers are replaced with inline formulas; e.g., $r = 0 \vee r < 0$ is actually written as `{r=0|r<0}` in the Bellerophon script.

systems of differential equations, see Listing 4.9. The `ODEStep` tactic directly proves properties by differential induction, with differential ghosts, and from the evolution domain constraints. The `ODEInvSearch` tactic cuts additional differential invariants, thereby strengthening the evolution domain constraints for `ODEStep` to ultimately succeed. Tactic `ODE` succeeds when `ODEStep` finds a proof; if `ODEStep` does not yet succeed, `ODEInvSearch` provides additional invariant candidates with differential cuts `dC` or by solving the ODE. This interaction between `ODEStep` and `ODEInvSearch` is implemented in Listing 4.9 by mixing recursion and repetition. Repetition is used in `ODE` so that `ODEStep` is prioritized over `ODEInvSearch` each time that a new invariant is added to the system. Recursion is used in `ODEInvSearch` so that a full proof search is started every time an invariant is successfully added to the domain constraint by `dC`. The `ODEInvSearch` tactic calls `ODEStep` on its second subgoal (the "show" branch of the `dC`) because differential cuts can be established in the right order without additional cuts.

Listing 4.9: Automated ODE Tactic for Non-Solvable Differential Equations.

```
1 ODEStep(pos)       = dI(pos) | dgAuto(pos) | dW(pos) | ...
2 ODEInvSearch(pos) = dC(nextCandidate); <(ODE(pos), ODEStep(pos))
3                       | solve(pos)
4 ODE(pos)           = ( ODEStep(pos) | ODEInvSearch(pos) )*
```

The `ODEStep` tactic finds a proof with `dI` when the post-condition is differentially inductive, meaning that the vector field of the differential equation points into the set described by the differential equation. The `dgAuto` tactic will also attempt to make properties differentially inductive by constructing differential ghosts for the postcondition, such as the ghosts introduced in the skydiver example. In case the evolution domain of a differential equation system is sufficiently strong, tactic `dW` succeeds from just the evolution domain constraints. The `ODEStep` tactic implemented in KeYmaera X contains other proof search techniques (marked ... above) that are guaranteed to terminate but refrain from performing differential cuts.

The invariant search `ODEInvSearch` constructs candidates for differential invariants heuristically [153], see `dC(nextCandidate)` in Listing 4.9, or systematically for solvable differential equations with `solve`. Tactic `solve` follows an axiomatic ODE solver approach [152] that implements a solver in terms of the differential invariants, cuts, and ghosts reasoning principles to avoid a trusted built-in rule for solving differential equations (such trusted built-in rules are necessary in other hybrid systems tools, e.g., in KeYmaera [154]).

The `ODE` tactic described above is a simplified version of the ODE tactic implemented in KeYmaera X, which contains additional automated search procedures and specializes proof search based upon the shape of the post-condition.

4.5.5 Tactical Automation for Hybrid Systems

The `solve` and `ODE` tactics provide some automation for continuous systems proofs. The `master` tactic builds on these to provide a full heuristic for hybrid systems in the canonical form $init \rightarrow [\{ctrl;\ plant\}^*]safe$. Tactic `master` combines the three basic reasoning principles that together cover the reasoning tasks arising in hybrid systems models of the above shape: propositional reasoning, symbolic execution of hybrid programs, and reasoning about loops and differential equations.

Listing 4.10: Proof Search Automation for Hybrid Systems.

```
master = OnAll(prop | step | close | QE | loop | ODE)*
```

In such proofs, branching is prevalent, e.g., due to non-deterministic choices in programs, as well as loop and differential induction. In the proofs so far we specified explicitly how the proof proceeds on each branch using `<()`. This approach is useful to specifically tailor tactics and provide user insight to certain subgoals. In a general-purpose search tactic, however, we neither know *a priori* how many branches there will be, nor how the specific subgoals on each branch are tackled best. The Bellerophon library lets us specify such general-purpose proof search with tactic alternatives `|`, repetition *, and continuing proof search on all branches with `OnAll`. The `prop` tactic is executed first on each subgoal. Running `prop` moves $init$ (of the above-mentioned canonical form $init \rightarrow [\{ctrl;\ plant\}^*]safe$) into the antecedent in the initial theorem, but also performs propositional reasoning on each new subgoal generated by the proof. This enables propositional simplifications both after symbolic execution and loop/ differential induction, as well as to uncover propositional truths handled by `close` and thereby avoid potentially expensive arithmetic reasoning in `QE`. The `step` tactic picks the canonical dynamical axioms for a formula (by indexing techniques) and applies it in the canonical direction. For example, when applied to $[\alpha \cup \beta]P$, the step tactic will produce a new subgoal $[\alpha]P \wedge [\beta]P$. The `step` tactic focuses on the portions of a program that do not need any decisions such as invariants for loops or differential equations. The `loop` tactic generates loop invariants [153] and performs loop induction for the outer control loops, whereas `ODE` handles differential equations. The KeYmaera X implementation of `master` contains several optimizations to the ordering of tactics based upon empirical experience.

The `ODE` and `master` tactics demonstrate how Bellerophons's combinators are used to construct proof search procedures out of components available in the Bellerophon standard library.

4.6 Related Work on Tactical Theorem Proving

The novel contributions of this chapter are the design and implementation of a tactics language and library for hybrid systems which have shown themselves to make tactical proving fruitful for realistic hybrid systems verification tasks.

Tactics Programming Languages Tactics combinators appear in many general-purpose proof assistants, such as NuPRL [38], MetaPRL [91], Isabelle [16], Coq [132], and Lean [47]. However, our goals differ: all of the above aim to work for as many proving domains as possible, while we optimize for hybrid systems proving. In pursuing this aim, we have developed a unique, extensive suite of tactical automation for hybrid systems resting on a small trusted core. We integrate key techniques for continuous systems (ODE solving, invariant generation, and conservation reasoning via differential ghosts) with heuristic simplifications for arithmetic that speed up the use of external real-closed field solvers.

Arithmetic Proving Proving theorems of first-order real arithmetic should not be confused with formalizing real analysis, though both are valuable. General-purpose proof assistants have been used to formalize much of real analysis [27, 81, 111, 167], and in fact some such formalizations [94, 98] have been used to prove the soundness of dℒ's proof calculus on which KeY-

maera X and Bellerophon rest [25]. However, the style of proof used is different: like other domains in which general-purpose provers excel, formalized analysis benefits from the forms of automation that these provers do well, such as automatically expanding definitions and applying syntactic simplification rules. Because hybrid systems verification is less definition-heavy and because simplification rules alone (e.g. ring axioms) do not make real arithmetic tractable, real arithmetic proofs face problems for which existing automation is insufficient. Since arithmetic proofs do arise in these provers as well, we believe our techniques to be of broader interest. While we provide new automation for important tasks, this does not preclude us from using existing tactical techniques for the subtasks where they are most appropriate, such as propositional reasoning and decomposing composite hybrid systems.

Tactical Proving Styles A set of patterns and anti-patterns have been proposed for Coq tactic programming in $\mathcal{L}_{\texttt{tac}}$ [34]. The suggestion is to use general-purpose automation as much as possible, conveying any problem-specific details through hints or lemmas. In keeping with this philosophy, the canonical usage of Bellerophon is to provide loop and sequences of differential invariants as hints to the automated `master` tactic. This reduces the proof to arithmetic. At this point the user can steer the proof further, e.g. by using Bellerophon's equational rewriting mechanisms to reduce complex arithmetic to simpler lemmas. This tactical proof-by-hint style can be mixed freely with other styles provided by the KeYmaera X user interface [134]. For example, a user might use the UI to identify and apply an arithmetic simplification, at which point the corresponding tactic is generated automatically. They might then integrate this tactic into a larger proof-search algorithm which then solves similar proof cases automatically.

4.7 Conclusion

Bellerophon and its standard library support both interactive and automated theorem proving for hybrid systems. The library provides users with a clean interface for expressing common insights that are essential in hybrid systems verification tasks. Bellerophon combinators allow users to combine these base tactics in order to implement proofs and proof search procedures. Through Bellerophon, KeYmaera X provides sound tactical theorem proving for hybrid systems.

Bellerophon provides a useful basis upon which the rest of this thesis is developed. The small core of KeYmaera X is solely responsible for soundness, but provides enough flexibility to reason in many radically different ways about hybrid systems. Bellerophon makes this flexibility easily accessible for programming both high-level hybrid systems verification strategies and concrete case study proofs.

Chapter 5

The Logic of Proofs for Differential Dynamic Logic

This chapter presents a novel logic that extends differential dynamic logic with *proof terms*, the logic of proofs for differential dynamic logic ($\text{LP}_{\text{d}\mathcal{L}}$) [1]. $\text{LP}_{\text{d}\mathcal{L}}$ reifies the structure of derivations of the uniform substitution calculus presented Chapter 2. $\text{LP}_{\text{d}\mathcal{L}}$ is related to the rest of this thesis document in two ways.

The primary relationship between $\text{LP}_{\text{d}\mathcal{L}}$ and the rest of this thesis stems from the observations in Chapter 3 and Chapter 4 that general-purpose provers are more general and have well-developed automation for some domains, but are not as productive as KeYmaera X for hybrid systems theorem proving. This is unfortunately because operating system kernels [107] and compilers [112, 114] are verified in these more general purpose systems, opening up the possibility of not just verified models but end-to-end verification guarantees for compiled control software. Thus, there are clear advantages to reifying the structure of derivations generated by KeYmaera X in the uniform substitution calculus described in Chapter 2. Although this thesis focuses on hybrid systems theorem proving and verifiably safe learning for control, other researchers have taken up the challenge of wedding KeYmaera X to theorem provers such as Isabelle and Coq [26], and those researchers take an approach based on proof reification (although do not use $\text{LP}_{\text{d}\mathcal{L}}$ itself).

The secondary relationship between $\text{LP}_{\text{d}\mathcal{L}}$ and the rest of this thesis is more aspirational. The clear trajectory to be established by the end of Part II is a long-running research program that combines reasoning and learning to build highly trustworthy and highly adaptive control systems. We believe achieving this goal will require building systems that interrogate the structure of their own reasoning process, and performing those introspective operations will require a logic such as $\text{LP}_{\text{d}\mathcal{L}}$that reifies the structure of proofs. Although this direction is not explored in this thesis, it does motivate the development of a sound logical foundations that goes deeper than the more expedient translational approach taken by Bohrer et al. [26].

[1] This chapter is based the paper *A Logic of Proofs for Differential Dynamic Logic: Toward Independently Checkable Proof Certificates for Dynamic Logics* published by Fulton and Platzer [61].

5.1 Introduction

Differential dynamic logic has an implicit notion of proof; there is no way of saying, in the logic, that some sequence of proof rule applications results in a proof of a $\mathsf{d}\mathcal{L}$ formula. Type theories [44, 131] and justification logics [12, 13] have explicit notions of proof; instead of only formulas φ these logics also have formulas of the form $t : \varphi$ meaning t *is a proof of* φ. Theorem proving tools built on logics with an explicit notion of proof have many attractive qualities, including an explicit syntax for exporting proofs generated by the theorem prover.

Unlike theorem provers based upon type-theoretic foundations, theorem provers in the dynamic logic tradition are not based upon logics with a formalized notion of explicit proof evidence. Like several other theorem provers, KeYmaera X ensures soundness by only allowing truth-preserving transformations on formulas, rather than by production of formally defined and independently checkable proof terms. Successful theorem provers and verification tools that are based on logics without proof terms – e.g., KeY, KeYmaera, many first-order theorem provers (e.g., the Logic Theory Machine [136]), Dafny, model checkers, etc. – demonstrates truth-preserving operations on formulas are enough to ensure the soundness of a theorem prover or verification tool.

Although truth-preserving operations are sufficient for ensuring soundness, proof terms address a number of limitations that have arisen during the development and use of the KeYmaera [154] and KeYmaera X theorem provers. KeYmaera and KeYmaera X do not:

- provide a clean separation between proof checking and proof search
- implement a mechanism for composing, reusing, or parameterizing *proofs* (rather than merely mechanisms for composing provability); or
- take advantage of procedures that require interrogating or modifying the structure of a proof.

One advantage of the approach KeYmaera X takes is that there is never a need to re-check proofs obtained via proof search because search always proceeds via operations defined in the soundness-critical core of KeYmaera X. However, ensuring soundness is not the only motivation for separating searching from checking. KeYmaera X allows for parallel speculative proof search, so persisting the particular execution trace of a proof search procedure requires storing and merging proof state using extra-logical operations. Introducing an explicit notion of evidence into differential dynamic logic is a more principled solution than post-hoc analysis of the execution of a search procedure.

The second challenge is surmountable within a single theorem proving session, but is problematic in cases where users collaborate on proofs. Proof terms provide a natural modularity mechanism and allow users to import proven lemmas from other users without re-executing an expensive proof search procedure or blindly trusting the source of the proof.

This chapter presents a *Logic of Proofs for Differential Dynamic Logic* ($\mathrm{LP}_{\mathsf{d}\mathcal{L}}$) . $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ provides an explicit notion of evidence in the form of proof terms – syntactic objects that correspond to deductions in (the uniform substitution calculus of) differential dynamic logic ($\mathsf{d}\mathcal{L}$). Concretely, we assign a syntactic term e to each derivation of ϕ in $\mathsf{d}\mathcal{L}$ such that $e : \phi$ – read as "e is a proof of ϕ" – is a theorem of $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$. We provide a semantics and an axiomatization for this language of proof terms and establish some basic results about the logic and its relation to $\mathsf{d}\mathcal{L}$.

The primary contributions of $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ are:

- a semantic model that extends the standard reachability relation semantics of differential dynamic logic with a notion of evidence (following Fitting [54]).
- a differential dynamic logic with an explicit notion of evidence – a Logic of Proofs for Differential Dynamic Logic ($\mathrm{LP}_{\mathsf{d}\mathcal{L}}$).
- a theorem establishing the correctness of this logic by proving that all pieces of evidence in $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ correspond to a deduction in $\mathsf{d}\mathcal{L}$.

These contributions constitute a logical foundation for hybrid systems with an explicit notion of evidence. $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ was designed with two goals in mind: extracting live control programs from $\mathsf{d}\mathcal{L}$ specifications, and enabling systematic transformations to proofs.

Although $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ was superseded by alternative approaches, the idea of constructing explicit proof witnesses lives on KeYmaera X. Bohrer et al. implemented a verified implementations of a differential dynamic logic proof checker in both Isabelle and Coq [25]. Using their implementation of a variant of our proof term calculus (in which proof terms simulate the proof transformations performed by the KeYmaera X core), these researchers were able to export proofs generated by KeYmaera X to the Isabelle theorem prover. These exported proofs were then checked by other theorem provers and ultimately incorporated into a chain of evidence tying hybrid systems tools to verified compilers [26].

Much of the work in Part II of this dissertation focuses on transformations that simultaneously change a hybrid program and its safety proof. Rapid progress on the Bellerophon language enabled us to implement these transformations as modifications to Bellerophon programs (instead of as modifications to proof terms).

Although the two goals for which $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ was originally designed have been at least partially solved using other methods, $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ none-the-less contributes a logical foundations which could be used to place the above-mentioned implementations on sound logical footing. However, using $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ to improve the robustness of proof transformations and proof translation remains future work.

5.2 Related Work on Representing Proofs

Logics containing explicit representations of proofs have a storied place in the history of mathematical logic and computer science. The Brouwer-Heyting-Kolmogorov semantics for intuitionistic logic is one early and prominent example [90, 109][2]. Type-theoretic theorem provers such as Coq [132] use proof terms as explicit notions of evidence. Conversely, differential dynamic logic has proved to be an excellent setting for verifying complex hybrid dynamical systems [158].

The approach taken in this chapter is motivated primarily by pragmatic concerns related to the construction of certified software controllers for cyber-physical systems. We are particularly interested in developing a notion of evidence that is easy to add to existing theorem provers for differential dynamic logic (or other dynamic logics). For this reason, we take a logic with roots in the modal logic tradition – the Logic of Proofs [12] – as our point of departure with existing work.

[2]For an account of the development of the BKH interpretation, see Section 5.2 of Troelstra's *History of Constructive Mathematics in the 20th Century* [168]

The syntactic restriction placed on formulas containing proof terms is perhaps the most a significant difference between $\text{LP}_{\mathsf{d}\mathcal{L}}$ and modal logics with notions of evidence. In $\text{LP}_{\mathsf{d}\mathcal{L}}$, it is not possible to construct a term of the form $e : e' : \phi$. For this reason, $\text{LP}_{\mathsf{d}\mathcal{L}}$ is considerably less expressive than what one might expect from a full logic of proofs for hybrid systems. However, our concern in this chapter is with *modeling deductions in* $\mathsf{d}\mathcal{L}$, rather than with studying provability in the context of hybrid dynamical systems.

$\text{LP}_{\mathsf{d}\mathcal{L}}$ contains several mechanisms for performing contextual equivalence and equational rewriting. There exist many logics and calculi with primitives for this style of rewriting [4, 172]. Effortless rewriting of deeply nested formulas is a major benefit of Hilbert-style logics, but comes at the cost of less structured proofs.

There are also many existing techniques for augmenting an existing logic with proof terms. The remainder of this section discusses why we chose to design and implement a novel logic rather than some of the most prominent alternatives, and why logic may prove useful in the future even though other approaches (ad hoc tactic rewriting and ad hoc proof term translation) have dominated implementation decisions in KeYmaera X thus far.

There are many reasons for implementing a new theorem prover – especially in the cyber-physical systems domain. KeYmaera X is designed as a platform for research on both automated and interactive theorem proving specifically for hybrid dynamical systems. Designing and implementing new tactics languages, proof construction GUIs, and other features is easier in a smaller system with significantly fewer lines of code, and KeYmaera X was specifically designed to support certain extensions (e.g., parallel proof search, control engineering-centric user interfaces) that Coq (for example) was not designed to support.

Proceeding from the premise that hybrid systems theorem proving benefits from a theorem proving system that is specifically tailored to differential dynamic logics, the primary benefit of the approach in this chapter is that it is parsimonious with the meta-theory of these logics. Both the syntax and semantics of $\text{LP}_{\mathsf{d}\mathcal{L}}$ are a straightforward extension of the semantics of differential dynamic logics.

The rationale for developing a custom theorem prover for differential dynamic logics apply equally to all of the alternatives discussed in this section. The following discussions of particular alternatives focus on more specific comparisons.

Encoding in a Proof Assistant. One alternative is encoding Fig. 2.1 and Fig. 2.2 in a proof assistant such as Coq [132] or Isabelle [140]. The Uniform Substitution algorithm implemented in KeYmaera X is constructive and is implemented in a proof assistant for a higher order logic [25], so this approach is certainly possible. These proof assistants have proof terms, so those proof terms would serve our goal of adding proof terms to $\mathsf{d}\mathcal{L}$.

Even given a formalization of the soundness proof for $\mathsf{d}\mathcal{L}$, the benefit of a proof constructed in a proof assistant remains questionable because the KeYmaera X core is small. For example, although the Coq core is more thoroughly audited than the KeYmaera X core, it is also far larger (the Coq core is approx. 20000 lines of code and the KeYmaera X core is approx. 2000 lines).[3] More importantly, especially for deep embeddings, much of the proof structure is likely lost and

[3]This argument is less strong for HOL Light [80] and Lean [47], both of which have implementations whose size and complexity is comparable to KeYmaera X.

difficult to recover.

Ultimately, the contribution of $\text{LP}_{\mathsf{d}\mathcal{L}}$ relative to encodings is its first-principles logical foundations for dynamic logic proofs.

Logical Frameworks. Logical frameworks [79] provide a potential counter-point to the above observation that formalizing the soundness proof for $\mathsf{d}\mathcal{L}$ would require considerable effort. Work toward a mechanization of Standard ML in Twelf [145] demonstrates that logical frameworks are particularly well-suited to reasoning about binding [122]; this strength is relevant in the context of $\mathsf{d}\mathcal{L}$ because binding structure is at the heart of admissibility constraints on uniform substitutions. However, the binding structure of hybrid programs is rich enough that encoding admissible uniform substitutions would require non-trivial effort. Furthermore, uniform substitution is only the first (and likely easiest) step of a mechanization of $\mathsf{d}\mathcal{L}$ in Twelf, Beluga [146], etc. because obtaining soundness proofs would also require proving the local soundness of the axioms in Fig. 2.2.

5.3 The Logic of Proofs for Differential Dynamic Logic

This section presents the syntax and semantics for $\text{LP}_{\mathsf{d}\mathcal{L}}$. Syntactically, the logic is the differential dynamic logic presented in [151], augmented with formulas of the form $e : \phi$ (where ϕ is a $\mathsf{d}\mathcal{L}$ formula and e is of a new syntactic category which we will call proof terms). The intended meaning of $e : \phi$ is that e serves as evidence for ϕ. Semantically, $\text{LP}_{\mathsf{d}\mathcal{L}}$ extends the semantics of $\mathsf{d}\mathcal{L}$ with meanings for formulas of the form $e : \phi$.

The choice of proof terms presented in this section is motivated by the typical structure of proofs in $\mathsf{d}\mathcal{L}$. Proofs in $\mathsf{d}\mathcal{L}$ combine equivalence/equational reasoning with uniform substitutions and uniform renamings. For example, consider the proof of $[x := 0 \cup x := 1]x \geq 0$ in Fig. 5.1. Each of the leafs of the proof is either an axiom of $\mathsf{d}\mathcal{L}$ or else a tautology of $\text{FOL}_{\mathbb{R}}$. These leafs are obtained from the original problem by performing equivalence rewriting, modus ponens, and identifying uniform substitutions that translate the resulting subgoals into $\mathsf{d}\mathcal{L}$ axioms. In this proof, uniform renaming is not necessary; however, renaming would be necessary for the formula $[y := 0 \cup y := 1]y \geq 0$ because the axiom for symbolically executing a discrete assignment mentions x instead of y.

5.3.1 Syntax

Definition 6 (Formulas)**.** *The formulas of $\text{LP}_{\mathsf{d}\mathcal{L}}$ are defined by extending the inductive definition of $\mathsf{d}\mathcal{L}$ formulas given in Def. 1 with formulas of the form $e : \phi$, where ϕ is a formula of $\mathsf{d}\mathcal{L}$ and e ranges over proof terms (defined below).*

Our definition of the grammar of $\text{LP}_{\mathsf{d}\mathcal{L}}$ formulas (e.g., the inclusion of $\mathsf{d}\mathcal{L}$ formulas) is parsimonious with the Justification Logic tradition rather than the type theory tradition.

$$\text{MP}\,\dfrac{\text{US}\,\dfrac{[\cup]\,\dfrac{}{[a\cup b]p(\bar{x})\leftrightarrow[a]p(\bar{x})\wedge[b]p(\bar{x})}}{[x:=0\cup x:=1]x\geq 0\leftrightarrow[x:=0]x\geq 0\wedge[x:=1]x\geq 0} \qquad \Delta}{[x:=0\cup x:=1]x\geq 0}$$

where $\Delta =$

$$\dfrac{\dfrac{\dfrac{\text{US}\,\dfrac{[:=]\,\dfrac{}{[x:=t]p(t)\leftrightarrow p(x)}}{[x:=0]x\geq 0\leftrightarrow 0\geq 0}\quad \text{Prop}\,\dfrac{\mathbb{R}\,\dfrac{}{0\geq 0}}{[x:=0]x\geq 0\leftrightarrow 0\geq 0\to[x:=0]x\geq 0}}{[x:=0]x\geq 0}\quad \text{MP, Prop, US}\,\dfrac{[:=]\,\dfrac{}{[x:=t]p(t)\leftrightarrow p(x)}\quad \mathbb{R}\,\dfrac{}{1\geq 0}}{[x:=1]x\geq 0}}{[x:=0]x\geq 0\wedge[x:=1]x\geq 0}}{([x:=0\cup x:=1]x\geq 0\leftrightarrow[x:=0]x\geq 0\wedge[x:=1]x\geq 0)\to[x:=0\cup x:=1]x\geq 0}$$

Figure 5.1: A Proof of $[x := 0 \cup x := 1]x \geq 0$ in the Uniform Substitution Calculus of $\mathsf{d}\mathcal{L}$. The proof of Δ is slightly abbreviated for readability; the proof for the $x := 1$ case is very similar to the proof of the $x := 0$ case.

The formulas of $\text{LP}_{\mathsf{d}\mathcal{L}}$ as defined in Def. 6 augment the formulas of $\mathsf{d}\mathcal{L}$ with an additional connective $e : \phi$. [4] This augmentation strictly extends the grammar of $\mathsf{d}\mathcal{L}$. Formulas such as $1 = 1 \wedge 2 = 2$ which do not contain proof terms remain formulas of $\text{LP}_{\mathsf{d}\mathcal{L}}$. However, grammatical constructions of the form $e : e' : \phi$ (and $e : e' : e'' : \phi$, and so on) are *not* formulas of $\text{LP}_{\mathsf{d}\mathcal{L}}$; i.e., proof terms provide evidence only for $\mathsf{d}\mathcal{L}$ derivations – not for $\text{LP}_{\mathsf{d}\mathcal{L}}$ derivations. Although the authors are interested in extending $\text{LP}_{\mathsf{d}\mathcal{L}}$ to properly treat formulas of these forms, our immediate motivations for explicitly representing proofs do not require such a rich language.

Pure $\text{LP}_{\mathsf{d}\mathcal{L}}$ formulas are formulas that do not allow the use of $\mathsf{d}\mathcal{L}$ connectives (such as $(e : \phi) \wedge (d : \phi)$). Pure $\text{LP}_{\mathsf{d}\mathcal{L}}$ formula either a formula of $\mathsf{d}\mathcal{L}$, or a formula of the form $e : \phi$ where e is a proof term and ϕ is a formula of $\mathsf{d}\mathcal{L}$.

Example 5 ($\text{LP}_{\mathsf{d}\mathcal{L}}$ formulas and non-formulas)**.** *The following are non-pure formulas of* $\text{LP}_{\mathsf{d}\mathcal{L}}$ *(where* e, d *are proof terms and* ϕ, ψ *are* $\mathsf{d}\mathcal{L}$ *formulas):*

- $(e : \phi) \wedge (d : \psi)$
- $(e : \phi) \rightarrow (d : \psi)$
- $[x := 0](j_{1=1} : 1 = 1)$

whereas $e : (\phi \wedge \psi)$ *is a pure formula of* $\text{LP}_{\mathsf{d}\mathcal{L}}$*:*

In most of this chapter we are concerned only with pure $\text{LP}_{\mathsf{d}\mathcal{L}}$ formulas, because these are the formulas that correspond to judgements

$$e \text{ is a } \mathsf{d}\mathcal{L} \text{ proof of } \phi$$

where ϕ is a formula of $\mathsf{d}\mathcal{L}$; i.e., pure $\text{LP}_{\mathsf{d}\mathcal{L}}$ formulas are *just* proof certificates for $\mathsf{d}\mathcal{L}$ derivations. In particular, our axioms and proof rules focus only on the pure fragment of $\text{LP}_{\mathsf{d}\mathcal{L}}$. A complete definition of the objects that may stand in for e occupies the remainder of this subsection.

Definition 7 (Proof Terms)**.** *Proof terms are defined by this grammar (with* e, d *as proof terms,* c *ranging over sets of proof constants,* σ *as a uniform substitution,* $\mathscr{B}$ *as a uniform renaming, and* ϕ *as* $\mathsf{d}\mathcal{L}$ *formulas as defined in Def. 1).*

$$
\begin{array}{lll}
\langle e, d\rangle ::= c_\phi & & \textit{Proof Constants} \\
\quad | \quad e \wedge d & & \textit{Conjunctions} \\
\quad | \quad e \bullet d & & \textit{Implicative Application} \\
\quad | \quad e \bullet_{\leftarrow} d \mid e \bullet_{\rightarrow} d & & \textit{Directional Equivalence Application} \\
\quad | \quad \sigma e \mid \mathscr{B} e & & \textit{Uniform Substitutions and Bound Renaming} \\
\quad | \quad CT_\sigma e \mid CQ_\sigma e \mid CE_\sigma e & & \textit{Equivalence/equational Reasoning}
\end{array}
$$

Proof terms are the syntactic objects of $\text{LP}_{\mathsf{d}\mathcal{L}}$ corresponding to deductions in $\mathsf{d}\mathcal{L}$.

Atomic/Axiomatic Terms. Proof constants serve as evidence for $\mathsf{d}\mathcal{L}$ axioms and $\text{FOL}_{\mathbb{R}}$ tautologies. In this chapter, we consider two sets of proof constants – i_A where A is any $\mathsf{d}\mathcal{L}$ axiom and j_T where T is any tautology of $\text{FOL}_{\mathbb{R}}$. We use c_ϕ whenever we mean to discuss both of these sets of proof constants.

The separation of atomic proof terms indexed by concrete axioms into disjoint sets is motivated by practical concerns that arise when implementing a theorem prover for hybrid systems.

[4] It is not misleading to think of $\_ : \_$ as a binary function mapping proofs terms and $\mathsf{d}\mathcal{L}$ formulas to $\text{LP}_{\mathsf{d}\mathcal{L}}$ formulas.

The benefit of separating atomic proof terms into sets is a clear separation between axiomatic and real arithmetic reasoning. Although the first-order theory of real arithmetic is decidable [37], the problem has extreme complexity [53]. Furthermore, KeYmaera X (as well as other theorem provers) that utilize decision procedures for real arithmetic are typically sound only modulo the soundness of an external implementation of the decision procedure being used. Distinguishing computationally trivial appeals to axioms from possibly expensive appeals to arithmetic decision procedures isolates a natural extension point for incorporating certificates of arithmetic facts e.g., by extracting witnesses from an implementation of a Coq implementation of the Cylindrical Algebraic Decomposition algorithm [129] or by using approaches such as [156] that are amenable to certificate generation. Isolating real arithmetic facts from axiomatic facts also makes it very easy to identify appeals to $\text{FOL}_{\mathbb{R}}$ tautologies in proofs, which could be useful for identifying when the reproducibility of a proof is going to depend upon possibly expensive appeals to a decision procedure.

Conjunctions. The $\wedge$ operator allows for the creation of evidence for conjunctive formulas. If $e : \phi$ and $d : \psi$ then $(e \wedge d) : \phi \wedge \psi$. This connective is also not strictly necessary if $\mathsf{d}\mathcal{L}$ contains appropriate propositional axioms but is useful because many $\mathsf{d}\mathcal{L}$ axioms contain conjunctions. Conjunctions represent the exact structure of a proof, so $\text{LP}_{\mathsf{d}\mathcal{L}}$ excludes the $+$ operator found in some Justification Logics ([13, Part II]) because we are interested only in single conclusion proof systems. From an implementation perspective, the most interesting multi-conclusion extensions are those that could serve as a category of values for a proof search specification language capable of describing decidable but non-deterministic forward proof search procedures.

Implications. The $\bullet$ operator allows the use of evidence of an implication, and corresponds to the modus ponens proof rule. For example, if $e : \psi \to \phi$ and $d : \psi$ then $e \bullet d : \phi$. This operator corresponds to the application operator of the Logic of Proofs and corresponds to application in the Simply Typed Lambda Calculus.

Equivalence Rewriting. The $\bullet_{\leftarrow}$ and $\bullet_{\rightarrow}$ operators are similar to the implication operator, but are used for equivalences instead of implications. The subscript on the operator indicates the direction in which the equivalence should be used. For example, if $e : \psi \leftrightarrow \phi$ and $d : \phi$ then $e \bullet_{\leftarrow} d : \psi$. The $\bullet_{\leftarrow}$ and $\bullet_{\rightarrow}$ operators are not strictly necessary because they can be replaced with axioms. If

$$\begin{aligned} &i : (\phi \leftrightarrow \psi) \to (\phi \to \psi), \\ &e : \phi \leftrightarrow \psi, \text{and} \\ &d : \phi \end{aligned}$$

then $(i \bullet e) \bullet d : \psi$. These operators are included because equivalence rewriting is a fundamental and pervasive operation in axiomatic proofs, so even the constant multiplier on the length of proof terms is enough to motivate the addition of operators.

Substitution and Renaming. Uniform substitution and renaming are essential parts of dL proofs and are witnessed by proof terms of the form σe and $\mathscr{B}e$, where σ and $\mathscr{B}$ are uniform substitutions and renamings respectively. Uniform substitutions do not map variables to variables, but variable renamings are necessary whenever a proof contains variables that do not occur in axioms. For example, a proof of $[a := 12]a = 12 \leftrightarrow 12 = 12$ is just a uniform renaming of x to a in the $[:=]$ axiom. KeYmaera X allows explicit uniform renamings during proving, and these explicit renamings are captured by the $\mathscr{B}$ proof terms.

Equivalence and Equational Reasoning. The CT_σ, CQ_σ, and CE_σ operators correspond to uniform substitution instances of the contextual equation and equivalence proof rules of dL (CT, CQ, and CE). For example, the proof term $\text{CT}_{\{c(\cdot)\mapsto\cdot^2, f(\cdot)\mapsto b, g(\cdot)\mapsto a\}}$ serves as evidence for the $\{c(\cdot) \mapsto \cdot^2, f(\cdot) \mapsto a, g(\cdot) \mapsto b\}$ uniform substitution instance of the CT proof rule:

$$\frac{a = b}{a^2 = b^2}$$

5.3.2 Semantics

The semantics of LP_{dL} formulas extends the semantics of the uniform substitution calculus presented in Chapter 2. As in dL, interpretations I in LP_{dL} give meaning to program constants, function, predicate and quantifier symbols [151].

Definition 8 (LP_{dL} Semantics)**.** *The semantics of an LP_{dL} formula χ is defined with respect to an interpretation I as the subset $[\![\chi]\!]^I \subseteq S$ of states in which χ is true and is defined inductively as follows (where i_A, j_T ranges over proof constants, e, d range over proof terms, and ϕ, ψ range over dL formulas):*

- $[\![\phi]\!]^I = [\![\phi]\!]^I_{\text{dL}}$ *where* $[\![\cdot]\!]_{\text{dL}}$ *is the denotation of dL given in [151]. The meaning of connectives* $\wedge, \neg, \exists, [\cdot], \langle\cdot\rangle$ *is also as in dL, e.g.,* $[\![\varphi \wedge \chi]\!]^I = [\![\varphi]\!]^I \cap [\![\chi]\!]^I$
- $[\![i_A : A]\!]^I = S$ *for dL axioms* A
- $[\![j_T : T]\!] = S$ *for* $\text{FOL}_{\mathbb{R}}$ *tautologies* T
- $[\![e \wedge d : \phi \wedge \psi]\!]^I = [\![e : \phi]\!]^I \cap [\![d : \psi]\!]^I = \{v \in S : v \in [\![e : \phi]\!]^I$ *and* $v \in [\![d : \psi]\!]^I\}$
- $[\![e \bullet d : \phi]\!]^I = \bigcup_\psi [\![e : (\psi \to \phi)]\!]^I \cap [\![d : \psi]\!]^I$
  $= \{v \in S : v \in [\![e : (\psi \to \phi)]\!]^I$ *and* $v \in [\![d : \psi]\!]^I$ *for some* $\psi\}$
- $[\![e \bullet_\leftarrow d : \phi]\!]^I = \bigcup_\psi [\![e : (\phi \leftrightarrow \psi)]\!]^I \cap [\![d : \psi]\!]^I$
  $= \{v \in S : v \in [\![e : (\phi \leftrightarrow \psi)]\!]^I$ *and* $v \in [\![d : \psi]\!]^I$ *for some* $\psi\}$
- $[\![e \bullet_\rightarrow d : \phi]\!]^I = \bigcup_\psi [\![e : (\psi \leftrightarrow \phi)]\!]^I \cap [\![d : \psi]\!]^I$
  $= \{v \in S : v \in [\![e : \psi \leftrightarrow \phi]\!]^I$ *and* $v \in [\![d : \psi]\!]^I$ *for some* $\psi\}$
- $[\![\sigma e : \sigma\phi]\!]^I = [\![e : \phi]\!]^I$ *if* σ *is admissible for* ϕ
  $= \{v \in S : v \in [\![e : \phi]\!]^I$ *and* σ *is admissible for* $\phi\}$.
- $[\![\mathscr{B}e : \mathscr{B}\phi]\!]^I = [\![e : \phi]\!]^I$ *if* $\mathscr{B}$ *is a uniform renaming of* ϕ
  $= \{v \in S : v \in [\![e : \phi]\!]^I$ *and* $\mathscr{B}$ *is a uniform renaming of* $\phi\}$
- $[\![\mathit{CT}_\sigma e : \sigma(c(f(\bar{x})) = c(g(\bar{x})))]\!]^I = [\![\sigma e : \sigma(f(\bar{x}) = g(\bar{x}))]\!]^I$
- $[\![\mathit{CQ}_\sigma e : \sigma(p(f(\bar{x})) \leftrightarrow p(g(\bar{x})))]\!]^I =$
  $[\![\sigma e : \sigma(f(\bar{x}) = g(\bar{x}))]\!]^I$

- $[\![CE_\sigma e : \sigma(C(p(\bar{x})) \leftrightarrow C(q(\bar{x})))]\!]^I =$
  $[\![\sigma e : \sigma(p(\bar{x}) \leftrightarrow q(\bar{x}))]\!]^I$

Undefined cases are empty.[5]

Note that the meaning of $e : \phi$ is always either S or $\emptyset$ because e either is a proof of ϕ or is not a proof of ϕ; in other words, the validity of a proof does not depend on the state in which the proof is evaluated.

We do not prove soundness in this chapter; instead, we establish a correctness result that is more useful in our context: whenever $e : \phi$ is a theorem of $\text{LP}_{\mathsf{d}\mathcal{L}}$, we can construct a $\mathsf{d}\mathcal{L}$ proof of ϕ, which implies that ϕ is valid. In this section, we take a similar approach. Instead of establishing a direct connection between the semantics and axioms and proof rules of $\text{LP}_{\mathsf{d}\mathcal{L}}$, we instead establish a projection from the semantics of $\text{LP}_{\mathsf{d}\mathcal{L}}$ to the semantics of $\mathsf{d}\mathcal{L}$.

Theorem 1 (Correctness of Proof Term Valuation)**.** *Consider any interpretation I, $v \in S$ and $\mathsf{d}\mathcal{L}$ formula ϕ. If $v \in [\![e : \phi]\!]^I_{LP_{\mathsf{d}\mathcal{L}}}$ then $v \in [\![\phi]\!]^I_{\mathsf{d}\mathcal{L}}$.*

Note that Theorem 1 pertains only to pure $\text{LP}_{\mathsf{d}\mathcal{L}}$ formulas; i.e., $\text{LP}_{\mathsf{d}\mathcal{L}}$ formulas of the form $e : \phi$ where e is a proof term and ϕ is a formula of $\mathsf{d}\mathcal{L}$.

Proof. The proof proceeds by induction on the structure of e, simultaneously for all ϕ.

Axiomatic Terms. Suppose $v \in [\![i_\psi : \phi]\!]^I_{\text{LP}_{\mathsf{d}\mathcal{L}}}$. By Def. 8, it must be that ϕ is ψ and ψ is an axiom of $\mathsf{d}\mathcal{L}$. Therefore, ϕ is an axiom of $\mathsf{d}\mathcal{L}$ so by soundness of $\mathsf{d}\mathcal{L}$, $[\![\phi]\!]^I_{\mathsf{d}\mathcal{L}} = S$. Finally, $v \in S$.

$\text{FOL}_\mathbb{R}$ Tautology Terms. Suppose $v \in [\![j_\psi : \phi]\!]^I_{\text{LP}_{\mathsf{d}\mathcal{L}}}$. By Def. 8, it must be that ϕ is ψ and ψ is a tautology of $\text{FOL}_\mathbb{R}$. Therefore, ϕ is a tautology of $\text{FOL}_\mathbb{R}$ so by soundness of $\mathsf{d}\mathcal{L}$, $[\![\phi]\!]^I_{\mathsf{d}\mathcal{L}} = S$. Finally, $v \in S$.

Case $e \wedge d$. Suppose $v \in [\![e \wedge d : \phi]\!]^I_{\text{LP}_{\mathsf{d}\mathcal{L}}}$. Inspecting the cases of Def. 8, it must be that

$$\phi = \varphi \wedge \psi$$

for some φ, ψ such that

$$v \in [\![e : \varphi]\!]^I_{\text{LP}_{\mathsf{d}\mathcal{L}}} \tag{5.1}$$

$$v \in [\![d : \psi]\!]^I_{\text{LP}_{\mathsf{d}\mathcal{L}}} \tag{5.2}$$

Applying the inductive hypothesis at (5.1) and (5.2), we have $v \in [\![\varphi]\!]^I_{\mathsf{d}\mathcal{L}}$ and $v \in [\![\psi]\!]^I_{\mathsf{d}\mathcal{L}}$ from which it follows that

$$v \in [\![\varphi]\!]^I_{\mathsf{d}\mathcal{L}} \cap [\![\psi]\!]^I_{\mathsf{d}\mathcal{L}} = [\![\varphi \wedge \psi]\!]^I_{\mathsf{d}\mathcal{L}}$$

by the definition of the semantics of $\mathsf{d}\mathcal{L}$ [151].

Case $e \bullet d$. Suppose $v \in [\![e \bullet d : \phi]\!]^I_{\text{LP}_{\mathsf{d}\mathcal{L}}}$. By Def. 8 we know that

$$v \in [\![e : \psi \to \phi]\!]^I_{\text{LP}_{\mathsf{d}\mathcal{L}}}$$

$$v \in [\![d : \psi]\!]^I_{\text{LP}_{\mathsf{d}\mathcal{L}}}$$

[5] E.g., $[\![(e \wedge d) : \phi]\!]^I = \emptyset$ whenever ϕ is not of the appropriate form. Likewise for the other cases.

for some ψ. Applying the inductive hypothesis to these facts establishes

$$v \in [\![\psi \to \phi]\!]^I_{\mathsf{d}\mathcal{L}}$$
$$v \in [\![\psi]\!]^I_{\mathsf{d}\mathcal{L}}$$

From these facts, a classical propositional encoding of $\psi \to \phi$, and elementary theorems of set theory, we obtain that

$$v \in ([\![\psi]\!]^I_{\mathsf{d}\mathcal{L}})^{\mathrm{C}} \cup [\![\phi]\!]^I_{\mathsf{d}\mathcal{L}}$$

(where X^{C} is the set complement $S \setminus X$ of X) which, because $v \in [\![\psi]\!]^I_{\mathsf{d}\mathcal{L}}$, implies $v \in [\![\phi]\!]^I_{\mathsf{d}\mathcal{L}}$.

Case $e \bullet_{\leftarrow} d$ **and** $e \bullet_{\rightarrow} d$**.** Symmetric to $e \bullet d$.

Case σe**.** Suppose that $v \in [\![\sigma e : \phi]\!]^I_{\mathrm{LP}_{\mathsf{d}\mathcal{L}}}$. Then by inspection of the cases of Def. 8, $\phi = \sigma(\phi')$ and $v \in [\![e : \phi']\!]^I_{\mathrm{LP}_{\mathsf{d}\mathcal{L}}}$ Applying the inductive hypothesis to this fact establishes $v \in [\![\phi']\!]^I_{\mathsf{d}\mathcal{L}}$. Note that σ is, by Def. 8, an admissible substitution for ϕ'. From this fact and the previously established fact that $v \in [\![\phi']\!]^I_{\mathsf{d}\mathcal{L}}$, it follows that $v \in [\![\sigma(\phi')]\!]^I_{\mathsf{d}\mathcal{L}}$ by the uniform substitution proof rule and the soundness of $\mathsf{d}\mathcal{L}$. So, by our previous observation that $\phi = \sigma(\phi')$, $v \in [\![\phi]\!]^I_{\mathsf{d}\mathcal{L}}$.

Case $\mathscr{B}e$**.** Suppose that $v \in [\![\mathscr{B}e : \phi]\!]^I_{\mathrm{LP}_{\mathsf{d}\mathcal{L}}}$. Then by inspection of the cases of Def. 8, $\phi = \mathscr{B}(\phi')$ and $v \in [\![e : \phi']\!]^I_{\mathrm{LP}_{\mathsf{d}\mathcal{L}}}$ Applying the inductive hypothesis to this fact establishes $v \in [\![\phi']\!]^I_{\mathsf{d}\mathcal{L}}$. Note that $\mathscr{B}$ is, by Def. 8, an admissible substitution for ϕ'. From this fact and the previously established fact that $v \in [\![\phi']\!]^I_{\mathsf{d}\mathcal{L}}$, it follows that $v \in [\![\mathscr{B}(\phi')]\!]^I_{\mathsf{d}\mathcal{L}}$ by the uniform substitution proof rule and the soundness of $\mathsf{d}\mathcal{L}$. So, by our previous observation that $\phi = \mathscr{B}(\phi')$, $v \in [\![\phi]\!]^I_{\mathsf{d}\mathcal{L}}$.

Case $CT_\sigma e$. Suppose that $v \in [\![CT_\sigma e : \phi]\!]^I_{\mathrm{LP}_{\mathsf{d}\mathcal{L}}}$. By inspection of the cases of Def. 8, $\phi = \sigma(c(f(\bar{x})) = c(g(\bar{x})))$ and $v \in [\![e : c(f(\bar{x})) = c(g(\bar{x}))]\!]$. Note that σ is admissible for $c(f(\bar{x})) = c(g(\bar{x}))$, so $v \in [\![\sigma(c(f(\bar{x})) = c(g(\bar{x})))]\!]^I_{\mathsf{d}\mathcal{L}} = [\![\phi]\!]^I_{\mathsf{d}\mathcal{L}}$. The remaining contextual cases are similar.

□

5.3.3 Axioms and Proof Rules

Axioms governing the construction of proof terms allow for the derivation of proof terms that describe proofs by substitution, uniform renaming, uniform substitution, and appeals to axioms and tautologies. This is sufficient to describe proofs constructed by the uniform substitution calculus of dL, and by extension most proofs constructed by the KeYmaera X theorem prover. The KeYmaera X theorem prover also contains a propositional sequent calculus and skolemization [147], so in practice some proofs constructed by KeYmaera X may not have proof terms in LP_{dL}. However, there exist proof term calculi for propositional sequent calculi, so this chapter focuses on the portions of KeYmaera X proofs that do not yet have an easily adaptable proof term calculus.

After stating the axioms and proof rules of LP_{dL} in Def. 9, we describe how each is used to construct proof terms for typical constructions.

Unlike dL, LP_{dL} does not use uniform substitutions. Therefore, the objects described in the following definition are axiom schemata and proof rules – not just formulas or pairs of formulas.

The axioms in Def. 9 correspond to the intuitive meanings for proof terms given in Section 5.3.1.

Proof Constant Axioms. The axiomatization of dL is included in LP_{dL} in the form of including all provable dL formulas (rule dL Axiom). Proof constants i_A and j_T internalize evidence for dL axioms and $\text{FOL}_{\mathbb{R}}$ tautologies. For example,

$$i_{[a;b]p(\bar{x})\leftrightarrow[a][b]p(\bar{x})} : [a;b]p(\bar{x}) \leftrightarrow [a][b]p(\bar{x}), \text{ and}$$
$$j_{x\geq 0\rightarrow x^2\geq 0} : x \geq 0 \rightarrow x^2 \geq 0$$

are both axioms of LP_{dL}. For brevity, we often use the names of axioms as subscripts instead of the axioms themselves. For example,

$$i_{[\cup]} : [a \cup b]p(\bar{x}) \leftrightarrow [a]p(\bar{x}) \wedge [b](\bar{x}).$$

Conjunction Proof Rule. The And proof rule enables construction of compound proof terms that serve as evidence for conjunctions. Constructing a proof term that allows for left and right projections of a conjunction is also possible using dL axioms and Application axiom, so these are not included as primitives. Unlike dL, proof term axioms and proof rules are schematic, so

$$\frac{d : x = y \qquad e : y = z}{(d \wedge e) : x = y \wedge y = z}$$

is a derivation in LP_{dL}.

Definition 9 (Axioms of LP_{dL})**.** *The following are axiom schemata of LP_{dL}, where φ, ψ range over LP_{dL} formulas, and c, f, g are function symbols and p, q are predicate symbols, and C a*

quantifier symbol.

$$\frac{}{\phi} \quad \text{(dℒ Axiom)}$$

$$\frac{}{i_A : A} \quad \text{(dℒ Constants)}$$

$$\frac{}{j_T : T} \quad \text{(FOL}_{\mathbb{R}}\text{ Constants)}$$

$$\frac{e : \phi \qquad d : \psi}{(e \wedge d) : (\phi \wedge \psi)} \quad \text{(And)}$$

$$\frac{e : (\phi \to \psi) \qquad d : \phi}{e \bullet d : \psi} \quad \text{(Application)}$$

$$\frac{e : (\phi \leftrightarrow \psi) \qquad d : \phi}{e \bullet_{\to} d : \psi} \quad \text{(Right Equivalence)}$$

$$\frac{e : (\phi \leftrightarrow \psi) \qquad d : \psi}{e \bullet_{\leftarrow} d : \phi} \quad \text{(Left Equivalence)}$$

$$\frac{e : \phi}{\sigma e : \sigma(\phi)} \quad \text{(US Proof Term)}$$

$$\frac{e : \phi}{\mathscr{B} e : \mathscr{B}(\phi)} \quad \text{(Renaming)}$$

$$\frac{\sigma e : \sigma(f(\bar{x}) = g(\bar{x}))}{CT_{\sigma} e : \sigma(c(f(\bar{x}) = c(g(\bar{x})))} \quad \text{(CT}_{\sigma}\text{)}$$

$$\frac{\sigma e : \sigma(f(\bar{x}) = g(\bar{x}))}{CQ_{\sigma} e : \sigma(p(f(\bar{x}) \leftrightarrow p(g(\bar{x})))} \quad \text{(CQ}_{\sigma}\text{)}$$

$$\frac{\sigma e : \sigma(p(\bar{x}) \leftrightarrow q(\bar{x}))}{CE_{\sigma} e : \sigma(C(p(\bar{x}) \leftrightarrow C(q(\bar{x})))} \quad \text{(CE}_{\sigma}\text{)}$$

and where the rules US Proof Term, CT_{σ}, CQ_{σ}, and CE_{σ} are applicable only whenever σ is admissible for the dℒ formulas to which it is applied, and only whenever σ has no free variables. The set of free variables of a substitution is defined in [151]. The formula ϕ in (dℒ Axiom) must be a dℒ formula provable in dℒ.

Application Proof Rules. The Application proof rule enables construction of proof terms that correspond to the use of the Modus Ponens rule in dℒ; for example,

$$\frac{d : p(x) \to q(x) \qquad e : p(x)}{e \bullet d : q(x)}$$

is a derivation in $\text{LP}_{\mathsf{d}\mathcal{L}}$. The Left Equivalence and Right Equivalence rules are definable in terms of the Application rule at the expense of more verbose proof terms.

Uniform Substitution Proof Rule. The US Proof Term axiom allows the construction of evidence that appeals to uniform substitutions. Similarly, uniform renaming is evidenced by Renaming. A schematic sequent calculus for dℒ is definable using uniform substitutions [64] and proof terms can be assigned to each of these proof rules. For example, the proof terms for the sequent calculus proof rule

$$\frac{\vdash [\alpha]\varphi \qquad \vdash [\beta]\varphi}{\vdash [\alpha \cup \beta]\varphi}$$

are $\sigma i_{[\cup]} \bullet_{\rightarrow} e : [\alpha]\varphi \wedge [\beta]\varphi$ where $e : [\alpha \cup \beta]\varphi$ and $\sigma = \{a \mapsto \alpha, b \mapsto \beta, p(\cdot) \mapsto \varphi\}$.

Equivalence/Equational Proof Rules. The CT_σ, CQ_σ, and CE_σ proof rules combine uniform substitutions with the proof rules CT, CQ, and CE from dℒ.

Example 6 demonstrates how these axioms and proof rules are combined with the axioms and uniform substitutions of dℒ to construct witnesses for dℒ proofs by constructing a proof term corresponding to the previous example.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{i_{[\cup]}\colon [a\cup b]p(\bar{x}) \leftrightarrow [a]p(\bar{x})\wedge[b]p(\bar{x})}\,\mathsf{dL}\text{ Constants}}{\sigma_1 i_{[\cup]}\colon [x:=0\cup x:=1]x\geq 0 \leftrightarrow [x:=0]x\geq 0\wedge[x:=1]x\geq 0}\,\text{US Proof Term} \quad \Delta_\wedge}{\sigma_1 i_{[\cup]} \bullet\!\leftarrow \underbrace{((\sigma_2 i_{[:=]} \bullet\!\leftarrow j_{0\geq 0})\wedge(\sigma_3 i_{[:=]}\bullet\!\leftarrow j_{1\geq 0}))}_{e_\wedge}\colon [x:=0\cup x:=1]x\geq 0}\,\text{Left Equivalence}$$

where $\Delta_\wedge$ is

$$\dfrac{\dfrac{\dfrac{\dfrac{}{i_{[:=]}\colon [x:=t]p(x)\leftrightarrow p(t)}}{\sigma_2 i_{[:=]}\colon [x:=0]x\geq 0\leftrightarrow x\geq 0} \quad \dfrac{}{j_2 : 0\geq 0}}{\sigma_2 i_{i[:=]}\bullet\!\leftarrow j_2 : [x:=0]x\geq 0}\,\text{And} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{i_{[:=]}\colon [x:=t]p(x)\leftrightarrow p(t)}\,\mathsf{dL}\text{ Constants}}{\sigma_3 i_{[:=]}\colon [x:=1]x\geq 0\leftrightarrow x\geq 0}\,\text{US Proof Term} \quad \dfrac{}{j_3 : 1\geq 0}\,\text{FOL}_{\mathbb{R}}\text{ Constants}}{\sigma_3 i_{[:=]}\bullet\!\leftarrow j_3 : [x:=1]x\geq 0}}{\underbrace{((\sigma_2 i_{[:=]}\bullet\!\leftarrow j_{0\geq 0})\wedge(\sigma_3 i_{[:=]}\bullet\!\leftarrow j_{1\geq 0}))}_{e_\wedge} : [x:=0]x\geq 0\wedge[x:=1]x\geq 0}\,\text{Left Equivalence}$$

Example 6 (A Simple Proof Term.)**.** *A proof of*

$$(\sigma_1 i_{[\cup]}\bullet\!\leftarrow((\sigma_2 i_{[:=]}\bullet\!\leftarrow j_{0\geq 0})\wedge(\sigma_3 i_{[:=]}\bullet\!\leftarrow j_{1\geq 0}))) : [x:=0\cup x:=1]x\geq 0$$

where

$$\begin{aligned}\sigma_1 &\equiv \{a\mapsto x:=0, b\mapsto x:=1, p(\cdot)\mapsto x\geq 0\}\\ \sigma_2 &\equiv \{t\mapsto 0, p(\cdot)\mapsto \cdot\geq 0\}\\ \sigma_3 &\equiv \{t\mapsto 1, p(\cdot)\mapsto \cdot\geq 0\}\\ i_{[\cup]} &\equiv i_{[a\cup b]p(\bar{x})\leftrightarrow[a]p(\bar{x})\wedge[b]p(\bar{x})}\\ i_{[:=]} &\equiv i_{[x:=t]p(x)\leftrightarrow p(t)}\end{aligned}$$

is given above. Intuitively, this property states that if x nondeterministically takes on 0 or 1, then $x\geq 0$. The proof proceeds by symbolic decomposition of the hybrid program $x:=0\cup x:=1$ using axioms of dL. Uniform substitution instances of the relevant symbolic decomposition axioms are necessary in order to complete the proof. Labels on the left side of the proof of Δ are elided for readability, but exactly match the labels on the right side.

5.4 Converting $\text{LP}_{\mathsf{d}\mathcal{L}}$ Proof Terms into $\mathsf{d}\mathcal{L}$ Proofs

We say that $\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} \phi$ whenever there is a proof of ϕ in $\text{LP}_{\mathsf{d}\mathcal{L}}$, and we say that $\vdash_{\mathsf{d}\mathcal{L}} \phi$ whenever there is a proof of ϕ in $\mathsf{d}\mathcal{L}$.

Lemma 1 (Inversion)**.** *The following are facts about $LP_{\mathsf{d}\mathcal{L}}$:*

- *If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} i_\phi : \psi$ then ϕ is ψ and ϕ is an axiom of $\mathsf{d}\mathcal{L}$.*
- *If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} j_\phi : \psi$ then ϕ is ψ and ϕ is a tautology of $FOL_{\mathbb{R}}$.*
- *If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e \wedge d : \phi$ then ϕ is $(\chi \wedge \psi)$ where $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e : \chi$ and $\vdash_{LP_{\mathsf{d}\mathcal{L}}} d : \psi$.*
- *If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e \bullet d : \phi$ then $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e : \psi \to \phi$ and $\vdash_{LP_{\mathsf{d}\mathcal{L}}} d : \psi$ for some ψ.*
- *If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e \bullet_{\leftarrow} d : \phi$ then $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e : \phi \leftrightarrow \psi$ and $\vdash_{LP_{\mathsf{d}\mathcal{L}}} d : \psi$ for some ψ.*
- *If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e \bullet_{\rightarrow} d : \phi$ then $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e : \psi \leftrightarrow \phi$ and $\vdash_{LP_{\mathsf{d}\mathcal{L}}} d : \psi$ for some ψ.*
- *If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} \mathbf{CT}_\sigma e : \phi$ then ϕ is $\sigma(c(f(\bar{x})) = c(g(\bar{x})))$, $\vdash_{LP_{\mathsf{d}\mathcal{L}}} \sigma e : \sigma(f(\bar{x}) = g(\bar{x}))$, and σ is admissible on all formulas to which it is applied.*
- *If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} \mathbf{CQ}_\sigma e : \phi$ then ϕ is $\sigma(p(f(\bar{x})) \leftrightarrow p(g(\bar{x})))$, $\vdash_{LP_{\mathsf{d}\mathcal{L}}} \sigma e : \sigma(f(\bar{x}) = g(\bar{x}))$, and σ is admissible on all formulas to which it is applied.*
- *If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} \mathbf{CE}_\sigma e : \phi$ then ϕ is $\sigma(C(p(\bar{x})) \leftrightarrow C(q(\bar{x})))$, $\vdash_{LP_{\mathsf{d}\mathcal{L}}} \sigma e : \sigma(p(\bar{x}) \leftrightarrow q(\bar{x}))$, and σ is admissible on all formulas to which it is applied.*
- *If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} \sigma e : \phi$ then $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e : \phi'$ and $\sigma(\phi') = \phi$ for some ϕ' such that σ is admissible for ϕ'.*
- *If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} \mathscr{B} e : \phi$ then $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e : \phi'$ and $\mathscr{B}(\phi') = \phi$ for some ϕ'.*

Proof. The proof involves a straightforward induction involving inspection of the conclusions of $\text{LP}_{\mathsf{d}\mathcal{L}}$ axioms. □

Theorem 2 (Proof terms justify theorems)**.** *Let e be a proof term and ϕ a $\mathsf{d}\mathcal{L}$ formula. If $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e : \phi$ then $\vdash_{\mathsf{d}\mathcal{L}} \phi$.*

Proof. The proof involves the construction of a $\mathsf{d}\mathcal{L}$ proof corresponding to the proof term e. We proceed by induction on the structure of e.

Case i_A**.** Suppose that $\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} i_A : \phi$. By Lemma 1, $\phi = A$ and is an axiom of $\mathsf{d}\mathcal{L}$. Therefore, $\vdash_{\mathsf{d}\mathcal{L}} \phi$.

Case j_T**.** Suppose that $\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} i_A : \phi$. By Lemma 1, $\phi = A$ and is a tautology of $\text{FOL}_{\mathbb{R}}$. Therefore, $\vdash_{\mathsf{d}\mathcal{L}} \phi$.

Case $e \wedge d$**.** Suppose that $e \wedge d : \phi$. By Lemma 1,

$$\phi = \chi \wedge \psi$$

and

$$\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} e : \chi \tag{5.3}$$
$$\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} d : \psi \tag{5.4}$$

Applying the inductive hypothesis to (5.3) and (5.4) establishes that

$$\vdash_{\mathsf{d}\mathcal{L}} \chi \tag{5.5}$$
$$\vdash_{\mathsf{d}\mathcal{L}} \psi \tag{5.6}$$

The schematic proof rule

$$(\wedge\text{R})\ \frac{\varphi \quad \Omega}{\varphi \wedge \Omega}$$

where φ and Ω are any $\mathsf{d}\mathcal{L}$ formulas that are derivable in $\mathsf{d}\mathcal{L}$ using the propositional tautology $\varphi \to \Omega \to \varphi \wedge \Omega$ and MP. From (5.5) and (5.6), andR derives $\vdash_{\mathsf{d}\mathcal{L}} \chi \wedge \psi$.

Case $e \bullet d$**.** Suppose that $\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} e \bullet d : \phi$. By Lemma 1,

$$\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} e : \psi \to \phi \tag{5.7}$$
$$\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} d : \psi \tag{5.8}$$

Applying the inductive hypothesis to (5.7) and (5.8) establishes that

$$\vdash_{\mathsf{d}\mathcal{L}} \psi \to \phi \tag{5.9}$$
$$\vdash_{\mathsf{d}\mathcal{L}} \psi \tag{5.10}$$

from which MP derives $\vdash_{\mathsf{d}\mathcal{L}} \phi$.

Case $e \bullet_{\to} d$**.** Suppose $\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} e \bullet_{\to} d : \phi$. By Lemma 1,

$$\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} e : \psi \leftrightarrow \phi \tag{5.11}$$
$$\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} d : \psi \tag{5.12}$$

are provable in $\text{LP}_{\mathsf{d}\mathcal{L}}$. Applying the inductive hypothesis to (5.11) and (5.12) establishes

$$\vdash_{\mathsf{d}\mathcal{L}} \psi \leftrightarrow \phi \tag{5.13}$$
$$\vdash_{\mathsf{d}\mathcal{L}} \psi \tag{5.14}$$

Note that

$$\vdash_{\mathsf{d}\mathcal{L}} (\psi \leftrightarrow \phi) \to (\psi \to \phi)$$

has a proof in $\mathsf{d}\mathcal{L}$. With (5.13), MP , thus, derives $\vdash_{\mathsf{d}\mathcal{L}} \psi \to \phi$. Applying MP once more to $\psi \to \phi$ with (5.14) establishes that $\vdash_{\mathsf{d}\mathcal{L}} \phi$.

Case $e \bullet_{\leftarrow} d$**.** Suppose $\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} e \bullet_{\leftarrow} d : \phi$. By Lemma 1,

$$\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} e : \phi \leftrightarrow \psi \tag{5.15}$$
$$\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} d : \psi \tag{5.16}$$

are provable in $\text{LP}_{\mathsf{d}\mathcal{L}}$. Applying the inductive hypothesis to (5.15) and (5.16) establishes

$$\vdash_{\mathsf{d}\mathcal{L}} \phi \leftrightarrow \psi \tag{5.17}$$
$$\vdash_{\mathsf{d}\mathcal{L}} \psi \tag{5.18}$$

Note that

$$(\phi \leftrightarrow \psi) \to (\psi \to \phi)$$

has a proof in $\mathsf{d}\mathcal{L}$. From this fact and (5.17), it follows by the modus ponens proof rule that $\vdash_{\mathsf{d}\mathcal{L}} \psi \to \phi$. Applying modus ponens once more to this fact and (5.18) establishes that $\vdash_{\mathsf{d}\mathcal{L}} \phi$.

Case CT$_\sigma e$. Suppose that $\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} \text{CT}_\sigma e : \phi$. By Lemma 1,

$$\phi = \sigma(c(f(\bar{x})) = c(g(\bar{x})))$$

where

$$\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} e : \sigma(f(\bar{x}) = g(\bar{x})) \tag{5.19}$$

and σ is admissible for $f(\bar{x}) = g(\bar{x})$. Applying the inductive hypothesis to (5.19) establishes

$$\vdash_{\mathsf{d}\mathcal{L}} \sigma(f(\bar{x}) = g(\bar{x})) \tag{5.20}$$

Also by Lemma 1, σ is admissible on this formula. Therefore, [151, Theorem 25] establishes that the σ uniform substitution instance of CT is sound in $\mathsf{d}\mathcal{L}$ and so $\vdash_{\mathsf{d}\mathcal{L}} \sigma(c(f(\bar{x})) = c(g(\bar{x})))$ by the σ uniform substitution instance of CT.

Case CQ$_\sigma e$. Suppose that $\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} \text{CQ}_\sigma e : \phi$. By Lemma 1,

$$\phi = \sigma(p(f(\bar{x})) \leftrightarrow p(g(\bar{x})))$$

where

$$\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} e : \sigma(f(\bar{x}) = g(\bar{x})) \tag{5.21}$$

and σ is admissible for $f(\bar{x}) = g(\bar{x})$. Applying the inductive hypothesis to (5.21) establishes

$$\vdash_{\mathsf{d}\mathcal{L}} \sigma(f(\bar{x}) = g(\bar{x})) \tag{5.22}$$

Also by Lemma 1, σ is admissible on this formula. Therefore, [151, Theorem 25] establishes that the σ uniform substitution instance of CQ is sound in $\mathsf{d}\mathcal{L}$ and so $\vdash_{\mathsf{d}\mathcal{L}} \sigma(p(f(\bar{x})) \leftrightarrow p(g(\bar{x})))$ by the σ uniform substitution instance of CQ.

Case CE$_\sigma e$. Suppose that $\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} \text{CE}_\sigma e : \phi$. By Lemma 1,

$$\phi = \sigma(C(p(\bar{x})) \leftrightarrow C(q(\bar{x})))$$

where

$$\vdash_{\text{LP}_{\mathsf{d}\mathcal{L}}} e : \sigma(p(\bar{x}) \leftrightarrow q(\bar{x})) \tag{5.23}$$

and σ is admissible for $p(\bar{x}) \leftrightarrow q(\bar{x})$. Applying the inductive hypothesis to (5.23) establishes

$$\vdash_{\mathsf{d}\mathcal{L}} \sigma(p(\bar{x}) \leftrightarrow q(\bar{x})) \tag{5.24}$$

Also by Lemma 1, σ is admissible on this formula. Therefore, [151, Theorem 25] establishes that the σ uniform substitution instance of CE is sound in $\mathsf{d}\mathcal{L}$ and so $\vdash_{\mathsf{d}\mathcal{L}} \sigma(C(p(\bar{x})) \leftrightarrow C(q(\bar{x})))$ by the σ uniform substitution instance of CE.

Case σe**.** Suppose that $\vdash_{\mathrm{LP}_{\mathsf{d}\mathcal{L}}} \sigma e : \phi$. By Lemma 1, $\phi = \sigma(\phi')$ and $\vdash_{\mathrm{LP}_{\mathsf{d}\mathcal{L}}} e : \phi'$ for some ϕ'. Note that σ is, by Def. 8, an admissible substitution for ϕ'. The induction hypothesis for the smaller proof term e gives $\vdash_{\mathsf{d}\mathcal{L}} \phi'$. Therefore, $\vdash_{\mathsf{d}\mathcal{L}} \sigma(\phi')$ (i.e., ϕ) is provable by US.

Case $\mathscr{B}e$**.** Similar to the case for σe. □

The fact that $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ is sound with respect to the semantics of $\mathsf{d}\mathcal{L}$ under proof term erasure is a corollary of this theorem.

Corollary 1 (Validity of Evident Formulas)**.** *If* $\vdash_{LP_{\mathsf{d}\mathcal{L}}} e : \phi$ *then* $[\![\phi]\!]^I_{\mathsf{d}\mathcal{L}} = S$ *where* S *is the set of all states.*

Proof. By Theorem 2, $\vdash_{\mathrm{LP}_{\mathsf{d}\mathcal{L}}} e : \phi$ implies $\vdash_{\mathsf{d}\mathcal{L}} \phi$. Note that $\mathsf{d}\mathcal{L}$ is sound,[151, Theorem 25] so $[\![\phi]\!]^I_{\mathsf{d}\mathcal{L}} = S$. By Def. 8, $[\![\phi]\!]^I_{\mathrm{LP}_{\mathsf{d}\mathcal{L}}} = [\![\phi]\!]^I_{\mathsf{d}\mathcal{L}} = S$. □

5.5 Checking Proof Terms Using Truth-Preserving Transformations

The soundness-critical core of KeYmaera X contains a set of truth-preserving operations on $\mathsf{d}\mathcal{L}$ formulas; these operations correspond to the axioms and proof rules of $\mathsf{d}\mathcal{L}$. Provable objects are the closest that KeYmaera X comes to proof certificates. A Provable is an object with a goal and a sequence of remaining subgoals, each of which is a sequent. A KeYmaera X proof certificate for a formula φ is a Provable object with no remaining subgoals and $\vdash \varphi$ as its goal. Provable objects may only be created by the soundness-critical core of KeYmaera X, so they are guaranteed to be constructed via a sequence of truth-preserving operations such as proof rules, axioms, or substitutions. However, a proof certificate does not record the actual sequence of truth-preserving operations through which it is produced. While memory-efficient, this state of affairs is less than ideal for reasons that were enumerated in the introduction.

Adding proof terms to KeYmaera X is relatively simple because $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ is in every way – syntactically, semantically, and axiomatically – parsimonious with $\mathsf{d}\mathcal{L}$. Never-the-less, more pragmatic implementation-focused concerns (e.g., interfacing with other tools and producing readable proof transformations) have obsoleted the use of $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ in KeYmaera X. None-the-less, we briefly describe some of the implementation details that are necessary for bridging the gap between theory and implementation for $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$.

The proof of Theorem 2 was written so that it suggests a procedure for proof term checking. The proof could have exploited completeness results at several points. Instead, we opted for explicitly constructing a syntactic $\mathsf{d}\mathcal{L}$ proof. For this reason, an $\mathrm{LP}_{\mathsf{d}\mathcal{L}}$ proof term checker can follow the structure of the proof of Theorem 2 – for each component of a proof term, the proof term checker constructs the sequence of truth-preserving operations described in the proof of Theorem 2. These truth-preserving operations are then executed by the KeYmaera X core. If each operation succeeds (e.g., no clashes occur during uniform substitutions), then the proof term checker returns true.

There are a few caveats. The inversion lemma relies on the existence of certain formulas; these formulas must be inferred automatically, or else proof terms must be augmented with additional annotations. In particular, inferring how the types of implications and substitutions decom-

pose is not completely trivial and is reminiscent of the type inference problem in typed functional languages. Additionally, in the proof of Theorem 2, there are some points where the truth of a particular theorem is asserting (e.g., via soundness). In each of these cases, KeYmaera X has either a tactic or an extra proof rule that provides exactly the required truth-preserving transformation. For example, the `keymaerax.TacticLibrary.AndR` tactic of KeYmaera X performs the action of the AndR schema referenced in the $e \wedge d$ case. The σ instances of CT, CQ, and CE (which are guaranteed to be sound by [151, Theorem 25]) that we appeal to in the $\text{CT}_\sigma e$, $\text{CQ}_\sigma e$, and $\text{CE}_\sigma e$ cases also have corresponding tactics in KeYmaera X.

5.6 Conclusion

Explicit notions of evidence provide a clean separation between proof checking and proof search and enable analyses that crucially depend upon an interrogation of the structure of proofs. The Logic of Proofs for Differential Dynamic Logic demonstrates that it is possible to construct a calculus of proof terms on top of an existing theorem prover. The broader implications of explicit proof terms for dynamic logics remains largely unexplored, especially in the context of AI for theorem proving. As discussed in the introduction, the second half of this thesis establishes a clear trajectory toward building systems that leverage not just theorems but also proofs of those theorems to control safely.

Part II

Verifiably Safe Learning

Chapter 6

An Introduction to Safe Learning

The formal methods discussed in Part I of this dissertation increase our confidence in safe system operation by providing highly trustworthy safety proofs for cyber-physical systems. Hybrid systems reachability proofs substantially increase our confidence in overall system safety by eliminating the possibility that our control software contains bugs. However, safety theorems are always stated with respect to a model of the underlying environment and typically consider highly nondeterministic control software. Verified controllers do not explain how to achieve high-level objectives other than remaining safe and, furthermore, do not provide any guarantee that modeling assumptions accurately represent reality.

The second part of this dissertation addresses these two shortcomings of the purely verificationist approach introduced in Part I. The techniques introduced in this chapter address two deeply related questions.

1. How do we obtain formal guarantees about controllers obtained via reinforcement learning?
2. How can we increase overall confidence in safe system operation even when the initial environmental model is incorrect?

Chapter 7 addresses the first question by explaining how to translate verification results for *nondeterministic* control software into verification results for *deterministic* control software obtained via reinforcement learning. Our approach, called Justified Speculative Control (JSC) also explains how to leverage insights from logical analysis of dynamical systems during the learning process. Chapter 8 introduces a falsification-based approach both for answering the second question in the special case where we have a set of possible models only one of which is correct. Finally, Chapter 9 closes the loop by explaining how to obtain safety proofs for a highly optimized controller starting with nothing except experimental data and a global safety specification. Along the way, we will see that combining model-based deductive verification with inductive model building provides a powerful paradigm for obtaining highly trustworthy control software even in situations where models are difficult or impossible to build at design time.

The remainder of this background chapter is organized into two sections. Section 6.1 discusses safe reinforcement learning, and Section 6.3 discusses three approaches toward controlling well without a perfect model of the environment.

6.1 Reinforcement Learning

We begin by recalling the mathematical preliminaries of Reinforcement Learning and discussing related work on safe reinforcement learning. Reinforcement learning algorithms solve the problem of controlling well even when some state transitions are not deterministic. This random transition structure is formalized in terms of Markov Decision Processes.

Definition 10. *A Markov Decision Process (MDP) is a tuple* (S, A, P, R) *where:*

- S *is a set of states,*
- A *is a set of actions,*
- $P(s_{pre}, a, s_{post}) : S \times A \times S \to \mathbb{R}$ *is the probability of transiting from* s_{pre} *to* s_{post} *when taking action* a*, and*
- $R(s_{pre}, a, s_{post}) : S \times A \times S \to \mathbb{R}$ *is the reward obtained by transitioning from state* s_{pre} *to state* s_{post} *via action* a*.*

Unless otherwise noted, we will assume that $\sum_{s_{post} \in S} P(s_{pre}, a, s_{post}) = 1$.

The goal of reinforcement learning is to discover a policy $\pi : S \to A$ that maximizes the long-term cumulative reward. The sequence of states visited and actions taken by the policy is referred to as the system's trajectory. The goal is to maximize the sum of rewards (as computed by R) expected over a trajectory.

This dissertation focuses on reinforcement learning algorithms that satisfy safety constraints stated in $\mathsf{d}\mathcal{L}$. for this reason, we introduce an abstraction over MDPs that interfaces well with the semantics of $\mathsf{d}\mathcal{L}$ and that translates probabilistic transitions into nondeterministic transitions.

Definition 11. *A reinforcement learning model is defined for a Markov Decision Process* (S, A, P, R) *whenever* S *is a mapping from a set of variables* V *into* $\mathbb{R}$ *by replacing the probabalistic transition function* P *with a reachability relation* E *so that* $s_{post} \in E(s_{pre}, a)$[1] *whenever* $P(s_{pre}, a, s_{post}) \neq 0$.

Standard reinforcement learning is inappropriate in safety-critical settings for two reasons. First, finding an optimal policy requires entering unsafe states and observing negative rewards. This problem is particularly acute in systems that use online reinforcement learning to discover policies directly on a hardware platform instead of in simulation. Second, there is no guarantee that a final policy avoids unsafe states because even optimal policies may enter unsafe states if the reward function is not carefully defined. These safety violations are particularly egregious in settings where global safety constraints are known a priori and are easily expressible, which is often the case in control problems.

García and Fernández survey several approaches toward safe learning [69]. García and Fernández define Safe RL as *"the process of learning policies that maximize the expectation of the return in problems in which it is important to ensure reasonable system performance and/or respect safety constraints during the learning and/or deployment processes"* .

This dissertation is concerned with *Verifiably Safe RL* which we define as the problem of obtaining or leveraging formal proofs of correctness for safe reinforcement learning algorithms.

Obtaining verification results for algorithms and techniques developed in the Artificial Intelligence research community is an emerging area of interest [161], but there is a rich history of

[1] Or $E(s_{\mathtt{pre}}, a) = s_{\mathtt{post}}$ for deterministic systems

research on safe control in the absence of perfect models. This chapter reviews how the work in Part II of this dissertation is related to prior work on safe reinforcement learning.

6.2 Safe Reinforcement Learning

Traditional research on safe reinforcement learning did not take advantage of formal verification. An excellent and comprehensive survey on safe learning by García and Fernández [69] decomposed these approaches into two broad categories: methods that obtain safety assurances by modifying the optimality criterion, and methods that modify the exploration process. In this section we review this work following the blueprint laid out by García and Fernández. Along the way, we extend their survey to take into account recent work on both safe learning and on *verifiably* safe learning.

6.2.1 Modifying the Criterion

The behavior and final output of a reinforcement learning algorithm is highly dependent upon the definition of an optimization criterion, or reward function, which defines the objective of the reinforcement learning algorithm by assigning quantitative values to states and/or actions. In the constrained criterion paradigm, safety properties[2] of safe reinforcement learning are achieved by modifying or transforming this criterion in a way that pushes the learning algorithm toward safe actions. García and Fernández survey these approaches [69]. We recall three of the main approaches toward modifying the optimization criterion discussed by García and Fernández and additionally discuss recent work which post-dates their survey that focuses on leveraging theorem provers for constraining reinforcement learning. We then conclude our discussion of safety through criteria modifications with a thorough comparison of these approaches to the work discussed in Part II of this document.

Worst Case Criterion

One extensively studied approach toward modifying the optimality criterion defines the optimality criterion in terms maximizing the worst-case outcome. This approach, first introduced by Heger [84] and later extended by Coraluppi et al. [39, 40], defines the objective function in terms of maximizing worst-case outcomes.

Given a set of trajectories Ω^π of alternative states and actions that occurs under policy π, where $E_{\pi,\omega}(\cdot)$ is the expectation with respect to the policy π and trajectory ω, and where r_t is the reward observed at time t, the objective of worst-case control is to find the policy $\pi \in \Pi$ with the maximal minimum outcome:

$$\max_{\pi\in\Pi} \min_{\omega\in\Omega^\pi} E_{\pi,\omega}(R) = \max_{\pi\in\Pi} \min_{\omega\in\Omega^\pi} E_{\pi,\omega}\big(\sum_{t=0}^{\infty} \gamma^t r_t\big)$$

[2] A safety property is more commonly referred to as a *specification* in the formal methods community. In the context of a dL formula $\psi \rightarrow [\alpha]\varphi$, the term safety property as used by much of the safe RL community refers only to the subformula φ. We adopt this meaning for the term safety property throughout the rest of this chapter in order to avoid confusion and discuss prior work on safe RL in its own terms.

Heger introduced an algorithm analogous to Q-learning, called $\hat{Q}$-learning, which optimizes for this minimax criterion [84].

The worse-case criterion family of objective criteria has the advantage of reducing risk in worst-case scenarios, but at the cost of extremely conservative behavior even when there are good (e.g., model-based) reasons to assert that some of those worst case outcomes are impossible given the current configuration of the system. Bagnell et al. observed that the mini-max formulation given above is closely related to work on H_∞ robust control [15]. Chris Gaskett demonstrated that this pessimism can be harmful and proposed an approach toward compensating for this pessimism by introducing a parameter that is used to choose between the classical Q-learning algorithm and Heger's $\hat{Q}$-learning algorithm [71].

Controlling Safely with Uncertainty in Parameters

Robust MDPs [138] and their associated reinforcement learning algorithms [124] provide an alternative formulation of the problem of learning safely. Instead of controlling safely and optimally with respect to a single MDP, the robust MDP framework studies MDPs in which transition probabilities are uncertain and/or estimated from data. The theory of robust Markov decision processes places special emphasis on the problem of controlling well given an explicit understanding of model uncertainty.

Bagnell et al. also address the problem of controlling well under model uncertainty by considering the problem of finding a policy that is optimal with respect to a class of models, characterized in terms of a set of possible transition matrices [15]. They show that when this set is finite and convex, computing a best policy with respect to the worst possible model is NP-hard but tractable in practice.

Constrained Criterion

Worst-case criterion and robust MDPs both approach safe control by modifying the problem setup or optimization criterion to take into account possible worst-case outcomes. An alternative approach, which also focuses on the optimization criterion, instead obtains safe policies by only allowing agents to choose from a set of control actions that are conjectured – but not formally proven – to be safe.

Altman's Constrained Markov Decision Processes provide a theoretical framework for characterizing constrained optimization of dynamical systems [6]. Geibel et al. introduced a reinforcement learning algorithm for MDPs with constraints [72]. Recent work by Achiam et al. [2] leverages this framework to constrain learning for high-dimensional control problems.

Kadota et al. consider the case of MDPs with multiple utility constraints, in which the goal is to maximize expected utility for one of the functions while maintaining lower bounds on the expectations for other utility functions [104]. Kadota et al. give a saddle-point theorem that establishes conditions on the existence of these optimal policies. Their work is relevant to safe RL because in some settings safety constraints can be characterized in terms of lower-bounds on utility functions.

Trust region policy optimization is a policy optimization algorithm that tends to learn robust policies via monotonic improvement in policy performance [163]. Unlike Shulman et al. and

Achiam et al. who focus on model-free reinforcement learning, we start with model-based reinforcement learning and account for the difficulty of building accurate models by explaining several ways in which formal verification can help cope with model deviation.

Logical Constraints on Learning

The difference between safe RL and verifiably safe RL is analogous to the difference between a correct piece of software and a formally verified piece of software. In theory it is possible to implement a correct software system without the use of formal methods; however, in practice, formal verification provides the only known technique for obtaining highly trustworthy proofs about large software systems. Whereas approaches toward Safe RL *assume* the presence of correct constraints, verifiably safe RL backs those constraints with formal proofs that the constraints are sufficient for establishing global safety guarantees.

Each of the approaches discussed above addresses the safe RL question but not the verifiably safe RL question. Several recent approaches toward safe learning move toward verifiably safe RL by leveraging existing theorem provers and logics; our work very much fits in this vein.

Alshiekh et al. and Hasanbeig et al. each propose an approach toward logically constraining reinforcement learning based on Linear Temporal Logic (LTL) [5, 83]. Like the Justified Speculative Control (JSC) algorithm discussed in Chapter 7, these approaches have a logical foundation. Unlike JSC, these approaches do not use a logic capable of expressing hybrid dynamical systems and do not explain what to do when a model deviation is detected. Therefore, these approaches do not solve the problem this thesis proposes addressing: providing verifiably safety guarantees for *cyber-physical systems* when reality deviates from modeling assumptions.

The use of LTL limits the applicability of these approaches in cyber-physical systems, but does succinctly capture many constraints on discrete or discretized planning and optimization problems. One fruitful avenue of future work could combine these approaches with JSC, using LTL-based approaches for expressing constraints on global, coarse-grained planning while using $\mathsf{d}\mathcal{L}$ for expressing constraints on more local, fine-grained control decisions.

By contrast, KeYmaera X and ModelPlex [64, 133] are able to automatically and correctly reduce reachability properties of hybrid dynamical systems to formulas of real arithmetic. Relative to these constrained criterion approaches, we introduce a methodology – based on hybrid systems theorem proving – for correctly and often automatically generating useful optimizing constraints from statements about reachability properties of hybrid dynamical systems.

Comparison with Our Work

Part II of this thesis contributes three new approaches toward safe learning: justified speculative control (Chapter 7), model update learning (Chapter 8), and hybrid program synthesis (Chapter 9). We now discuss what is novel about each of these contributions relative to the prior work on modifying criterion discussed above.

Justified speculative control (JSC) is a new approach toward safe RL that combines a constrained criterion with a modified criterion by leveraging theorem proving.

Relative to constrained criterion approaches [2, 6, 72], JSC provides a strong justification that monitoring constraints are well-founded and correspond to intuitive safety properties for

analyzable dynamical models. This is an important contribution because constrained criterion approaches require the system designer to explicitly state safety constraints, usually as purely arithmetic properties that do not mention e.g., differential equations or discrete dynamics. However, the problem of coming up with good constraints is analogous to the problem of coming up with good reward functions. Although it is often easy to state the dynamical systems interpretation of a constraint (e.g., the robot does not run into the interlocutor whose movement is governed by given ODEs), translating these dynamical descriptions into a arithmetic descriptions that are useful as costs or constraints for an optimization algorithm is non-trivial and error prone. The problem of reducing reachability properties about dynamical systems to unquantified real arithmetic constraints is ultimately reducible to the problem of full-blown hybrid systems theorem proving which is undecidable[87]; in fact, this is exactly the methodology that KeYmaera X takes toward verification.

Relative to work on worst-case criterion and robust MDPs [15, 39, 40, 84, 124, 138], JSC distinguishes between situations where the model is accurate and situations where the model is inaccurate. Furthermore, instead of optimizing for the worst case or controlling well with respect to an uncertainty set, JSCQ leverages the model itself as a reward signal when model deviation occurs. Instead of optimizing for worst-case scenarios or attempting to control robustly with respect to a class of models, JSCQ instead optimizes for correcting mismatches between its model and reality. Our approach avoids any need to explicitly model uncertainty bounds in the model and also avoids overly conservative bounds in situations where a single, known, non-worst-case model is in fact accurate.

Our extensions to JSCQ in Chapter 8 and Chapter 9 attempt to learn a more accurate model which is then used to constrain the learning process Chapter 8 and Chapter 9 take up the problem of extending JSCQ to multiple models; the approaches discussed in these chapters are closely related to the stationary uncertainty formulation of the robust control problem discussed in [138, 139], where uncertainty is fixed once and for all for a given control policy.

JSC improved on previous work in the ways following.

- Unlike [2, 6, 15, 39, 40, 72, 84, 124, 138] all of our approaches transfer guarantees obtained from computer-checked safety proofs to reinforcement learning algorithms, closing the gap between constraints/rewards/uncertainty sets used for safe RL and the underlying kinematic models that those constraints are based upon.
- Unlike [15, 39, 40, 84, 124, 138], we distinguish between situations where an accurate model is available and situations where an accurate model is not available, thereby avoiding worst-case decision making except when necessary. Also unlike these approaches, we demonstrate how to use models as utility functions for repairing model deviations, thereby avoiding the need to explicitly model uncertainty in the form of uncertainty sets used in the robust MDP setting [138, 139].
- Unlike [2, 6, 72], we consider the problem of controlling well even when empirical observations at runtime invalidate models that constraints are based on.
- Unlike [5, 83], we do all of this in the setting of an expressive logic for reasoning about the safety of cyber-physical systems, which are important for the reasons discussed in Part I.

Because μlearning and our approach toward hybrid program synthesis both build upon JSC, these comparisons extend to the work discussed in Chapter 8 and Chapter 9 as well. Additionally,

our approach toward hybrid program synthesis can be thought of as addressing one of the robust control problems laid out in [139] but, again, with the crucial distinction that we provide formal proofs of correctness to justify our claims of safe control.

6.2.2 Initial Knowledge Approaches

Another approach toward safe learning attempts to initialize the learner in order to direct policy exploration away from unsafe states [49]. Our approach is analogous; the guards on control decisions in our hybrid programs are a form of initial knowledge. Unlike most approaches that leverage initial knowledge, we explicitly characterize the difference between states where our initial knowledge is trustworthy from states where our initial knowledge is not trustworthy. This is important because often the correctness of control policies is not trivial.

6.2.3 Analysis of Learned Policies

So far, our discussion of safe reinforcement learning has focused on methods of modifying the definition/behavior of the learning process. Modified criterion and initial knowledge approaches ensure safety by modifying the learning/optimization process.

An alternative approach toward safe reinforcement learning suggests analyzing the policies after they are constructed from a learning process. We now turn our attention to this class of methods, which all focus on the *output* of a reinforcement learning algorithm.

In the simplest case, a tabular policy extracted after performing a tabular Q-learning algorithm could be analyzed for safety from each table entry. Obviously, this approach is completely infeasible for the same reason that tabular learning on large or continuous state spaces is intractable: the set of states and actions is far too large to check each one for safety. For this reason, both RL and safe RL that analyze learned policies instead focus on representations such as neural nets.

Katz et al. introduce an SMT-based approach for analyzing deep neural networks and apply this technique to analysis of a DNN implementation of parts of a collision avoidance for aircraft [105]. Wang et al. take a similar approach toward reasoning about security properties [170].

Other examples of related work on analysis of learned policies focus on perception rather the control. Examples include the predictor/verifier framework of Dvijotham et al. [51], and VeriVis by Pei et al. [144], and a growing body of work on both robustness metrics for neural networks and applications of SMT solvers to verification of neural networks. These approaches focus on perception instead of control, but also focus on analysis of a learned classifier.

Analyzing learned models is attractive when the modifying the learning process is intractable or impossible, which is often the case in practical settings where a cutting-edge algorithm is not trivial to modify with a constrained criterion. The approach is especially relevant in poor engineering environments where safety analysis has been treated as a post-hoc concern.

Although recent successes demonstrate the feasibility of analyzing very large learned policies, the curse of dimensionality tells us that these approaches will always require clever optimization. The approaches discussed in this thesis leverage the insight that safety problems are often of lower dimension than optimization problems; many variables relevant to overall fitness (e.g., fuel efficiency, passenger comfort, etc.) are not relevant to the safety-critical concerns

about the system. Furthermore, unlike our approach, analyzing learned policies does not provide guarantees about the learning process itself.

6.2.4 Summary of Related Work on Safe Learning

Our work on Justified Speculative Control, introduced in Chapter 7, makes two contributions relative to prior work on safe learning. First, we leverage hybrid systems verification results and runtime monitor synthesis to appropriately sandbox the exploration process, instead of relying on more ad-hoc sources of knowledge about how to act safely. The chain of evidence transfers from a high-level model to runtime monitors and ultimately to the reinforcement learning process via the theorems presented in this thesis. Second, we distinguish between *optimizing among known safe policy options* and *speculating about portions of the state space that are not a priori modeled*. This distinction is crucial to determine what level of speculation should be allowed, and when.

When compared to existing approaches to reinforcement learning, our approach either 1) suggests a way to strengthen the existing approach by incorporating not just a presumably safe policy but a *formally verified* safe policy[3]; or 2) is compositional with the existing approach (by further modifying our exploration process to perform more robust decision making when the model monitor is already violated).

6.3 Viewpoints on Controlling Without a Perfect Model

Automatically modifying or constructing programs based upon logical specifications, test data, and/or environmental data is a common approach used by researchers in many different disciplines. This section reviews approaches from the control theory, software engineering, AI/ML, and programming languages research communities.

Unlike existing work, the verification-preserving model update algorithms discussed in Chapter 9 learn how to modify a *continuous* system in response to data, discover a *corresponding* update to a *discrete* system that preserves a hybrid reachability property, and then synthesize a formal and computer-checked proof that the resulting combination of discrete/continuous model updates continue to satisfy relevant safety invariants.

6.3.1 Model and System Identification

System identification algorithms use data to build a discrete, continuous, or even hybrid dynamical systems model of the observed process. System identification is an enormous and mature area of research. Diester provides a historical overview of system identification [48], Garg et al. survey approaches toward system identification for control that decomposes techniques across several different dimensions [70], and Juloski et al. survey and compare approaches toward system identification for certain classes of hybrid systems [103].

[3]The difference between safe reinforcement learning and verifiably safe reinforcement learning is analogous to the difference between unverified software and verified software – namely, the existence of a formal specification that precisely characterizes the meaning of safety and a proof justifying correctness these safety claims.

A thorough survey of system identificaiton techniques is beyond the scope of this chapter because our work is largely compositional with system identification procedures; we focus on the problem of maintaining formal safety guarantees when incorporating identification into safety-critical control. When viewed from the perspective of system identification, Chapter 8 considers the problem of verifiably safe model selection and Chapter 9 discusses how to combine classical model identification approaches with program synthesis

6.3.2 Program Synthesis and Repair

Program synthesis algorithms attempt to automatically generate programs in a fixed programming language. The basis for synthesis might be a formal specification, a test suite, or a set of I/O pairs. Automatic program repair algorithms are a special class of synthesis algorithms that attempt to synthesize bug-fixing patches for existing programs. Approaches toward program repair differ on the basis of the programming language under consideration, the domain of relevance, the method for producing patch candidates, and the basis for accepting or rejecting a repair.

GenProg by Le Goues et al. [121] uses genetic programming to generate repairs for C programs using test cases as the primary definition of correctness. Rothenberg and Grumberg [159] leverage SAT and SMT solvers to generate repairs using logical assertions as a basis for correctness. Le et al. [120] leverage a combination of deductive verification and genetic programming to generate repairs.

Our proposed approach leverages data-driven approaches to decide how environmental models should change. Especially for autonomous systems, program repairs based upon designer intent are irrelevant to the choice of an accurate environmental model because environmental models capture physical realities rather than designer intents. However, once a repair to the environmental model is identified, we leverage logic-driven approaches to decide how a controller should change in response to identified changes in environmental models. This combination that allows us to adapt to unforeseen environmental behaviors without sacrificing reachability proofs or losing track of designer intent.

When viewed from the perspective of program synthesis and repair, the proposed work

1. contributes the first library of mutations for continuous programs (ODEs) and hybrid programs,
2. demonstrates how to combine data-oriented mutations for environmental models with logically constrained mutations for controller models in a way that preserves reachability properties for the combined hybrid dynamical system, and
3. shows how program repairs provide a setting for characterizing situations in which learning algorithms are capable of operating safely.

Programming Language Representations in Reinforcement Learning Recent work leverages programming language theory as part of a reinforcement learning algorithm. For example, Verma et al. use a simple functional programming language to ensure intrepretability of learned policies [169]. Our work has a similar goal, but focuses on a language with a rich combination of discrete and continuous dynamics. Like other related work on using LTL specifications to constrain reinforcement learning, the approach proposed by Verma et al. might compose well with

our current and proposed work by providing an attractive setting for planning problems while dℒ-based approaches provide an attractive setting for problems where interpretability is fundamentally limited by the lack of differential equations in the policy language (i.e., most controls problems).

Ghosh et al. [73] propose an approach toward constrained optimization that monitors for safety violations using a specification monitor and uses a grammatical approach toward bounding model deviation. Unlike the proposed work, Ghosh et al. do not consider the case where the grammar can express not just arithmetic constraints but entire hybrid dynamical systems and also do not consider the problem of preserving formal proofs.

Other Related Work on Hybrid Systems We propose a data-driven approach toward safe control, mediated by model identification algorithms specialized to preserve verification results. Many other data-driven approaches toward hybrid systems control are suggested in the literature; for example, Kushner et al. take a data-driven approach toward control for an artificial pancreas [117] and Althoff et al. suggest a demonstration-driven approach [75]. Kumar et al. introduce an approach toward learning based upon Hamiltonian control [113] inspired by earlier work by Nerode and Kohn [108].

Our proposed approach uses (online mutations of) specifications to monitor cyber-physical systems. Our monitors are generated by KeYmaera X using Modelplex; Bartocci et al. review various other approaches toward monitoring CPS [18], none of which construct proofs linking monitors to models. Some of our proposed model updates make use of both offline and online verification; Johnson et al. suggest other approaches toward the online component of this verification effort [102]. Some (but not all) of our proposed model updates are refinements of hybrid programs; we could show these are verification-preserving using Loos' differential refinement logic [127].

6.4 Conclusion

The second part of this thesis demonstrates how to leverage the logical model-based verification approaches discussed in Part I to obtain safety guarantees for learned control policies. Chapter 7 explains how to transfer model-based verification results to policies obtained via reinforcement learning and also explains how verification guarantees can be used to direct learning when modeling errors are discovered at runtime. Chapter 8 uses runtime model falsification to extend the approach discussed in Chapter 7 to scenarios in which there are multiple possible models of the world. Finally, Chapter 9 considers the problem of learning both the model and the controller from data.

Perhaps the most important insight from this work is that logic has much more to offer than mere sandboxing. Programming languages and their verification logics provide a promising semantic target for explainable and verifiable machine learning.

6.5 Overview of Related Work

The proposed thesis leverages several research areas: hybrid systems verification (HSV), reinforcement learning/optimal control (RL/OC) constrained reinforcement learning (CRL), model/system identification (MI/SI), and program synthesis/repair (PS&R). Table 6.1 summarizes all of these research areas as they relate to our previous and proposed work. The Safe columns indicate whether the technique guarantees safety in model space ($\in$ MS) and outside of model space ($\notin$ MS); for techniques that have multiple phases (e.g., reinforcement learning), safety is taken to mean safety throughout system execution. The HS column indicates whether the family of techniques is applicable to hybrid systems. The "Changes Model" column indicates whether the family of techniques will produce a human-readable model explaining why the system behaved as it did (either inside or outside of model space). The "Explainable" column indicates whether the approach offers some other method for ensuring that safety guarantees are explainable to system designers and other stakeholders.

Table 6.1: Summary of Related Research Fields.

Approach	**Safe $\in$ MS**	**Safe $\notin$ MS**	**HS**	Changes Model	Explainable
JSC+VPMU[4]	Formal	Formal	Yes	Yes	Yes
JSC[62]	Formal	Informal	Yes	No	Yes
Software Verification	Formal	No	No	No	Yes
HS Verification[5]	Formal	No	Yes	No	Yes
PS&R w/ formal specs[6]	Some	Some	No	Yes (not HS)	Yes
PS&R w/ test cases	No	No	No	Yes (not HS)	Yes
MI/SI[7]	Some (not formal)	Some (not formal)	Some	Some	Some
RL/OC[8]	No	No	Some	No	Some
Constrained RL/OC	Informal	No	Some	No	Some
LTL FM for RL[5, 83]	Formal	No	No	No	Yes
MC for NNs (e.g., [105, 170])	Formal	No	Some	No	Yes

[4]Proposed.
[5]A more in-depth comparison of software and hybrid systems verification techniques is presented in Table 3.1.
[6]Section 6.3.2 compares JSC+VPMU to program synthesis and repair techniques in more detail.
[7]Section 6.3.1 discusses the relationship between JSC+VPMU and model/system identification in more detail.
[8]Section 6.2 discusses the relationship between JSC and (constrained) learning/optimization in more detail.

Chapter 7

Justified Speculative Control

Autonomous vehicles should be deterministic, efficient, and safe. Part I introduced our approach toward for ensuring safety for nondeterministic controllers. Verified hybrid programs distinguish between safe and unsafe actions in each state, but do not single out the specific action that should be taken in order to achieve an objective other than safety. The car in Example 2 has two different options in most of the state space, and can *always* choose to activate its brakes. Although KeYmaera X tells us that this controller is safe, the resulting theorem does not explain the sequence of actions that will help the car actually *reach* the stop sign.

KeYmaera X establishes a safe set of actions, but does not explain which of these actions ought to be taken.

Fortunately, reinforcement learning solves exactly the problem that KeYmaera X does not solve. Given an appropriate reward signal, a reinforcement learning algorithm could tell us which sequence of actions should be taken so that the car reaches the stop sign without over-shooting the stop sign. However, learning this control policy might require thousands of iterations before the algorithm eventually learns how to avoid over-shooting the stop sign. This situation is unacceptable in the real world, where over-shooting a stop sign might result in property damage or even loss of life. In addition, the exploration of obviously unsafe or unfeasible policies also contributes to the notorious data-inefficiency of reinforcement learning algorithms.

This chapter introduces an approach that combines the best of both learning and verification[1]. In our approach, KeYmaera X constrains the search space for a reinforcement learning algorithm so that only safe actions are taken, while reinforcement learning is free to search the subset of safe policies for a policy that achieves objectives other than safety. Our approach, called justified speculative control (JSC), ensures that verification results transfer to policies learned via reinforcement learning. This approach has the nice auxiliary advantage of increasing the data efficiency of reinforcement learning, because obviously unsafe actions do not need to be explored during reinforcement learning.

KeYmaera X, like all formal methods tools, can only provide guarantees relative to a model of the world. When the model is inaccurate, guarantees disappear. For example, if the stop sign in Example 2 begins to move forward[2], then the car has left the modeled portion of the state space

[1] This chapter is based on the paper *Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning* published by Fulton and Platzer [62].

[2] E.g., consider a situation in which the stop sign is held by a partially occluded construction worker along a work

and might over-shoot the stop sign even though its controller is verified. We call the set of states where the model is accurate the *model space*; traditional verification results only apply within their model space.

Designing a safe autonomous system requires either building a perfect model of the world, or else ensuring that the system will behave well outside of model space.

7.1 Runtime Monitoring for dℒ

Central to our own approach toward safe RL is the ability to check, at runtime, whether or not the current state of the system can be explained by a dℒ formula. The KeYmaera X theorem prover provides a mechanism for translating a dℒ formula of the form $P \rightarrow [\alpha^*]Q$ into a formula of real arithmetic, which checks whether the present behavior of a system fits to this model. The resulting arithmetic is checked at runtime and is accompanied by a correctness proof. This algorithm, called ModelPlex [135], can be used to extract provably correct monitors that check compliance with the model as well as with the controller. If non-equivalence transformations have been used in the ModelPlex monitor synthesis proofs, the resulting monitor may be conservative, i.e. give false alarms. But if the monitor formula evaluates to true at runtime, the execution is guaranteed to be safe.

Controller Monitors

ModelPlex controller monitors are boolean functions that monitor whether or not the controller portion of a hybrid systems model has been violated. The monitor takes two inputs – a "pre" state and a "post" state. The controller monitor returns true if and only if there is an execution of the $ctrl$ fragment of the program that, when executed on the "pre" state, produces the "post" state. For example, the controller monitor for Example 1 is:

$$(v_{post} = v \land p_{post} = p \land a_{post} = A) \lor$$

$$(v_{post} = v \land p_{post} = p \land a_{post} = 0)$$

where a_{post} is the value of a chosen by the controller. Similarly, v_{post} and p_{post} are the values v and p chosen by the controller. Therefore, this controller monitor states that the controller may choose $a := A$ or $a := 0$, but may not change the values of p or v.

We write the controller monitor as a function

$$\mathit{CM} : S \times A \rightarrow \mathit{Bool}$$

mapping a current state $s \in S$ and an action $\mathit{act} \in A$ to a boolean value. This formulation is equivalent to the pre/post state formulation (e.g., $v_{post} = \mathit{act}(v_{pre})$) where

$$\mathit{act}(s)$$

is the state reached by performing the action *act* in state s.

zone and the construction worker begins to walk forward.

Model Monitors

ModelPlex can also produce full model monitors, which check that the *entire* system model is accurate – including the model of the system's physics – for a single control loop. The full model monitor returns true only if the controller for the system chooses a control action that is allowed by the model of the system and also the observed physics of the system correspond to the differential equations describing the system's physical dynamics. The full model monitor for Example 2 is:

$$(t_{post} \geq 0 \ \wedge \ a_{post} = A \ \wedge \ v_{post} = At_{post} + v_{pre} \wedge$$

$$p_{post} = \frac{At_{post}^2}{2} + v_{pre}t_{post} + p_{pre}) \vee$$

$$(t_{post} \geq 0 \wedge v_{post} = v_{pre} \wedge p_{post} = v_{post}t_{post} + p_{pre} \wedge a_{post} = 0)$$

Each side of the disjunction corresponds to a control decision, and the constraints on v and p come from solving the differential equation $p' = v, v' = a$.

We write the ModelPlex monitor as a function

$$\mathit{MM} : S \times A \times S \to \mathit{Bool}$$

where S is a set of states and A is a set of actions allowed by the controller; the first argument is the state before the control action, the second argument specifies the control action, and the third argument specifies the state after following the plant with the chosen control action.

The ability to perform verified runtime monitoring is essential to our approach – these arithmetic expressions are the conditions that allow us to determine when to use a speculative controller, and when to avoid deviating from the various options available in the verified nondeterministic control policy. We define model monitors semantically.

This section explains how justified speculative control guarantees safety within model space, discusses one approach toward controlling well outside of model space, and discusses other approaches toward safe and verifiable reinforcement learning. We also discuss the limitations of this approach; those limitations will motivate the work discussed in Chapter 8 and Chapter 9.

Justified Speculative Control extends model-based safety theorems to policies obtained through reinforcement learning. Given a verified model $\mathit{init} \to [\{ctrl; plant\}^*]\mathit{safe}$, if controller monitor $\mathit{CM}(s, \mathit{act}) =$ True then

$$(s, \mathit{act}(s)) \in [\![\mathit{ctrl}]\!]$$

and if model monitor $\mathit{MM}(s_{pre}, \mathit{act}, s_{post}) =$ True then

$$(\mathit{act}(s_{pre}), s_{post}) \in [\![\mathit{plant}]\!].$$

In this chapter, we use model monitors to transfer a safety proof of the above form into guarantees about a reinforcement learning process acting in domains where the environment never results in a model monitor returning false.

7.2 The Justified Speculative Learning Algorithm

The listing below presents a generic reinforcement learning algorithm with justified speculation. This algorithm corresponds to the approach described in the introduction – the system chooses among a set of verified safe actions whenever the environment is accurately modeled, and otherwise selects any action in the action space.

The inputs are a reinforcement learning model (S, A, R, E), a strategy $choose$ for selecting actions, a function $update$ that records state transitions, and a predicate $done$ over states indicating which states are terminal. Each of these functions has access to the learning model (S, A, R, E) and, typically, some additional state (e.g., a Q table, policy/value function approximator, etc.).

The Justified Speculative Control algorithm leverages the ModelPlex runtime monitors to ensure that, whenever the system is accurately modeled, only safe actions are taken. The model monitor is used to determine if the previously observed state, previous control action, and current state are accurately described within the system model. Whenever the model monitor is true, the controller monitor is then used to prune the action space to only known-safe actions.

JSC also takes as input both ***CM*** : $A \times S \to \mathbb{B}$ as a controller monitor and ***MM*** : $S \times A \times S \to \mathbb{B}$ as model monitor where S is the set of states and A the set of actions.

Justified Speculative Learning (a.k.a. JSC).

```
1  JSC(init, (S,A,R,E), choose, update, done, CM, MM) {
2    prev := curr := init;
3    a0   := NOP;
4    while (!done(curr)) {
5      if (MM(prev, a0, curr))
6        u := choose({a ∈ A | CM(a,curr)});
7      else
8        u := choose(A);
9      prev := curr;
10     curr := E(u, prev);
11     update(prev, u, curr);
12   }
13 }
```

Lines 2 – 3 begin the process in an initial state; the model monitor will always return true for inputs $s_1, \texttt{NOP}, s_2$ where $s_1 = s_2$. Lines 4 – 12 choose the next action and execute the chosen action until reaching a terminal state. If the model describes the transition from the previous state to the current state via the chosen control action, then a safe action that comports with the controller action is chosen (Lines 5 – 6). Otherwise, the system is allowed to speculate because the model that lead to MM does not accurately characterize the system E (lines 7 – 8). This code assumes that the model's control policy exhibits a *liveness* property; i.e., the set

$$\{a \in A \mid \textbf{\textit{CM}}(a, curr)\}$$

is always non-empty; we allow non-liveness in the theoretical treatment following this section[3].

[3] The theoretical treatment following this section handles model monitors for non-live models, but such models are often broken.

Finally, on lines 9 – 11, the state is updated according to the environment and the learning model (e.g., a Q-table or NN) is updated.

7.3 Safe Learning

The JSC algorithm explores safely whenever the environment E is accurately modeled by the $\mathsf{d}\mathcal{L}$ theorem from which *CM* and MM are defined. This section presents a precise statement of this assertion, effectively demonstrating how to transfer hybrid systems formal verification results to reinforcement learning.

We begin by recalling the definition of a learning process, which is essentially a dynamical systems encoding of the pseudo-code for the JSC algorithm. Re-stating this algorithm as a dynamical system allows us to give a precise argument without defining a formal semantics for pseudo-code.

Definition 12 (Learning Process)**.** *A tuple of sequences (s_i, u_i, L_i) is a **learning process** for*

$$(init, (S, A, R, E), choose, update, done, \textbf{\textit{CM}}, \textbf{\textit{MM}})$$

if it satisfies the recurrence relations

$$u_i = choose_{L_i}(\{u_i \in A \mid \textbf{\textit{specOK}}(u, s, i)\}) \tag{7.1a}$$

$$s_{i+1} = E(u_i, s_i) \tag{7.1b}$$

$$L_{i+1} = update(L_i, s_i, u_i) \tag{7.1c}$$

where $s_0 \models init$, L_i is a sequence of learned models, and

$$specOK(u, s, i) \equiv \textbf{\textit{CM}}(u_i, s_i) \vee \neg MM(s_{i-1}, u_{i-1}, s_i)$$

The three sequences L_i, u_i, s_i all terminate whenever there is no choice for u_i (i.e., an empty set is passed into the $choose$ function), or else when $done(s_i)$. Indices i are non-negative and the predicate $MM(s_{i-1}, u_{i-1}, s_i)$ evaluates to true whenever $i < 1$.

The sequences u, s, L are the selected control action, state, and learned model (e.g., Q table or NN) at each step of the JSC algorithm. The recurrence relations are equivalent to the computations performed in the JSC pseudo-code, except the liveness caveat discussed in the previous section.

Corollary 2 (Meaning of Controller Monitor)**.** *Suppose CM is a controller monitor for $P \rightarrow [\{\textit{ctrl}; \textit{plant}\}^*]Q$, $s \in S$ is a state and $u : S \rightarrow S$ is a controller. Then **CM**(u, s) implies $(s, u(s)) \in [\![\textit{ctrl}]\!]$.*

Corollary 3 (Meaning of Model Monitor)**.** *Suppose MM is a model monitor for init $\rightarrow [\{\textit{ctrl}; \textit{plant}\}^*]Q$, and that (u, s, L) is a learning process. If **MM**(s_{i-1}, u_{i-1}, s_i) for all i then $s_i \models Q$, and also $(s_i, u_i(s_i)) \in [\![\textit{ctrl}]\!]$ implies $(u_i(s_i), s_{i+1}) \in [\![\textit{plant}]\!]$.*

Lemma 2. *Suppose init $\rightarrow [\{\textit{ctrl}; \textit{plant}\}^*]$safe. If $s \models$ init and $(s, t) \in [\![\textit{ctrl}]\!]$ then $t \models [\textit{plant}]$safe.*

Proof. Validity of $init \rightarrow [ctrl; plant]safe$ implies that if $s \models init$ then $s \models [ctrl; plant]safe$ as well. Therefore, $s \models [ctrl][plant]safe$ by soundness of the box compose axiom. From this fact and the semantics of the box modeality, $(s, t) \in [\![ctrl]\!]$ implies $t \models [plant]safe$. □

Lemma 3. *If A is a sequence of actions selected by JSC and S is a sequence of states at the beginning of each control loop then for all $0 \leq i \leq |A|$, $A_i(S_i) \models [plant]safe$.*

Proof. The proof proceeds by induction on the sequence of actions A.

We have assumed $S_0 \models init$. If JSC selects action A_0, then $CM(A_0, S_0)$; i.e., $(S_0, A_0(S_0)) \in [\![ctrl]\!]$. From these two facts, it follows that $A_0(S_0) = S_1 \models [plant]safe$ by Lemma 1.

The proof for the inductive step is similar. Suppose $A_i(S_i) \models [plant]safe$. It suffices to show $A_{i+1}(S_{i+1}) \models [plant]safe$. If JSC selects action A_{i+1}, then $CM(A_{i+1}, S_{i+1})$ by line X; i.e., $(S_{i+1}, A_{i+1}(S_{i+1})) \in [\![ctrl]\!]$. From these two facts, it follows that $A_{i+1}(S_{i+1}) \models [plant]safe$ by Lemma 1. □

We are now ready to state the first major safety property – that JSC does not violate the system's safety properties *during* reinforcement learning if the environment is accurately modeled.

Definition 13. *An environment E is **accurately modeled** by a system $\{ctrl; plant\}^*$ for a set of actions A and states S if for all $s \in S$ and $u \in A$,*

$$(s, u(s)) \in [\![ctrl]\!] \text{ implies } (u(s), E(s, u)) \in [\![plant]\!] \tag{7.2}$$

Theorem 3 (JSC Explores Safety in Modeled Environments)**.** *Assume a valid safety specification*

$$\models \ init \rightarrow [\{ctrl; plant\}^*]safe \tag{7.3}$$

i.e., any repetition of $\{ctrl; plant\}$ starting from a state in init will end in a state described by safe. Further assume that the proof of this property proceeded by identifying a loop invariant J. Then $u_i(s_i) \models$ safe for all u_i, s_i satisfying the learning process for

$$(init, (S, A, R, E), choose, update, done, CM, MM)$$

where CM and MM are the controller and model monitor for init $\rightarrow [\{ctrl; plant\}^*]$safe.

Proof. We prove, by induction on the sequences s_i and u_i, the stronger property that

$$s_i \models J$$

and

$$(s_i, u_i(s_i)) \in [\![ctrl]\!] \wedge u_i(s_i) \models [plant]J$$

Base Case For the first control action, notice that $s_0 \models init$ is a pre-condition of the algorithm. We've assumed J is a loop invariant, so $\models init \rightarrow J$ as well. From these two facts, $s_0 \models J$.

If the sequence terminates at s_0 then the proof is complete. Otherwise, there must exist some action u_0 satisfying the constraint in (7.1a). The negation of the model monitor MM is false initially because $i - 1 = 0 - 1 < 0$, so $CM(u_0, s_0)$ must be true. By Corollary 2, $(s_0, u_0(s_0)) \in [\![ctrl]\!]$. Therefore, $u_0(s_0) \models [plant]J$ by 2 and our assumption that J is a loop invariant.

Inductive Case Assume $s_i \models J$ and $u_i(s_i) \models [plant]J$ for some $i \geq 0$.

By (7.1b), $s_{i+1} = E(u_i, s_i)$. We have assumed that E is accurately modeled, so $(u_i(s_i), s_{i+1}) \in [\![plant]\!]$. From this fact and the inductive hypothesis, $s_{i+1} \models J$.

If $done(s_{i+1})$ or if no u_{i+1} satisfying (7.1a) exists then the proof is complete. Otherwise, there must be some u_{i+1} that does satisfy this constraint. Either $MM(s_i, u_i, s_{i+1})$ false or else the $CM(s_i, u_i)$ is true. Notice that $MM(s_i, u_i, s_{s+1})$ is true due to our assumption that the environment is accurately modeled and the inductive hypothesis. Therefore, $CM(u_{i+1}, s_{i+1})$ must be true. By Corollary 2, $(s_{i+1}, u_{i+1}(s_{i+1})) \in [\![ctrl]\!]$.

The second part of the induction hypothesis follows from this fact, 2, and our assumption that J is a loop invariant: $u_{i+1}(s_{i+1}) \models [plant]J$. □

Theorem 3 states that whenever the environmental model is accurate, every state we reach via JSC satisfies the safety property *safe*. The proof of this property exploits the fact that any proof of *init* $\rightarrow$ [{*ctrl*; *plant*}*]*safe* will proceed by identifying a loop invariant [150].

7.4 Safe Policy Extraction

The ultimate output of the learning algorithm is a control policy π. Many reinforcement algorithms allow this policy to be extracted after some learning period. If a known-safe fallback policy is provided, Theorem 3 also extends verification results to extracted policies.

Theorem 4 (Safety of Extracted Policies)**.** *Assume*

$$\models \textit{init} \rightarrow [\{\textit{ctrl}; \textit{plant}\}^*]\textit{safe}$$

and assume the existence of a policy called $fallback$ that is safe with respect to this specification. Further assume that $fallback \subseteq [\![ctrl]\!]$.

Consider a learning process (u_i, s_i, L_i) in which E is an accurate model and CM, MM are controller and model monitors for $init \rightarrow [\{ctrl; plant\}^*]safe$.

If π is a policy such that $\pi(s) = t$ implies either (s, t) is explored during the learning process or else $(s, t) \in fallback$, then π is safe with respect to the above specification.

Proof. Let $f \equiv \pi \circ [\![plant]\!]$ and suppose J is the loop invariant which witnesses the safety of $fallback$. It suffices to show that $f^n(s_0) \in J$ for any $n \geq 0$ and $s \in [\![init]\!]$. The proof proceeds by induction on n.

Notice that $s_0 \in [\![init]\!]$, so by our assumption that J is a loop invariant, $s \in [\![J]\!]$ as well.

Assume $s_i = f^i(s_0) \in J$ for $i \geq 0$ and let $t = \pi(s_i)$. If $t = fallback(s_i)$ then $plant(t) \in J$ because $fallback$ is safe. Therefore, $f^{n+1}(s_0) = (\pi \circ [\![plant]\!])(s_i) \in J$.

Otherwise, it must be the case that the (s_i, t) is an explored action in the learning process. In that case, $\pi(s_i) = u_i(s_i)$ for some u_i such that either $CM(u_i, s_i)$ or $\neg MM(s_{i-1}, u_{i-1}, s_i)$. Notice that $MM(s_{i-1}, u_{i-1}, s_i)$ follows from our assumptions that E is an accurate model and that $fallback \subseteq [\![ctrl]\!]$. Therefore, $CM(u_i, s_i)$. By Corollary 2, $(s_i, t) \in [\![ctrl]\!]$.

So we know $s_i \models J$, $(s_i, \pi(s_i)) \in [\![ctrl]\!]$, and $\models J \rightarrow [ctrl; plant]J$. By 2, $\pi(s_i) \models [plant]J$ which implies $plant(t) \models J$. Therefore, $(\pi \circ [\![plant]\!])(s_i) = f^{n+1}(s_0) \models J$. □

Table 7.1: A Comparison of JSC and Classical Q-Learning in a Modeled Environment.

	JSC			Normal		
Training steps	Crash	Fall Behind	Steady	Crash	Fall Behind	Steady
1,000	0	12559	289	10644	2162	42
10,000	0	12538	310	10462	2291	95
100,000	0	12375	473	10492	2284	72

7.5 Experimental Validation

The theoretical results presented in the Provably Safe Learning section apply to generic reinforcement learning algorithms. This section presents two sets of simulations that expore JSC in the concrete setting of classical Q-learning on a simple model of adaptive cruise control. The first set of experiments validate the theoretical results within this concrete setting. The second set of experiments consider a case where the environment deviates from modeling assumptions, violating the key assumption made in Theorem 3. The second set of experiments demonstrate that verification can help improve the learning process itself.

The JSC algorithm is parametric in the approach toward reinforcement learning. We choose Q-learning because it is a simple algorithm, and our example is simple enough that discretized Q-learning is possible. We leave exploration of JSC-style control in more complex environments, and with more effective reinforcement learning algorithms, as future work.

The setting of this experiment is a simple model of adaptive cruise control. These experiments are implemented in a new linear Adaptive Cruise Control[4] OpenAI Gym environment [28] based on [128].

7.5.1 Adaptive Cruise Control

Adaptive Cruise Control (ACC) is an increasingly common feature in passenger vehicles. Unlike traditional cruise control, ACC adjusts the speed of a car relative to the speed and distance of a leader car. The safety property for adaptive cruise control is simple: an actuated follower car must avoid crashing into a leader car.

Our experiment considers Adaptive Cruise Control for two cars. We use relative coordinates to reduce the state space, so instead of two positions $pos_{follower}$ and pos_{leader} we have a single relative position

$$r_{pos} = |pos_{leader} - pos_{follower}|$$

A relative coordinate system allows us to exploit symmetries in the state and action space during learning and increases the likelihood that the system will return to a modeled state after an environmental perturbation.

The relative position of the two cars r_{pos}, is non-negative whenever the cars do not collide. The relative velocity between the cars is zero whenever the cars are stationary relative to one another, positive whenever the cars are moving apart, and negative whenever the cars are moving

[4]This implementation is available at github.com/nrfulton/JSC

Table 7.2: A Comparison of JSC and Q-Learning with Error Injection (.05 error rate).

	JSC			Normal		
Training steps	Crash	Fall behind	Steady	Crash	Fall behind	Steady
1,000	3	12539	306	10950	1745	153
10,000	7	12502	339	10546	2215	87
100,000	5	12359	484	10561	2242	45

closer together. We consider a discrete action space – the actuated car may brake with constant force B, accelerate with constant force A, or maintain its current relative velocity.

Model 1 (Relative Acceleration Along a Straight Line)**.**

$r_{pos} \geq 0 \wedge A > 0 \wedge B > 0 \wedge T > 0 \wedge r_{pos} \geq \frac{r_{vel}^2}{2A} \rightarrow$
$[\{\{r_{acc} := A$
$\quad \cup\ ?r_{pos} - \frac{-BT+r_{vel}^2}{2A} + r_{vel}T - \frac{BT^2}{2} \geq 0;$
$\quad\quad r_{acc} := -B$
$\quad \cup\ ?r_{vel} = 0;\ r_{acc} := 0$
$\quad\}; \{r'_{pos} = r_{vel}, r'_{vel} = r_{acc}, c' = 1 \& c \leq T\}$
$\}^*] r_{pos} \geq 0$

The above model presents a $\mathsf{d}\mathcal{L}$ formula that corresponds to this system. On every iteration of the control loop, the follower actuates the relative acceleration r_{acc}. By choosing a positive second derivative for relative acceleration, the car *slows* its rate of approach toward the leader; therefore, the action $r_{acc} := A$ is always permitted. However, choosing a negative relative acceleration *increases* the cars' rate of approach toward one another; therefore, choosing the action $r_{acc} := -B$ requires first checking the resulting braking distance. Finally, if the car is already stopped ($r_{vel} = 0$), then it may remain stopped ($r_{acc} := 0$). From this model, we extract controller and model monitors for JSC Q-learning using ModelPlex. Notice that this controller can be rather inefficient. In particular, one safe but inefficient deterministic implementation of this model could choose to always increase the distance between the two cars by choosing $r_{acc} := A$.

7.5.2 Experimental Setup and Results

This section presents two experiments. In the first experiment, we validate the safe learning theorem presented in the previous section. In the second experiment, we go beyond provably safe sandboxing to determine whether verification results might be useful even when models are inaccurate; i.e., even when the *accurately modeled* assumption (as stated in Def. 13) is violated.

JSC in an Accurately Modeled Environment Our first experiment validates our theoretical results by the performance of JSC and Q-learning in an accurately modeled environment. The results of this experiment are recorded in Table 7.1. We ran training for a specified number of steps ($n = 1,000; 10,000;$ and $100,000$). We then iterated across each of the initial states and determined how the policy behaved in each case, omitting in which no safe policy was available (i.e., states where even braking with maximum braking force would result in crashes within the chosen discretized space).

The Crash columns indicate, for the policy extracted by each approach, the number of initial states that resulted in a crashing state. The Fall Behind columns indicate the number of initial states that resulted in a policy that brakes too often; i.e., where the follower car loses track of the lead car. The Steady columns indicate the number of initial states that resulted in an ideal policy – one that does not lose the lead car, but also does not result in a collision.

We observe that the JSC controller never enters unsafe states during training. But we also observe that JSC encounters significantly fewer fall-behind states and more steady states. This result indicates that learning with JSC is more efficient than normal learning. The experimental observation that JSC is more efficient at learning effective policies deserves further exploration.

JSC in an Environment that Admits Speculation The results presented above demonstrate, via proof and experiment, that Justified Speculative Control effectively transfers proofs of correctness from verified models to reinforcement learning algorithms.

In our second experiment (Table 7.2), we force speculation by simulating an environment that behaves differently from the modeled system. We introduce a slight perturbation to the relative position of the cars (-2 units) with 5% probability. The policy extracted after running our JSC algorithm for any period of time results in relatively few crashes (< 10 out of $20,000$ possible initial states result in a crash). All of these crashing states are attributable to actuator faults.

During each experiment, we extracted the learned policy for JSC and for classical Q-learning after N simulations. We then evaluated the resulting policy from every possible initial position. These results demonstrate that JSC can control reasonably well even when there are small perturbations between the model and the simulated environment.

7.6 Quantitative JSC

In our final set of experiments, we move beyond viewing logical constraints as mere sandboxes for a learning process or targets for sandbox synthesis. ModelPlex works by reducing a $\mathsf{d}\mathcal{L}$ formula to a quantifier-free formula of real arithmetic. We exploit this fact to generate *quantitative real-valued signals* from our qualitative boolean-valued functions.

Our modified algorithm uses the distance between the current state and the modeled environment as an objective function during speculation; when the system leaves the modeled state space, the reinforcement learning agent optimizes for returning to the modeled portion of the state space. We call this approach JSC**Q**.

Table 7.3: Crashing States for JSC and JSCQ Control.

Perturbation	JSC	JSCQ
5%	3	2
25%	18	16
50%	41	34

For small positional perturbations, JSCQ and JSC perform equally well. However, we found that as the positional perturbation increases, the modified algorithm begins to outperform the original algorithm. Table 7.3 presents these results.

7.7 Conclusion

Leveraging an existing model and control policy substantially increases the data efficiency of JSC relative to naïve learning. Unlike traditional verification approaches, JSC explains how to optimize for goals that are not safety-crtical. Verification results transfer not only to final policies, but also to the learning process itself. This might enable the use of online reinforcement learning in production environments. This combination on improved data efficiency and safety guarantees for the training process itself might enable the use of reinforcement learning in real, non-simulated environments.

The justified speculative control algorithm discussed in this chapter presents one approach toward verifiably safe learning. Reinforcement learning is sandboxed by monitors derived from verification results, and a quantitative version of the monitor provides a signal that appears to give reasonable heuristics in unmodeled environments. However, this approach has two significant limitations.

Limited Applicability Justified speculative control is currently limited to simple hybrid systems with few state variables and linear continuous dynamics. There are two reasons for this limitation. First, the ModelPlex tactic can only generate exact monitoring conditions when the differential equations have a solution that exists in the first order theory of real closed fields[5]. Second, justified speculative control assumes it is possible to enumerate the entire state space during the training phase. These two limitations are crippling: JSC is not applicable to hybrid systems with nonlinear continuous dynamics and does not work well for models with large state spaces. Because of these limitations, the justified speculative control algorithm introduced by Fulton et al. [62] is only validated on very simple models.

Lack of Guarantees for Unmodeled Environments Ideally, an approach toward verified safe autonomy should come with guarantees of safety and/or optimality even in cases where the original model is slightly wrong. Small deviations from the model should not typically lead to catastrophic failures.

Overcoming these limitations requires two new contributions. First, we must expand justified speculative control to large and/or continuous state and action spaces. Second, we must find a way to identify and resolve systematic disparities between models and reality. And when a more accurate model is discovered, this new model must be incorporated into justified speculative control.

[5] in particular, these solutions may not contain any exponential or trigonometric functions

Chapter 8

Model Update Learning

The justified speculative control algorithm presented in Chapter 7 takes a first step toward verifiably safe learning by transferring safety guarantees to learned controllers within model space and by leveraging insights from verification during exploration outside of model space. Other examples of approaches toward controlling well inside model space include shielding [5], logically constrained learning [83], and constrained Bayesian optimization [73].

Each of these approaches provide formal safety guarantees for reinforcement learning and/or optimization algorithms by stating assumptions and safety constraints in a formal logic, generating monitoring conditions based upon safety constraints and environmental assumptions, and then leveraging these monitoring conditions to constrain the learning/optimization process to a known-safe subset of the state space.

Existing formal methods for learning and optimization consider the problem of constrained learning or constrained optimization [5, 62, 62, 73, 83]. They address the question: assuming we have a single accurate environmental model with a known safety constraint, how can we learn an efficient control policy respecting this safety constraint?

Safety proofs for well-modeled environments are necessary but not sufficient for ensuring that reinforcement learning algorithms behave safely. In fact, a significant motivator for using reinforcement learning in safety-critical systems is the fact that explicit and accurate environmental models are not readily available. Although some formal methods suggest ways in which formal constraints might be used to inform control even when modeling assumptions are violated, none of these approaches provide formal safety guarantees when environmental modeling assumptions are violated.

Holistic approaches toward safe reinforcement learning should provide formal guarantees even when a single, a priori model is not known at design time. We call this problem **verifiably safe off-model learning**. In this chapter we introduce a first approach toward obtaining formal safety proofs for off-model learning[1]. Our approach consists of two components: (1) a model synthesis phase that constructs a large set of candidate models together with provably correct control software, and (2) a runtime model identification process that selects between available models at runtime in a way that preserves the safety guarantees of all candidate models.

Model update learning is initialized with a set of models. These models consists of a set

[1] The work in this chapter extends a vision paper on *Safe AI for CPS* by Fulton and Platzer [63].

of differential equations that model the environment, a control program for selecting actuator inputs, a safety property, and a formal proof that the control program constrains the overall system dynamics in a way that ensures the safety property is never violated.

Instead of requiring the existence of a single accurate initial model, we introduce *model updates* as syntactic modifications of the differential equations and control logic of the model. We call a model update *verification-preserving* if there is a corresponding modification to the formal proof establishing that the modified control program continues to constrain the system of differential equations in a way that preserves the original model's safety properties.

Beginning with a single provably safe model, we construct at design time a large class of possible models (and their associated controllers) by iteratively applying updates to both the ODEs and their associated control programs in a way that preserved verification guarantees. Runtime falsification is then used to choose between these possible models.

We begin by introducing a conceptual framework called *verification-preserving model updates (VPMUs)*. These updates preserve verification guarantees by changing both the controller and the model in systematically similar ways. Each model consists of a set of differential equations that serve as candidate models for the environment, a control program, a safety property, and a formal proof that the control program constrains the overall system dynamics to a safe subset of the state space. Instead of assuming that an initial model (or set of models) is perfectly accurate, we introduce *model updates* as syntactic modifications of differential equations and controllers. We call a model update *verification-preserving* if there is a corresponding modification to the formal proof establishing that the modified control program continues to constrain the system of differential equations in a way that preserves the original model's safety properties.

The canonical example of a verification-preserving model update is parameter instantiation: a static parameter p that is known to be between p_{min} and p_{max} and whose precise value is possible to determine only at runtime. Instantiating p with a concrete value θ between p_{min} and p_{max} is a verification-preserving model update, as long as p is uniformly replaced with θ in both the model and the proof.

Verification-preserving model updates are inspired by the fact that different parts of a model serve different roles. The continuous portion of a model is typically an assumption about how the world behaves, and the discrete portion of a model is derived from these equations. For this reason, many of our updates inductively synthesize ODEs (i.e., in response to data) and then deductively synthesize control logic from the resulting ODEs.

This chapter also introduces an algorithm, called model update learning (μlearning), that explains how to solve this problem. μlearning uses runtime falsification to control both safely and efficiently in the presence of many feasible models. In experimental validation, we show that μlearning combined with generic model updates enables provably safe control even when an initial model contains inaccuracies.

Our experiments demonstrate that the μlearning framework is surprisingly effective.

This chapter introduces a first approach toward verifiably safe off-model learning. Our contributions enabling this approach include:

1. A set of verification-preserving model updates (VPMUs) that systematically update differential equations, control software, and safety proofs in a way that preserves verification guarantees while taking into account possible deviations between an initial model and fu-

ture system behavior.

2. A reinforcement learning algorithm, called model update learning (μlearning), that explains how to transfer safety proofs for a set of feasible models to a learned policy. The learned policy will attempt to falsify models at runtime.

In addition to proving a safety theorem for μlearning similar to the safety theorem given in Chapter 7 for justified speculative control, our contributions are also experimentally validated on several examples of safety-critical control problems. One of our examples also demonstrates that μlearning can also be leveraged to extending safety results for reinforcement learning algorithms to hierarchical reinforcement learning algorithms.

VPMUs and μlearning provide one approach toward verifiably safe off-model learning. Instead of considering completely model-free learning, we assume that there is an initial model and a set of model updates such that an accurate model can be generated by successively applying model updates to the initial model. Although the most accurate model cannot be determine at design time, runtime falsification can be used to select the most accurate model. In Chapter 9, we will explore a purely data-oriented approach toward this problem.

8.1 Verification-Preserving Model Updates

A *verification-preserving model update* (VPMU) is a transformation of a hybrid program accompanied by a proof that the transformation preserves key safety properties. VPMUs capture situations in which a model and/or a set of data can be updated in a way that captures possible runtime behaviors which are not captured by an existing model. VPMUs are a reasonable approach toward constructing a robust set of possible models whenever system designers can characterize likely ways in which an existing model will deviate from reality.

Definition 14 (VPMU)**.** *A verification-preserving model update is a pair of mappings* `modelUpdate` *and* `proofUpdate` *which take as input an initial* dL *formula* φ *with an associated Bellerophon proof* `e` *of* φ*, and produce as output a new* dL *formula* `modelUpdate`(φ) *and a proof* `proofUpdate`$(\texttt{e})$ *of* `modelUpdate`(φ).

An Example The simplest example of a VPMU instantiates a parameter whose value is not known at design time but can be determined at runtime via system identification. Consider the program p modeling a car whose acceleration depends upon both a known control input $accel$ and parametric values for maximum braking force $-B$ and maximum acceleration A:

$$p \equiv \texttt{init}_p \rightarrow [\{\texttt{ctrl}_p; c := 0; \texttt{plant}_p\}^*]\texttt{safe}_p$$

where

$$\begin{aligned}
\texttt{init}_p &\equiv A > 0 \land B > 0 \land T > 0 \land \texttt{obsPos} - \texttt{pos} > \frac{\texttt{vel}^2}{2B} \\
\texttt{safe}_p &\equiv \texttt{obsPos} > \texttt{pos} \\
\texttt{plant}_p &\equiv \{\texttt{pos}' = \texttt{vel}, \texttt{vel}' = \texttt{accel}, c' = 1 \& \texttt{vel} \geq 0 \land T \geq 0\}
\end{aligned}$$

and the controller $\texttt{ctrl}_p$ allows the car to choose between braking, accelerating, or maintaining the current acceleration (with associated safety checks for each case):

$$\begin{aligned}
&\texttt{accel} := -B \cup \\
&\{ ?obsPos - (pos + \frac{T^2{}_{\texttt{accel}}}{2} + T \cdot {}_{\texttt{vel}}) > \frac{({}_{\texttt{vel}} + {}_{\texttt{accel}}T)^2}{2B}; \\
&\quad \texttt{accel} := A \} \cup \\
&?obsPos - pos + T \cdot \texttt{vel}) > \frac{\texttt{vel}^2}{2B}; \texttt{accel} := 0
\end{aligned}$$

The proof of this formula is given by the following Bellerophon proof script which identifies $\texttt{pos} > \frac{\texttt{vel}^2}{2B}$ as the critical loop invariant and defers the rest of the deductive reasoning to our automated theorem prover:

$$\texttt{implyR}(1); \texttt{loop}(\texttt{pos} - \texttt{obsPos} > \frac{\texttt{vel}^2}{2B}); \texttt{OnAll}(\texttt{auto})$$

The value B represented the maximum braking force of the car. The permissible range always depends upon context: tire pressure, tread, road conditions, and (perhaps most importantly) models of vehicle predictability. The above model can be updated to work for a concrete instantiation by choosing e.g., $B = 12$. Concretely, choosing $B = 12$ is achieved through a model update that replaces every occurrence of B with 12 in both the model and the proof script.

Notice that choosing a concrete value for our maximum braking force allows more precise and less pessimistic control; otherwise, the controller must either a) assume that B could take on any non-negative value and therefore control in an extremely conservative fashion; or b) assume that the system designer chose and correctly instantiated a concrete value for B.

As noted in the introduction, verification-preserving model updates are inspired by the fact that different parts of a model serve different roles. The above example aptly demonstrates this separation. The continuous portion of a model (`pos'=vel,vel'=accel`) is typically an assumption about how the world behaves, and the discrete portion of a model (`ctrl`) is derived from these equations.

Before giving further concrete examples of model updates, we consider how a set of feasible models computing using VPMUs can be used to provide verified safety guarantees for a family of reinforcement learning algorithms. The primary challenge is to maintain safety with respect to all feasible models while also avoiding overly conservative monitoring constraints by falsifying some of these models at runtime. After introducing several generic model updates and explaining how to generate a set of feasible models from a set of initial models and a set of model updates, Section 8.4 introduces a reinforcement learning algorithm that uses model updates. Finally, in Section 8.5, we introduce several concrete examples in which model update learning is useful and discuss the role of domain-specific model updates.

8.2 From Model Updates to Feasible Models

This chapter considers two types of model updates: generic and domain-specific. Generic updates are applicable in a broad class of systems and are not specific to any particular model or

application domain. Conversely, domain-specific model updates are transformations which are very specific to a particular class of dynamics (e.g., sinusoidal) or models (e.g., adaptive cruise control).

This section introduces several generic verification-preserving model updates (some domain specific updates are introduced in Section 8.5). For each generic update, we discuss how the model is modified, how the controller is modified, and how the proof is modified. After discussing several generic model updates that are used in our later examples, we also introduce a simple algorithm for generating a set of feasible models given a set of initial models and a set of updates. All of these updates are implemented using the KeYmaera X theorem prover. These implementations often contain significant implementation details; this section attempts to explain the transformations at a high level.

8.3 A Model Update Library

So far, we have established how to obtain safety guarantees for reinforcement learning algorithms given a set of formally verified dℒ models. We now turn our attention to the problem of generating such a set of models by systematically modifying dℒ formulas and their corresponding Bellerophon tactical proof scripts. This section introduces five generic model updates that provide a representative sample of the kinds of computations that can be performed on models and proofs to predict and account for runtime model deviations.

Parameter Instantiation The simplest example of a VPMU instantiates a parameter whose value is not known at design time but can be determined at runtime via system identification. Consider the program p modeling a car whose acceleration depends upon both a known control input $accel$ and parametric values for maximum braking force $-B$ and maximum acceleration A: $p \equiv \texttt{init}_p \rightarrow [\{\texttt{ctrl}_p; c := 0; \texttt{plant}_p\}^*]\texttt{safe}_p$ where

$$\begin{aligned}\texttt{init}_p &\equiv A > 0 \land B > 0 \land T > 0 \land \texttt{obsPos} - \texttt{pos} > \frac{\texttt{vel}^2}{2B}\\ \texttt{safe}_p &\equiv \texttt{obsPos} > \texttt{pos}\\ \texttt{plant}_p &\equiv \{\texttt{pos}' = \texttt{vel}, \texttt{vel}' = \texttt{accel}, c' = 1 \& \texttt{vel} \geq 0 \land T \geq 0\}\end{aligned}$$

and the controller allows the car to choose between braking, accelerating, or maintaining the current acceleration (with associated safety checks for each case):

$$\begin{aligned}&\texttt{accel} := -B \cup\\ &\{\ ?\texttt{obsPos} - (\texttt{pos} + \frac{T^2\texttt{accel}}{2} + T \cdot \texttt{vel}) > \frac{(\texttt{vel} + \texttt{accel})^2}{2B};\ \ \texttt{accel} := A\ \} \cup\\ &?\texttt{obsPos} - \texttt{pos} + T \cdot \texttt{vel}) > \frac{\texttt{vel}^2}{2B}; \texttt{accel} := 0\end{aligned}$$

The proof of this formula is given by a following Bellerophon tactic. The tactic first moves the initial conditions into the antecedent, and identifies $\texttt{pos} > \frac{\texttt{vel}^2}{2B}$ as the critical loop invariant.

Establishing a formula is invariant throughout the execution of a loop requires proving several subgoals. The tactic then uses the KeYmaera X automated tactic `master` on each of these subgoals (via the `OnAll` tactic):

$$\texttt{implyR}(1); \texttt{loop}(\texttt{pos} - \texttt{obsPos} > \frac{\texttt{vel}^2}{2B}, 1); \texttt{OnAll}(\texttt{master})$$

The value B represented the maximum braking force of the car. The permissible range always depends upon context: tire pressure, tread, road conditions, and (perhaps most importantly) models of vehicle predictability. The above model can be updated to work with for a concrete instantiation by choosing e.g., $B = 12$. Concretely, choosing $B = 12$ is achieved through a model update that replaces every occurrence of B with 12 in both the model and the tactic.

Notice that choosing a concrete value for our maximum braking force allows more precise and less pessimistic control; otherwise, have to assume that B could take on any non-negative value and therefore control in an extremely conservative fashion.

Automatic Compact Domain Parameter Instantiation The automatic parameter instantiation update does not require the user to identify a parameter or a parameter value. Instead, we search the formula for parameters that are restricted to a compact domain and then choose discrete values from this domain at a user-defined level of precision.

- Model and Controller Update: given a formula init $\rightarrow [\alpha]$safe, the automatic parameter instantiation update identifies a parameter p such that init $\rightarrow p_{min} \leq p \leq p_{max}$ for some $p_{min}, p_{max} \in \mathbb{R}$. The update produces several new models by performing the normal parameter instantiation update for finitely values between p_{min} and p_{max} using a user-specified step size.
- Proof Update: For each choice of parameter p and value $\theta \in [p_{min}, p_{max}]$, the proof update for automatic parameter instantiation is identical to the proof update for normal parameter instantiation.

The automatic parameter instantiation update improves the basic parameter instantiation update by automatically detecting which variables are parameters and by constraining the instantiation of these parameters to values that are permitted by the original model.

Together, parameter instantiation and automatic parameter instantiation provide a method for system designers to automatically explore the space of possible parameter values when possible, while also providing system designers with the option to (carefully) instantiate parameters with concrete values obtained experimentally.

Replace Worst-Case Bounds with Approximations Models designed for the purpose of safety verification are often worst-case analyses. Often a variable occurring in the system is bounded above (or below) by its worst-case value. Worst-case analyses are sufficient for establishing safety but are often overly conservative. The approximation model update replaces worst-case bounds with approximate bounds obtained via series expansions. The model update identifies a variable x satisfying the following conditions:

- The variable x has a differential equation,

- the solution to $x' = f(x)$ can be approximated both above and below by a Taylor series[2], and
- there are parameters (or numbers) x_{max} or x_{min} such that $x \leq x_{max}$ or $x_{min} \leq x$ occurs in either the initial conditions for the model or the evolution domain constraint of the model.

If such a variable x exists, the following update may be applied:

- The differential equations do not change.
- Controller update: For each control choice of the form $?g_i;\ u_i := \theta_i$, we replace all instances of x_{max} and x_{min} occurring in g_i with either $\text{low}(x)$ or $\text{hi}(x)$. In some cases, occurrences of x are also replaced by $\text{low}(x)$ or $\text{hi}(x)$ depending on the context in which x occurs. A similar transformation happens in all θ_i for $u_i \neq x$.
- Proof update: Prior to any differential tactic[3], we introduce a series approximation tactic that establishes $\text{low}(x) \leq x \leq \text{hi}(x)$ as an invariant of the differential equations.

Add Disturbance/Noise Terms Models often assume perfect sensing and actuation. A common way of robustifying a model is to add a piecewise constant noise term to the system's dynamics. Doing so while maintaining safety invariants requires also updating the controller so that safety envelope computations incorporate this noise term.

The **Add Disturbance Term** update introduces noise terms to differential equations. If the user does not specify the term that should have noise added, the update identifies an actuated variable u; a variable is considered an actuator if it occurs on the right-hand side of a differential equation and is assigned to in the discrete fragment of the hybrid program. Given a variable u – identified by the user or automatically according to the above criteria – the update is defined as follows.

- Differential Equations: the differential equations $c, v' = f$ (where c is an arbitrary system and f an arbitrary polynomial) are modified to $c, v' = f + n$. For example, the equations $x' = v, v' = a$ can be updated with a noise term $x' = v, v' = a + n$.
- Controller: Each controller guard that mentions f is updated to $f + n$.
- Initial Condition: When introducing a parameter, the user may choose the specify an upper bound on the noise term, a lower bound on the noise term, both, or neither. The initial conditions init are then updated to $\text{init} \wedge n_{min} \leq n \leq n_{max}$ where $n_{min}, n_{max} \in \mathbb{R}$ are (optional) user-specified upper and lower bounds on n. Although n is a fresh variable and therefore has no dynamics, the bounds may be arbitrary terms mentioning both free and bound variables of the original model.
- Proof Update: All loop invariants, cuts, and differential cuts in the proof update mentions of f to $f + n$.

Our library of generic updates also includes a multiplicative version of this update, where the (optionally bounded) noise term is multiplied to the right-hand side of a specified ODE.

[2]Our current implementation current considers only a few possible Taylor expansions, but demonstrates the general principle.

[3]a tactic operating on a differential equation containing $x' = f$

Change Static Points to Dynamic Points The generic model update library contains several updates that change the model by making a static point (x, y) dynamic. For example, one such update introduces the equations $\{x' = -y, y' = -x\}$ to a system of differential equations in which the variables x, y do not have differential equations. The controller is updated so that any statements about separation between (a, b) and (x, y) require global separation of (a, b) from the circle on which (x, y) moves. The proof is also updated by prepending to the first occurrence of a differential tactic on each branch with a sequence of differential cuts that characterize circular motion. Our model update library also contains similar updates for changing a static obstacle into one that moves along lines in either coordinate.

8.3.1 Linear Hybrid Program Synthesis

Complete model and controller synthesis algorithms can be characterized in terms of model updates. One such example is our synthesis algorithm for systems whose ODEs have solutions in a decidable fragment of real arithmetic (a subset of linear ODEs), called the **Learn Linear Dynamics** update. Unlike other model updates, we do not assume that any initial model is provided; instead, we learn a model (and associated control policy) entirely from data. We defer extended discussion of this update to Chapter 9

Significance of Selected Updates The updates described in this section demonstrate several possible modes of use for VPMUs and μlearning. VPMUS can update existing models to account for systematic modeling errors (e.g., missing actuator noise or changes in the dynamical behavior of obstacles). VPMUs can automatically optimize control logic in a proof-preserving fashion. VPMUS can also be used to generate accurate models and corresponding controllers from experimental data made available at design time, without access to any prior model of the environment.

8.3.2 From Updates to Candidates

The rest of this chapter will consider the problem of starting with a set of feasible models and learning which model is the best representation of observed reality. The following algorithm provides a way of generating such a set of feasible models from a set of initial models and a set of available updates.

The algorithm is very simple: we continue applying all applicable updates to all available models until reaching some finite horizon. The condition on the tenth line of this listing checks that the update proof actually proves the updated model. This step is necessary because, as our above discussion of generic updates indicates, many generic VPMUs are useful in practice but are fundamentally heuristics. which may not actually preserve verification for many concrete examples.

The rest of this thesis considers the following question: given a set of feasible models[4], can we safely learn which model is accurate?

[4]generated, for example, using this algorithm, the generic updates discussed above, and perhaps some domain-specific updates

Listing 8.1: Model Update Generation Pseudocode.

```
1  def generate(models, updates):
2    for m,p in models:
3      assert proveBy(m,p)
4    until reaching horizon:
5      for m,p in models:
6        for mu,pu in updates:
7          continue if m not in dom(mu) or
8                      p not in dom(pu)
9          newm, newp = mu(m), pu(p)
10         if proveBy(newm,newp):
11           models.append( (newm, newp) )
12   return map(lambda mp: mp[0], models)
```

Line 3 of Listing 8.1 asserts that the initial set of model/proof pairs are actually proofs; i.e., that the Bellerophon tactic `p` proves the formula `m`. For each fo the currently available proven models `m,p`, the algorithm applies all available updates to obtain a new model and a new tactic `mu,pu`. These updates might fail on some models; if so, the update is simply excluded from consideration for the model (line 7). However, if the update is applicable and if `pu` proves `mu`, then the new model and proof are added to the set of models (line 9-11). The process repeats until reaching a specified horizon (e.g., a fixed number of iterations or a fixed point).

8.4 Learning with Updates

Verification-preserving model updates generate a large class of *feasible models*. This section introduces Model Update learning, a class of reinforcement learning algorithms that leverage the class of models generated by applying verification-preserving model updates in order to provide safety guarantees for control problems in which a single accurate model cannot be selected at design time.

Model Update learning (μlearning) algorithms all have the same basic idea: begin with a set of *feasible models* and act safely with respect to all feasible models. Whenever a model does not comport with observed dynamics, the model becomes infeasible and is therefore removed from the set of feasible models. We introduce three variations of μlearning: a basic algorithm for a finite set of models, an extension of the basic algorithm to countable model sets, and an algorithm that prioritizes actions that rule out feasible models (adding an *eliminate* choice to the classical explore/exploit tradeoff).

Model Update learning demonstrates how to use a set of models generated by exhaustively applying model updates in order to safely learn efficient policies by combining the JSC algorithm presented in Chapter 7 with online falsification. However, the algorithms presented in this section are equally applicable in settings where a large set of feasible models is provided a priori (i.e., from some source other than exhaustive application of a class of model updates to an initial set of models).

8.4.1 Monitored Models

All μlearning algorithms use monitored models; i.e., models equipped with ModelPlex controller monitors and model monitors.

Definition 15. *A **monitored model** m is a tuple (φ, cm, mm) where:*

- *φ is a dL formula of the form $\texttt{init} \rightarrow [\{\texttt{ctrl};\texttt{plant}\}^*]\texttt{safe}$ where $\texttt{ctrl}$ is a loop-free program containing no ODEs, $\texttt{plant}$ is a single system of differential equations, and the entire formula φ contains exactly one modality.*
- *The formulas cm and mm are the control monitor and model monitor for φ, as defined in Corollary 2 and Corollary 3.*

Notation This chapter uses slightly different notation from Chapter 7 for similar or identical objects. For example, in the above definition, we use *cm* and *mm* instead of *CM* and *MM* to refer to monitors. We also refer to the formula monitored by these monitors as φ. These changes in notation are made to help avoid confusion between monitored models m and their corresponding formulas φ, between models and sets of models, between monitored models and sets of monitored models, and between sets and sequences of each. Throughout this chapter, single monitors and single formulas will be denoted either greek symbols or by lowercase letters, sets will be denoted by uppercase letters (e.g., M for a sequence of monitored models), and sequences will be denoted by bold and uppercase letters (e.g., $\mathbf{S}$ for a sequence of sets). We will also always use m or m_i to refer to *monitored* models whereas φ refers to just the formula used to generate the monitors.

Monitored models may have a continuous action space because of both tests and the nondeterministic assignment operator. Some μlearning algorithms focus on models with a finite action space.

Definition 16. *A **monitored model over a finite action space** is a monitored model (φ, cm, mm) where $\varphi \equiv \texttt{init} \rightarrow [\{\texttt{ctrl};\texttt{plant}\}^*]\texttt{safe}$ and $[\![\texttt{ctrl}]\!](s)$ is finite for all states s.*

Similarly, monitored models over a finite action space are monitored models for which $[\![\texttt{ctrl}]\!]$ has a finite codomain on every input state.

Distinguishing between plausible models with qualitatively similar dynamics requires knowing how much time passes globally and between control inputs. For example, $x' = 1$ and $x' = 42$ are impossible to distinguish between without knowing how much time has passed. For this reason, we consider the class of models with explicit timers.

Definition 17 (Time-aware Monitored Model)**.** *A time-aware monitored model is a monitored model (φ, cm, mm) such that φ contains both a global timer t, and also a local clock c that is reset at each control step. Concretely, $\varphi \equiv \texttt{init} \rightarrow [\{\texttt{ctrl};\texttt{plant}\}^*]\texttt{safe}$ where*

- *$\texttt{init} \models c = 0 \land t = 0$,*
- *$\texttt{ctrl}$ does not change t and also sets $c = 0$ for every input state, and*
- *$\texttt{plant}$ contains the equations $c' = 1, t' = 1$.*

8.4.2 Model Update Learning: The Basic Algorithm

Model update learning, or μlearning, leverages verification-preserving model updates to maintain safety while selecting an appropriate environmental model.

Definition 18 (μlearning Process)**.** *A learning process P_M for a finite set of monitored models M is defined as a tuple of countable sequences $(\boldsymbol{U}, \boldsymbol{S}, \boldsymbol{Mon})$ where $\boldsymbol{U}$ is a sequence of actions in a finite set of actions A, elements of the sequence $\boldsymbol{S}$ are states, and $\boldsymbol{Mon}$ are monitored models with $\boldsymbol{Mon}_0 = M$. Let*

$$specOK_m(\boldsymbol{U}, \boldsymbol{S}, i) \equiv cm(\boldsymbol{S}_i, \boldsymbol{U}_i) \vee \neg mm(\boldsymbol{S}_{i-1}, \boldsymbol{U}_{i-1}, \boldsymbol{S}_i)$$

such that cm and mm are the monitors for the model m. Let $specOK$ always returns true for $i = 0$.

*A μ**learning process** is a learning process satisfying the following additional conditions:*

- *action availability: in each state $\boldsymbol{S}_i$ there is at least one action u such that for all $m \in \boldsymbol{Mon}_i$, $u \in \mathtt{specOK}_m(\boldsymbol{U}, \boldsymbol{S}, i)$,*
- *actions are safe for all feasible models:*
  $\boldsymbol{U}_{i+1} \in \{u \in A \,|\, \forall(\varphi, cm, mm), (\varphi, cm, mm) \in \boldsymbol{Mon}_i \rightarrow cm(\boldsymbol{S}_i, u)\}$
- *feasible models remain in the feasible set:*
  If $(\varphi, cm, mm) \in \boldsymbol{Mon}_i$ and $mm(\boldsymbol{S}_i, \boldsymbol{U}_i, \boldsymbol{S}_{i+1})$ then $(\varphi, cm, mm) \in \boldsymbol{Mon}_{i+1}$.

Note that μlearning processes are defined over an environment $E : A \times S \rightarrow S$ that determines the sequences $\mathbf{U}$ and $\mathbf{S}$[5], so that $\mathbf{S}_{i+1} = E(\mathbf{U}_i, \mathbf{S}_i)$.

In our algorithms, the set $\mathbf{Mon}_i$ never retains elements that are inconsistent with the observed dynamics at the previous state.

Notice that the safe actions constraint is not effectively checkable without extra assumptions on the range of parameters. Two canonical choices are discretizing options for parameters or including an effective identification process for parameterized models.

Our safety theorem focuses on time-aware μlearning processes, i.e., those whose models are all time-aware; similarly, a finite action-space μlearning process is a μlearning process in which all models $m \in M$ have a finite action-space. The basic correctness property for a μlearning process is the safe reinforcement learning condition: the system never takes unsafe actions.

Definition 19 (μlearning process with an accurate model)**.** *Let $P_M = (\boldsymbol{S}, \boldsymbol{U}, \boldsymbol{Mon})$ be a μlearning process and let E be a mapping from actions (i.e., deterministic loop-free and ODE-free programs) and states (i.e., mappings from variables to values) to states. We say that an element $m^* = (\varphi^*, cm^*, mm^*) \in \boldsymbol{Mon}_0$ is an accurate model for E if it has the properties following:*

1. *$\varphi^* \equiv (\mathtt{init}_m \rightarrow [\{\mathtt{ctrl}_m; \mathtt{plant}_m\}^*]\mathtt{safe})$,*
2. *$\vdash_{\mathsf{dL}} \varphi^*$,*
3. *$(s, u(s)) \in [\![\mathtt{ctrl}_m]\!]$ implies $(u(s), E(u, s)) \in [\![plant]\!]$ for E projected to the state variables occurring in m^*, and*
4. *$mm^*(\boldsymbol{S}_i, \boldsymbol{U}_i, \boldsymbol{S}_{i+1})$ for all $i \geq 0$ on which the sequences are defined.*

If exactly one such element exists in $\boldsymbol{Mon}_0$, we call that element m^* the distinguished *and/or* accurate *model and say that the process P_M is accurately modeled with respect to E.*

[5]Throughout the chapter, we denote by $\mathbf{S}$ a specific sequence of states and by S the set of all states

The mapping E is often called an environment. We will often elide the environment for which the process P_M is accurate when it is obvious from context.

Theorem 5 (Safety)**.** *If P_M is a μlearning process with an accurate model, then $\boldsymbol{S}_i \models \texttt{safe}$ for all $0 < i < |\boldsymbol{S}|$.*

Proof. Let m^* be the distinguished model for $P_M = (\mathbf{S}, \mathbf{U}, \mathbf{Mon})$. Proceed by induction on the length of $\mathbf{S}$ with the hypothesis that $\mathbf{S}_i \models \texttt{safe}$ and $m^* \in \mathbf{Mon}_i$. Let E be the environment with respect to which P_M is defined.

By the definition of a μlearning process with an accurate model, $\mathbf{S}_0 \models \texttt{safe}$ and $m^* \in \mathbf{Mon}_0$.

Now, assume $\mathbf{S}_i \models \texttt{safe}$ and $m^* \in \mathbf{Mon}_i$. By the definition of a μlearning process, we therefore know that either

$$\mathit{cm}^*(\mathbf{S}_{i-1}, \mathbf{U}_{i-1}(\mathbf{S}_{i-1}))$$

or else

$$\neg \mathit{mm}^*(\mathbf{S}_{i-1}, \mathbf{U}_{i-1}(\mathbf{S}_{i-1}), \mathbf{S}_i)$$

However, the second cannot be the case due to the hypothesis that m^* is the accurate model (Def. 19). Therefore, it must be that $(\mathbf{S}_i, \mathbf{U}_i(\mathbf{S}_i)) \in [\![\texttt{ctrl}_{m^*}]\!]$. From this fact, Def. 19, and the fact that m^* is distinguished, it follows that $(\mathbf{U}_i(s), E(\mathbf{U}_i, \mathbf{S}_i)) \in [\![\mathit{plant}]\!]$. This, together with the fact that $\vdash_{\mathsf{d}\mathcal{L}} \varphi^*$, the shape assumption on φ^*, and the soundness of $\mathsf{d}\mathcal{L}$, implies that $\mathbf{S}_{i+1} \models \texttt{safe}$.

What remains to be shown is that $m^* \in \mathbf{Mon}_{i+1}$. Notice that $\mathit{mm}^*(\mathbf{S}_i, \mathbf{U}_i(\mathbf{S}_i), \mathbf{S}_{i+1})$ must be true because m^* is the accurate model. By the inductive hypothesis know also that $m^* \in \mathbf{Mon}_i$. By definition, *feasible models remain in the feasible set* (Def. 18); i.e., the above two facts establish that $m^* \in \mathbf{Mon}_{i+1}$. □

Listing 8.2 defines a μlearning algorithm that produces a μlearning process. The inputs are:

1. A set of models `M` each with a method $\texttt{models} : S \times A \times S \to \mathbb{B}$ which implements the evaluation of its model monitor in the given previous and next state and actions and a method $\texttt{safe} : S \times A \to \mathbb{B}$ which implements evaluation of its controller monitor,
2. an action space `A` and an initial state $\texttt{init} \in S$,
3. an environment function $\texttt{env} : S \times A \to S \times \mathbb{R}$ that computes state updates and rewards in response to actions, and
4. a function $\texttt{choose} : \wp(A) \to A$ that selects an action from a set of available actions and `update` updates a table or approximation. Our approach is generic and works for any reinforcement learning algorithm; therefore, we leave these functions abstract.

This algorithm augments an existing reinforcement learning algorithm, defined by `update` and `choose`, by restricting the action space at each step so that actions are only taken if they are safe with respect to *all* feasible models. The feasible model set is updated at each control set by removing models that are in conflict with observed data.

The μlearning algorithm rules out incorrect models from the set of possible models by taking actions and observing the results of those actions. Through these experiments, the set of relevant

models is winnowed down to either the distinguished correct model m^*, or a set of models M^* containing m^* and other models that cannot be distinguished from m^*.

Listing 8.2: The Basic μlearning Algorithm.

```
def μlearn(M,A,init,env,choose,update):
  s_pre = s_curr = init
  act   = None
  while(not done(s_curr)):
    if act is not None:
      M = {m ∈ M : m.models(s_pre,act,s_post)}
    avail = {a ∈ A : ∀ m ∈ M, m.safe(a)}
    act = choose(avail)
    s_pre = s_curr
    (s_curr, reward) = env(s_curr, act)
    update(s_pre, act)
```

8.4.3 Active Verified Model Update Learning

Clever μlearning takes actions that help rule out models $m \in M$ that are not m^*. Removing models from the set of possible models weakens the monitoring condition, allowing less conservative and more accurate control decisions.

This section introduces a refinement of the μlearning algorithm that prioritizes differentiating between models by active learning. Instead of choosing a random safe action, the refined algorithm prioritizes actions that differentiate between available models. We begin by explaining what it means for an algorithm to perform good experiments.

Definition 20 (Active Experimentation)**.** *A μlearning process with an accurate model m^* has locally active experimentation if there exists an action a that is safe for all feasible models (see Def. 18) in state s_i, and if taking action a results in the removal of m from the model set, then $|M_{i+1}| < |M_i|$.*

Definition 21 (Distinguishing Actions)**.** *Consider a μlearning process $(\boldsymbol{U}, \boldsymbol{S}, \boldsymbol{Mon})$ with a model m^* that is accurate with respect to E (see Def. 19). An action a distinguishes m from m^* if there is some $i > 0$ such that $a = \boldsymbol{U}_i$, $m \in \boldsymbol{Mon}_i$ and $m \notin \boldsymbol{Mon}_{i+1}$.*

The *active μlearning algorithm*, presented in Listing 8.3, uses model monitors to select distinguishing actions, thereby performing active experiments which winnow down the set of feasible models. The inputs to `active-μlearn` are the same as those to Listing 8.2 with two additions:

- models are augmented with an additional prediction method `p` that returns the model's prediction of the next state given the current state, a candidate action, and a time interval.
- An elimination rate `er` is introduced, which plays a similar role as the classical explore-exploit rate except that we are now choosing whether to insist on choosing a good experiment.

The `active-μlearn` algorithm is guaranteed to make some progress toward winnowing down the feasible model set whenever $0 < \texttt{er} < 1$.

Listing 8.3: μlearning with Active Experimentation.

```
def active-μlearn(···):
  s_pre = s_curr = init
  act   = None
  while(not done(s_curr)):
    if act is not None:
      M = {m ∈ M :
            m.models(s_pre,act,s_post)}
    avail = {a ∈ A : ∀ m ∈ M. m.safe(a)}
    if rand() > er:
      avail = {a ∈ avail : ∃ m,n ∈ M m.p(curr,a) ≠ n.p(curr,a)}
    act = choose(avail)
    update(s_pre, act)
```

Theorem 6. *Let $P_M = (\boldsymbol{S}, \boldsymbol{U}, \boldsymbol{Mon})$ be a finite action space μlearning process with an accurate model m^*. Then $m^* \in \boldsymbol{Mon}_i$ for all $0 \leq i \leq |\boldsymbol{Mon}|$.*

Proof. Let $m^* = (\varphi^*, \boldsymbol{cm}^*, \boldsymbol{mm}^*)$ and let E be the environment over which P_M is defined. By Def. 19,

$$m^* \in \mathbf{Mon}_0$$

Suppose now that $m^* \in \mathbf{Mon}_i$ for some $i \geq 0$. The crucial observation is that $\boldsymbol{mm}^*(\mathbf{S}_i, \mathbf{U}_i, \mathbf{S}_{i+1})$, which will directly imply that $m^* \in \mathbf{Mon}_{i+1}$ due to the fact that feasible models remain in the feasible set (Def. 18).

So, it suffices to show that $\boldsymbol{mm}^*(\mathbf{S}_i, \mathbf{U}_i, \mathbf{S}_{i+1})$. However, notice that this follows directly from the definition of an accurate model (Def. 19). □

Theorem 7. *Let P_M be a finite action space μlearning process with an accurate model m^*.[6] If each model $m \neq m^*$ has in each state s an action a_s that is safe for all models and distinguishes m from m^*, then $\lim_{i\to\infty} \boldsymbol{Pr}(m \notin M_i) = 1$.*

Proof. Consider $P_M = (\mathbf{S}, \mathbf{U}, \mathbf{Mon})$. Note that $m^* \in \mathbf{Mon}_i$ for all $0 \leq i \leq |\mathbf{Mon}|$ is directly implied by Theorem 6. Let $k = |M_i| - 1$ be the number of non-m^* elements in $\mathbf{Mon}_i$. It suffices to show that $\lim_{i\to\infty} \Pr(k = 0) = 1$. Notice that because each $m \neq m^*$ is falsifiable at step i, the probability that $k-1$ non-m^* elements remain in after n steps is $\sum_{j=0}^{k-1} \mathtt{er}^j(1-\mathtt{er})^{n-j}$ which tends to 0 as $n \to \infty$. □

Corollary 4. *Let $P_M = (\boldsymbol{S}, \boldsymbol{U}, \boldsymbol{Mon})$ be a finite action space μlearning process generated by an environment E and with an accurate model m^*. If each model $m \neq m^*$ has in each state s an action a_s that is safe for all models and distinguishes m from m^*, then **Mon** converges to $\{m^*\}$ a.s.*

Proof. The conclusion follows from Theorem 7 as long as m^* is never removed. Proceed by induction on the length of **Mon**. In the base case, recall that $m^* \in \mathbf{Mon}_0$. Suppose, then, that $m^* \in \mathbf{Mon}_i$. We assume $m^* = (\varphi^*, \boldsymbol{cm}^*, \boldsymbol{mm}^*)$ is accurately modeled with respect to E, which by Def. 19 implies $\boldsymbol{mm}^*(\mathbf{S}_i, \mathbf{U}_i, \mathbf{S}_{i+1})$ for all elements in the sequences **U** and **S**. Therefore,

[6] Recall that μlearning is initialized with a finite set of models.

$\mathbf{Mon}_{i+1}$ must contain m^* by the inductive hypothesis applied to the third condition ("feasible models remain in the feasible set") of Def. 18. □

8.4.4 Limitations of Locally Good Experimentation

Although locally active experimentation is not strong enough to ensure that P_M eventually converges to a minimal set of models[7], our experimental validation demonstrates that this heuristic is none-the-less effective on some representative examples of model update learning problems.

Example 7 (limitation of locally active experimentation)**.** *Consider the following three models:*

$$x \geq 0 \wedge t = 0 \rightarrow [\texttt{ctrl}; \{x' = x, t' = 1\}]x \geq 0 \quad (m^*)$$
$$x \geq 0 \wedge t = 0 \rightarrow [\texttt{ctrl}; \{x' = 0, t' = 1\}]x \geq 0 \quad (m_1)$$
$$x \geq 0 \wedge t = 0 \rightarrow [\texttt{ctrl}; \{x' = 5, t' = 1\}]x \geq 0 \quad (m_2)$$

where $\texttt{ctrl} \equiv x := 0 \cup ?t = 0; x := 1$*. According to Def. 20, choosing the action* $x := 0$ *is an active experiment because this action will distinguish between the second and third models.*

Unfortunately, taking this action will result in the next control step happening in a state where $t \neq 0$*, meaning that the only safe action is* $x := 0$ *for the rest of time. Obviously, the first and third models are not possible to distinguish between using the only remaining action* $x := 0$*.*

This limitation is no counter-example to the above theorems because states where $t \neq 0$ do not have any experiments that distinguish m_1 from m^*.

8.5 Experimental Validation

The μlearning algorithms introduced in this chapter are designed to answer the following question: given a set of possible models that contains the one true model, how can we *safely* perform a set of experiments that allow us to efficiently discover a minimal safety constraint? In this section we present several experiments which demonstrate the use of μlearning in safety-critical settings.

Overall, these experiments empirically validate our theorems and further demonstrate that the μlearning framework is useful even when the assumptions underpinning our theoretical results are violated.

Our simulations use a conservative discretization of (8.1) and we translated monitoring conditions by hand into Python from ModelPlex's C output [133]. Although we evaluate our approach in a research prototype implemented in Python for the sake of convenience, there is a verified compilation pipeline for models implemented in $\mathsf{d}\mathcal{L}$ that eliminates uncertainty introduced by discretization and hand-translations [26].

8.5.1 Adaptive Cruise Control

Adaptive Cruise Control (ACC) is a common feature in new cars. ACC systems change the speed of the car in response to the changes in the speed of traffic in front of the car; e.g., if the car in

[7]Consider e.g., $x \geq 0 \wedge t = 0 \rightarrow [\{\{?t = 0; x := 1 \cup x := 0\}; \{x' = F, t' = 1\}\}^*]x \geq 0$ for $F = 0, 5, x$.

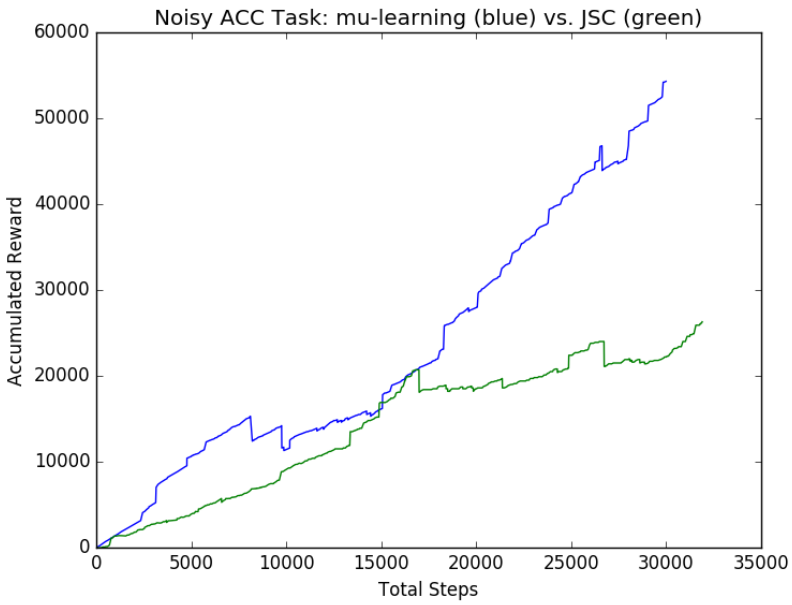

Figure 8.1: Comparison of Justified Speculative Control and μlearning on Adaptive Cruise Control. The cumulative reward obtained by Justified Speculative Control [62] (green) and μlearning (blue) during training over 1,000 episodes with each episode truncated at 100 steps.

front of an ACC-enabled car begins slowing down, then the ACC system will decelerate to match the velocity of the leading car.

Our first set of experiments consider a simple linear model of ACC, as used in previous chapters. The generic constant disturbance update computes the following model update, in which the acceleration set-point is perturbed by an unknown parameter p; i.e., the relative position of the two vehicles is determined by the equations

$$\text{pos}'_{\text{rel}} = \text{vel}_{\text{rel}}, \text{vel}_{\text{rel}} = \text{acc}_{\text{rel}} + p \tag{8.1}$$

where p has a constant but unknown value.

In [62], the authors consider the collision avoidance problem when a noise term is added so that $\text{vel}'_{\text{rel}} = p\text{acc}_{\text{rel}}$. We are able to outperform the approach in [62] by combining the **Add Noise Term** and **Parameter Instantiation** updates; we outperform in terms of both avoiding unsafe states and in terms of cumulative reward. These two updates alloww us to insert a multiplicative noise term p into these equations, synthesize a provably correct controller, and then choose the correct value for this noise term at runtime. Unlike [62], μlearning avoids all safety violations.

The graph in Fig. 8.5.1 compares the Justified Speculative Control approach of [62] to our approach in terms of cumulative reward; in addition to substantially outperforming the JSC algorithm of [62], μlearning also avoids 204 more crashes throughout a 1,000 episode training process.

8.5.2 Pedestrian Crossing

The second experiment tests the μlearning algorithm without using any model updates. This experiment the problem of anticipating whether a pedestrian will move into a cross-walk. The system must differentiate between two models of the pedestrian, both of which are given a priori by the system designer. In the first model, m_0, a pedestrian at position (ped_x, ped_y) continues to walk along a sidewalk indefinitely (ped_x never changes). In the second model, m_1, a pedestrian enters the crosswalk at some point between $c_{min} \leq ped_y \leq c_{max}$ (the boundaries of the crosswalk). The overall task is visualized in Fig. 8.2.

The μlearning algorithm exhibits exactly the correct course of action. Before ped_y exceeds c_{max} the agent is unable to rule out either model and therefore maintains safety wrt $M = \{m_0, m_1\}$ by braking sufficiently often to maintain separation in the y coordinate. As soon as the pedestrian passes c_{max} the agent notices that m_1 is no longer feasible, discards the model from the feasible set, and accelerates through the crosswalk.

8.5.3 SCUBA Diving

SCUBA diving is a safety-critical activity. SCUBA dive computers help divers remain safe by alerting the diver when oxygen supply is running low. Unfortunately, monitoring the tank directly requires the use of expensive sensors and underwater transmitters that are often too expensive for hobbyist use. Fortunately, it is possible to monitor oxygen consumption indirectly using a cheaper heart rate monitor. A simple model relating heart rate to oxygen consumption is given by the system of differential equations:

$$hr' = b(hr_{ss} - hr), tank' = -\tau hr$$

where *tank* is the amount of oxygen remaining in the diver's tank, hr_{ss} is a steady-state heart rate whose value is determined by the diver's level of exertion, *hr* is the diver's heart rate, and b, τ are parameters specific to the diver's physiology. We have designed a safe SCUBA ascent protocol using these equations[8].

We consider the problem of identifying the true values of the parameters b and τ. This problem is different from the cruise control and pedestrian crossing problems in two ways:

1. these parameters may take on values from a compact domain meaning that there is an infinite set of possible models, and
2. identifying the true model requires performing a regression to fit observed data to parameter values.

This problem is characterized as a model learning problem by introducing a parameter instantiation model update. This update allows parameters to take on any value from within a compact domain. We consider two variations on this algorithm.

In the first variation, we introduce compact domains for each parameter and use the automatic compact parameter instantiation update to generate many feasible models (one for each parameter instantiation). The system then chooses the correct model from this finite set of models.

[8] available at github.com/nrfulton/scuba-release

In the second variation, regression, instead of falsification, is used to select parameter values at runtime. We have evaluated this algorithm on a set of feasible values for each parameter, and find that the controller requires very few data points to discover accurate parameter values. This experiment is particularly interesting because it replaces a discretization step with regression. We expand on this idea in the next chapter.

8.5.4 Mode Switching

Model update learning can be extended to provide formal guarantees for hierarchical reinforcement learning algorithms [17]. If each feasible model m corresponds to a subtask, and if and all states satisfying termination conditions for subtask m_i are also safe initial states for any subtask m_j reachable from m_i, then μlearning directly supports safe hierarchical reinforcement learning by re-initializing M to the initial (maximal) model set whenever reaching a termination condition for the current subtask.

The mode switching task has four separate models, one of which encodes two possible behaviors. The first three models capture the behavior of the car in the intersection, which may either move straight or make a left-hand turn. The left-hand turn is modeled as circular motion, and the critical invariant which is monitored at runtime is the fact that the car under control chooses to remain static and is therefore completely separated from the circle along which the left-turning car will move. This model is only valid until the left-turning car enters the intersection. The other two models allow the car to move along the x-axis after having left the intersection, or along the y-axis through the intersection. The final model for the mode switching task capture the behavior of the pedestrian. The pedestrian may either choose whether to enter the crosswalk. A single model captures the pedestrian's behavior (which may also be modeled using two separated models, one for each behavior).

We implemented a variant of μlearning that performs this re-initialization and validated this algorithm in an environment where a car must first navigate an intersection containing another car and then must avoid a pedestrian in a crosswalk (as illustrated in Fig. 8.5.4). This example focuses on demonstrating that safe hierarchical reinforcement learning is simply safe μlearning with safe model re-initialization. The switching conditions for these models were constructed manually. We also constructed the model monitors by hand instead of using those generated by ModelPlex (although the ModelPlex monitoring conditions could also be used with additional engineering effort).

8.6 Related Work on Safe Off-Model Learning

Related work falls into three broad categories: safe reinforcement learning, runtime falsification, and program synthesis

Related work on Safe Reinforcement Learning

Our approach toward safe reinforcement learning differs from existing approaches that do not include a formal verification component (e.g., as discussed in the introductory chapter of Part II

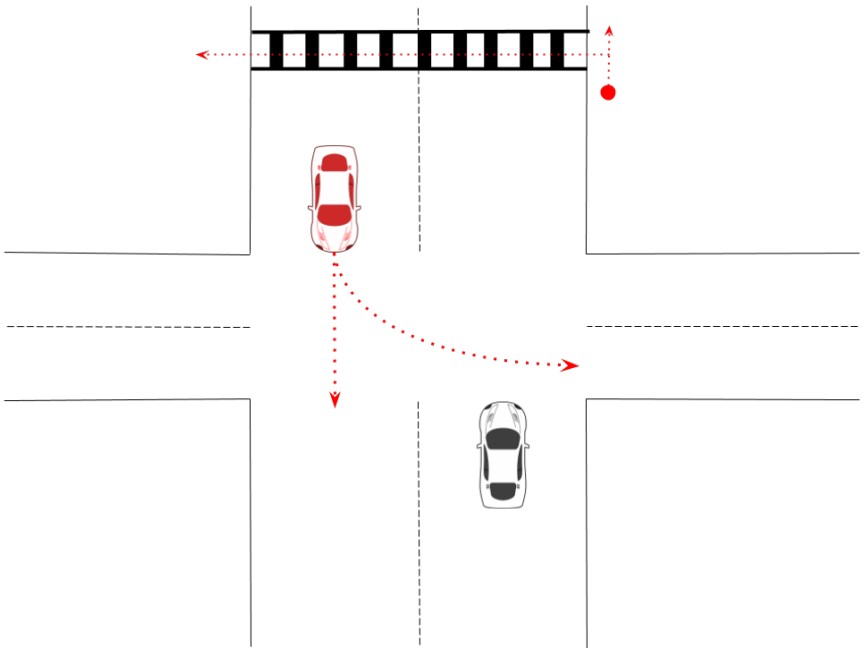

Figure 8.2: A Visualization of the Intersection and Pedestrian Task.

of this thesis) because we focused on *verifiably* safe learning; i.e., instead of relying on oracles or conjectures, constraints are derived in a provably correct way from formally verified safety proofs. The difference between verifiably safe learning and safe learning is significant, and is equivalent to the difference between verified and unverified software.

Alshiekh et al. and Hasanbeig et al. propose approaches based on Linear Temporal Logic [5, 83]. Alshiekh et al. synthesize monitoring conditions based upon a safety specification and an environmental abstraction. In terms of this formalism, the goal of off-model learning is to systematically expand the environmental abstraction based upon both design-time insights about how the system's behavior might change over time and also based upon observed data at runtime. Jansen et al. extend the approach of Alshiekh et al. by observing that constraints should adapt whenever runtime data suggests that a safety constraint is too restrictive to allow progress toward an over-arching objective [100]. Herbert et al. address the related problem of safe motion planning by using offline reachability analysis of pursuit-evasion games to pre-compute a safety bubble for the online planner [60, 88].

The above-mentioned approaches have an implicit or explicit environmental model. Even when these environmental models are accurate, reinforcement learning is still necessary because these models focus exclusively on safety and are often nondeterministic. Resolving this non-determinism in a way that is not only safe but is also effective at achieving other high-level objectives is a task that is well-suited to reinforcement learning.

Unlike the above-mentioned work on verifiably safe RL, we are interested in how to provide formal safety guarantees even when there is not a single accurate model available at design time. Achieving this goal required two novel contributions. First, we contributed a way to generate a robust set of feasible models given some combination of an initial model and data on previous

runs of the system (because formal safety guarantees are stated with respect to a model). Given such a set of feasible models, we then contributed an algorithm that learns how to safely identify which model is most accurate so that the system is not over-constrained at runtime.

Thus, compared to prior work on applying verification and reachability analysis tools to safe reinforcement learning [5, 60, 62, 83, 88, 100], our approach is the first to combine model synthesis at design time with model falsification at runtime so that safety guarantees capture a wide range of possible futures instead of relying on a single accurate environmental model. Safe off-model learning is an important problem because autonomous systems must be able to cope with unanticipated scenarios, and ours is the first approach toward verifiably safe off-model learning.

To achieve these goals, we built on the safe learning work and theorem proving technology discussed in previous chapters. In particular, the approach discussed in this chapter required the following combination of capabilities unique to KeYmaera X and Bellerophon.

1. A modeling language that combines combine discrete and continuous dynamics so that we could produce explainable models of system dynamics (e.g., systems of differential equations as opposed to large state machines)
2. The ability to systematically modify models in a way that separates assumptions (where it is reasonable to apply inductive program synthesis techniques) from derived control laws and user-provided safety specifications (where deductive synthesis, bootstrapped with the output of an inductive synthesis phase, makes much more sense).
3. The ability to systematically modify proofs alongside their models.
4. A monitor synthesis algorithm capable of using information latent in safety proofs.

Related Work on Safe Model-Free Reinforcement Learning

Several recent papers focus on providing safety guarantees for model-free reinforcement learning. Trust Region Policy Optimization [163] defines safety in terms of monotonic policy improvement, a much weaker notion of safety than the constraints guaranteed by our approach. Constrained Policy Optimization [2] extends TRPO with guarantees that an agent nearly satisfies safety constraints during learning. Our approach is model-based instead of model-free, and instead of focusing on learning safely without a model we focus on identifying accurate models from data obtained both at design time and at runtime. Learning concise dynamical systems representations has one substantial advantage over model-free methods: safety guarantees are stated with respect to an explainable model that captures the safety-critical assumptions about the system's dynamics. Synthesizing explainable models is important because safety guarantees are always stated with respect to a model; therefore, engineers must be able to understand inductively synthesized models in order to understand what safety properties their systems do (and do not) ensure.

Related Work on Program Synthesis

Our approach includes a model synthesis phase that is closely related to program synthesis and program repair algorithms [106, 121, 159]. Relative to work on program synthesis and repair,

VPMUs are unique in several ways. We are the first to explore hybrid program repair. Our approach combines program verification with mutation. We treat programs as *models* in which one part of the model is varied according to interactions with the environment and another part of the model is systematically derived (together with a correctness proof) from these changes. This separation of the dynamics into inductively synthesized models and deductively synthesized controllers enables our approach toward using programs as representations of dynamic safety constraints during reinforcement learning.

Although we are the first to explore hybrid program repair, several researchers have explored the problem of synthesizing hybrid systems from data. This work is closely related to our **Learn Linear Dynamics** update. Sadraddini and Belta provide formal guarantees for data-driven model identification and controller synthesis [162]. Relative to this work, our **Learn Linear Dynamics** update is continuous time, synthesizes a computer-checked correctness proof (as opposed to a least violation criterion implied by the synthesis algorithm both not independently checked by a theorem prover), and does not require dynamics to be compact, bounded, and locally connected. Unlike Asarin et al. [14], our full set of model updates is capable of synthesizing nonlinear dynamical systems from data (e.g., the static $\rightarrow$ circular update) and produces computer-checked correctness proofs for permissive controllers.

8.7 Conclusion

This chapter introduced verification-preserving model updates, an approach toward learning safely even when a single, accurate environmental model is not available. We achieve this goal by updating the system's model of the world in response to observations without losing formal safety guarantees. The next chapter builds upon the VPMU framework developed in this chapter and, in particular, expains our linear model synthesis algorithm in more detail.

Chapter 9

Hybrid Program Synthesis

Classical approaches toward control engineering are model-based. Engineers write down a model of the system under control, derive control laws that satisfy safety and efficiency properties, and then implement a controller in a general purpose programming language. This is the approach that KeYmaera X [64], ModelPlex [133], and the VeriPhy pipeline [26] were originally designed to support. Environmental models might take the form of a system of differential equations, a system of difference equations, a state machine, or some form of automata. The safety-critical nature of many controls problems has inspired the development of formal methods for proving that these model-based controllers are safe.

The justified speculative control (JSC) and model update learning algorithms demonstrate that model-based controllers are amenable to formal verification and that we can automatically generate robust generalizations of model-based controllers that preserve proofs. JSC and μlearning both assume access to a more-or-less accurate model, even if there are many other plausible models and even if reality occasionally strays from the model.

JSC and μlearning start with models and move toward reality by modifying and/or leveraging these models. This chapter instead starts from observed data, generates a model that fits the data, and then generates both controllers and correctness proofs for the resulting model. Our approach synthesizes provably correct controllers from a global safety specification after inductively synthesizing an environmental model from raw data.

The previous chapter discussed the **Learn Linear Dynamics** update, in which the set of models provided to the μlearning may rely on experimental data available at design time. This chapter further develops this approach by considering the problem of proof-preserving inductive program synthesis for hybrid dynamical systems. We discuss the **Learn Linear Dynamics** update in more detail and briefly discuss how this idea may be extended to nonlinear systems.

Synthesizing verified models at design-time from data has three desirable properties:

1. The engineer does not need to provide an explicit environmental model of the system. Instead, we generate models from experimental data provided at design-time.
2. The environmental models that we generate are **explainable**. Unlike most model-free approaches, our approach provides engineers with human-readable models in the form of hybrid programs.
3. We are able to automatically synthesize and **formally verify** control laws based upon these

explainable environmental models.

Concretely, the problem solved in this chapter is take as input a safety specification, a finite set of control choices (a.k.a. actions as discussed in Chapter 8), and a global safety specification expressed as a quantifier-free arithmetic formula and produce as output:

1. a system of differential equations `odes` that fit some experimental data, and
2. tests (a.k.a. guards) g_i for each control action u_i together with initial conditions `init` such that $\texttt{init} \rightarrow [\{\texttt{ctrl};\texttt{odes}\}^*]\texttt{safe}$.

The remainder of this chapter is organized as follows. Section 9.1 presents our methodology for environments that can be accurately as a system of ordinary differential equations with nilpotent constant coefficients. In this setting we are able to state and prove both soundness and completeness results for the synthesis algorithm. We also present several experiments that demonstrate important design criteria for the cost function that guides the synthesis algorithm. Section 9.2 then considers nonlinear environmental dynamics by leveraging *verification-preserving model updates*, a mechanism for shaping the model search space in a way that guarantees we will be able to synthesize a safe controller.

9.1 Proof-Generating Synthesis for Linear Systems

This section tackles the problem of synthesizing proven-correct control software from a set of experiments, a finite set of control choices expressed as a sequence of assignments, and a global safety specification. The algorithm has three distinct phases:

1. The model identification phase during which a set of differential equations c is inductively synthesized from data.
2. The controller synthesis phase, which takes as input the differential equations c from the previous step and the desired global safety constraint *safe* and generates maximally permissive guards φ_i are computed for each control program u_i; and
3. The verification phase, during which a proof of $\varphi_{init} \rightarrow [\{\{\cup_i ?\varphi_i; u_i\}; c\}^*]$*safe* is generated. where φ_{init} are a set of initial conditions generate from the output of step 2 and *safe* is the given safety constraint.

We begin with an overview of the algorithm and then discuss, in detail, how each phase of this algorithm is implemented.

9.1.1 Overview of of Synthesis Process for Linear Systems

The **Learn Linear Dynamics** update discussed in the previous chapter takes as input: (1) data from previous executions of the system, and (2) a desired safety constraint. From these two inputs, the update computes a set of differential equations `odes` that comport with prior observations, a corresponding controller `ctrl` that enforces the desired safety constraint with corresponding initial conditions `init`, and a Bellerophon tactic `prf` which proves $\texttt{init} \rightarrow [\{\texttt{ctrl};\texttt{odes}\}^*]\texttt{safe}$ For example, inputs for the system described by Example 2 would be a sequence of records with the form:

```
{accel:  2, T: 1, pos:  {0,2,6,12,...}, v:  {1,3,5,7,...}}
```

and `safe`: *pos* < *obstaclePos*.

This update is implemented as an exhaustive search of the set of fully parametric ODES whose solutions exist in a decidable fragment of real arithmetic. By fully parametric, we mean that all coefficients are stated as parameters instead of as concrete values. For example, instead of $x' = v, v' = 12a$ the update will simply return $x' = p_0 v, v' = p_1 a$. The update searches for the parametric model(s) that sufficiently minimize a cost function defined by:

$$\textit{cost}(\texttt{odes}) = f(\textit{incorrect}(\texttt{experiments}, \texttt{odes}), \textit{complexity}(\texttt{odes}))$$

where *incorrect* counts the number of elements in each execute trace that do not comport with the solution to *incorrect* and *complexity* is a model complexity measure (e.g., a count of the number of nodes in the ODE's AST weighted by the degree of the highest total degree polynomial occurring in the right-hand side of the differential equations).

Insisting on parametric differential equations has several advantages. The search is guaranteed to terminate and return an optimal model or set of models under mild assumptions on the shape of the *cost* function. Using parameters instead of concrete values also helps avoid overfitting to the particular execution traces available at design time. However, these advantages come with a complication: computing *incorrect* for parametric models requires instantiating these parameters. To address this complication, the **Learn Linear Dynamics** update uses a least squares regression with a subset of the I/O trace data to fit parameters for each trace, and then computes *incorrect* using the remaining trace data.

So far, we have explained how **Learn Linear Dynamics** chooses an ODE (or set of candidate ODEs) by searching the space of parametric ODEs for systems that minimize a cost function defined in terms of model accuracy and model complexity. However, we have not explained how a corresponding controller is computed or how a proof is synthesized. We now turn our attention to these two problems.

Controllers are computed by searching for an action[1] that is globally safe from some subset of state space. Formally, there should be some discrete program u which chooses actuator inputs such that $[\{u; \texttt{odes}\}^*]\texttt{safe} \leftrightarrow \varphi$ for some quantifier-free formula φ of real arithmetic[2]. Every other control choice q has a guard g_q which ensures that φ is true after taking the action g from the current state; i.e., the final controller has the form $\texttt{ctrl} \equiv u_s \cup ?\varphi(\textit{soln}(x_0, u_1, T)); u_1) \cup \cdots \cup ?\varphi(\textit{soln}(x_0, u_n, T)); u_n$ where $u_1 \ldots u_n$ is the set of available actions expressed as loop-free deterministic discrete programs (e.g., in the case of Example 2, $accel := A$ and $accel := -B$) every other choice of control inputs must stay within this safe subset of state space where the distinguished safe action can maintain safety invariants The formula φ provides a loop invariant so that the tactic `unfold; loop(`φ`,1)<(QE,QE,master)` will prove the formula $\varphi \rightarrow [\{\texttt{ctrl};\texttt{plant}\}^*]\texttt{safe}$ The `master` tactic is an automated tactic that performs symbolic decomposition of the loop-free control program, solves the system of differential equations using the KeYmaera X axiomatic ODE solver, and then resolves the resulting quantified arithmetic using a real arithmetic decision procedure.

[1] i.e., a mapping from controllable/actuated variables to concrete numeric values.

[2] KeYmaera X can automatically reduce this formula to a quantified arithmetic formula, and quantifier elimination for the resulting arithmetic is decidable because of our restriction on the set of linear ODEs considered by this update

9.1.2 Model Identification

Model identification is the problem of generating a dynamical system – e.g., a set of ordinary differential equations – from data. We solve this problem using a combination of inductive program synthesis and regression. The synthesis algorithm produces a set of parameteric ODEs as candidates. A regression algorithm is then used to instantiate these parameteric models so that each model may be numerically evaluated against a set of evaluation data. This evalution results in a score, which is used by the syntheizer to select the next set of parametric models. The process repeats until reaching a finite horizon. Under very mild assumptions, this algorithm will converge.

The basic algorithm is presented in Fig. 9.1.2. We use linear least squares for fitting parameters. The interesting design points in this algorithm are selecting the `score` and `newModels` functions. The choices are related, since `newModels` has access to computed scores and will typically use these scores to select the next set of models.

The Baisc Inductive Synthesis Algorithm.

```
1  //An experiment maps the initial conditions (i.e., control inputs) and an
       independent variable t:ℝ⁺ to a state S.
2  type Experiment = U × ℝ⁺ × S
3  inputs:
4    es: Set[Experiment], a set of experiments.
5    n : ℕ, the bound/timeout on the synthesis process.
6    ms: List[ODESystem], a (possibly empty) set of initial models.
7    split: A function that splits a set into two parts.
8    score: Set[Experiment] × ODESystem → ℝ, a scoring function defining the
         quality of the model. Lower scores are better.
9    newModels: Set[ODESystem × ℝ] → Set[ODESystem]
10   threshold: ℕ, a cutoff point for models' scores.
11 es1,es2 = split(es)
12 while i < n:
13   regression, eval = split(es1)
14   // fs = ms with parameters fit using regression.
15   fs = ms.map(mᵢ => fit-params(regression, mᵢ))
16   // associate each fitted model fᵢ with a score.
17   fs.zip( fs.map(fᵢ => score(eval, fᵢ)) )
18   // remove all models above a threshold and then generate new models.
19   // (note: lower scores are better.) There is an implicit mapping from
20   ms = newModels(fs.filter((fᵢ, score) => score < threshold).map( (fᵢ,score)
         => fᵢ)
21 return ms.min(mᵢ => score(es2, mᵢ))
```

Line 13 first splits the data into two sets; the first set of data is used to fit parameters and the second is used for computing the model's score. Line 15 uses regression to fit the ODE's parameters by first solving the ODE and then using a least squares regression to fit the parameters of the ODE. Lines 17 – 21 then compute scores, filter out models above a threshold, generate a new set of models, and at last return the model with the lowest total cost. Notice that line 21 can be replaced with another filter with a score threshold so that the inductive synthesis algorithm

returns a *set* of models instead of a single model; this is the approach discussed in Chapter 8.

Convergence Criteria

An interesting theoretical question is whether the finite horizon n is strictly necessary. Under what conditions is the algorithm in Fig. 9.1.2 guaranteed to converge to a fixed point?

Theorem 8 (Conditions on convergence of the linear inductive synthesis algorithm)**.** *Denote by ms_i the value of ms after the i^{th} iteration of the loop in Fig. 9.1.2 and consider arbitrarily large n. Assume all of the auxiliary mappings used in Fig. 9.1.2 terminate and are functions. Also assume that $\texttt{newModels}(ms_i) \subset ms_i$ unless ms_i is a fixed point.*

Denote by $\texttt{len}(m)$ the syntactic weight of a model[3]*. Crucially, assume $\texttt{score}(es, m) \geq \delta\,\texttt{len}(m)$ for some $\delta > 0$. Then the sequence ms_i converges to a fixed point in finite time.*

Proof. The first paragraph of assumptions directly implies that $ms_i \subseteq ms_{i+1}$ and that $|ms_i| < |ms_{i+1}|$ unless ms_i is a fixed point. Because this sequence of sets never loses elements and always grows in size monotonically until reaching a fixed point, it suffices to show that there is some finite supremal set ms_n such that $ms_k \subseteq ms_n$ for any k.

If $\texttt{len}(m) > \frac{\texttt{threshold}}{\delta}$ then $\texttt{score}(m) \geq \delta\frac{\texttt{threshold}}{\delta} = \texttt{treshold}$ and therefore m is excluded from ms by line 20. There are obviously finitely many systems of fully parametric linear ODEs whose syntactic weight is less than $\frac{\texttt{threshold}}{\delta}$. Therefore, the desired m_n is the set of all models with syntactic weight less than $\frac{\texttt{threshold}}{\delta}$. □

Although convergence is a nice property, it is both unnecessary and insufficient as a criteria for effective model identification. Therefore, we now turn our attention to the problem of empirically evaluating several different choices for the `score` and `newModels` functions.

Candidate Score Functions and Synthesis Algorithms

In this chapter we consider the following class of candidate scoring functions:

$$\texttt{score}(es, m) = k\texttt{weight}(m) + (1-k)|\{e \in es : \texttt{predict}(m, e_u, e_{in}) \neq e_{out}\}|$$

where $0 < k < 1$ is a parameter, m a system of differential equations, the `weight` function is simply the size of the AST of the right-hand sides of m except for exponents whose weights are equal to their values[4], and `predict` maps each model, input state, and time direction to the model's predicted output state.

In addition to considering several candidate cost functions we also consider several candidate synthesis algorithms.

The first and simplest synthesis algorithm is our `baseline`. This approach completely disregards scoring; the cost function is still used to select a *final* model but does not direct the iterative synthesis process. At each iteration i, `baseline` simply returns all models with syntactic weight equal to i. This continues until $i = \texttt{threshold}$ so that the user-defined threshold determines the maximum size of the model. Notice that `threshold` therefore controls both the

[3] e.g., the number of nodes in its AST

[4] i.e., the weight of $x' = x^2 + x + 1$ is 6 while the weight of $x^{100} + 1$ is 102

highest possible degree of the model and also, for models with many dependent variables, the degree to which these variables may interact.

The second synthesis algorithm we consider leverages the error in variables to determine which differential equations should be changed. For example, given an equation of the form $x' = \theta_x, o' = \theta_o$, the synthesis algorithm will only introduce new candidates for θ_x if the set of incorrect experiments es_{out} for the best available model contains a sufficient number of errors caused by the existing models' predication of the x variable's value.

9.1.3 Controller Synthesis

Model identification is inherently inductive, moving from empirical observations of the world to a mathematical model that comports with those observations. However, once a model is determined, deriving a controller that is safe with respect to the model is purely deductive. This section explains how to derive a controller from a safety specification φ and a system of ODEs `plant`.

Definition 22 (The Controller Synthesis Problem)**.** *Given:*

- *a safety specification* `safe` *stated in FOL*$_{\mathbb{R}}$*,*
- *a system of differential equations* `plant`*,*
- *a finite set of control variables* U*, and*
- *for each control variable* $u \in U$ *a set of possible values* $F_u \subseteq \mathbb{R}$*,*

the ***controller synthesis problem*** *is to derive a set of initial conditions* `init`*, a definition of the set of possible control vectors* α*, and a safety guard* ψ *satisfying the conditions following:*

- $(s,t) \in [\![\alpha]\!]$ *with* $t(u) = v$ *iff* $v \in F_u$*, and*
- $\texttt{init} \rightarrow [\{\alpha; ?\psi; \texttt{plant}\}*]\texttt{safe}$*.*

The *finite controller synthesis problem* assumes that each F_u is finite. The *time-aware controller synthesis problem* assumes an explicit timer t is reset at least every T time units, so that the final formula has the form:

$$\texttt{init} \wedge T > 0 \rightarrow [\{\alpha; ?\psi; t := 0; \{\texttt{plant}, t' = 1 \wedge t \leq T\}\}^*]\texttt{safe}.$$

Observe that the finite time-aware controller synthesis problem reduces to the problem of associating with each possible set of control inputs a test program that prevents the system from choosing those control inputs unless doing so is safe.

Let V be the set of all possible control vectors expressed as programs. For example, if $U = \{a\}$ and $F_a = \{-B, A\}$ then $V = \{a := -B, a := A\}$. Also, if $U = \{x, y\}$ with $F_x = \{1, 2\}$ and $F_y = \{3, 4\}$ then $V = \big\{\{x := 1; y := 3\}, \{x := 1; y := 4\}, \{x := 2; y := 3\}, \{x := 2; y := 4\}\big\}$. Then the controller synthesis problem is to generate, for each $v \in V$, a formula φ_v such that:

$$\vdash \texttt{init} \wedge T > 0 \rightarrow [\{\{\cup_{v \in V} ?\varphi_v; v\}; t := 0; \texttt{plant}\}^*]\texttt{safe}$$

In this chapter we are concerned with solving both the controller synthesis problem and also obtaining an actual proof that the chosen controller is safe. We first discuss the problem of synthesizing a controller and then turn our attention to automating theorem proving.

Solving the controller synthesis problem in full generality requires finding the fixed point of a quantified arithmetic predicate. Unfortunately, calculating this fixed-point is intractable for even simple examples. We therefore focus on the two important subclasses of this problem that usually arise in practice. All of the examples considered in this chapter fit into these classes.

```
inputs:
  init: The initial state.
  U: a finite set of control actions.
  plant: A system of ODEs.
  T: upper bound on the time between control steps.
  safe: S → 𝔹 is the safety constraint.
  soln: S × U × ℝ⁺ → S is the state reached when following plant for T time
     from an initial state.
output:
  a test program for each control action in U.
val fallback = fallback-in(U, plant)
if fallback is defined:
  U.map(u => {
    if u == fallback:
      return {?true;}
    else:
      return {?QE(∀0 ≤ t ≤ T∀s ≥ t safe(solve(solve(x₀, u, t), fallback, s);}
  })
else:
  U.map(u => return {?safe(solve(u, T));}
```

This algorithm first checks if there is some action – called a fallback action – that is safe for all time from some subset of state space. If such an action exists, then that action is always permitted.

This strategy assumes the existence of an action that is safe for all forward time from some subset of state space. For example, for the safety constraint $x \geq 0$ and the system of differential equations $\{x' = v, v' = a, t' = 1 \& t \leq T\}$, the action $a := -B$ for $B > 0$ is globally safe from the region $x > \frac{v^2}{-2B}$.

Generating Initial Conditions Assuming that there is such an action, it is possible to determine the existence of such an action u_{fallback} and constraint F_u by attempting to prove

$$[u_{\mathit{fallback}}; \mathit{ODEs}]\mathit{safe}$$

where *ODEs* is the *plant* without time-based domain constraints with the tactic `unfold; solve (1); partialQE`. If the result of this tactic is not *false*, then u_{fallback} is a globally safe action from the set of states described by the resulting arithmetic. Notice that the resulting arithmetic also defines the initial conditions for the model.

If there is no action that is safe for all forward time for some subset of state space, then the system simply checks that the current action is safe for the next time step. Notice that this is not sufficient for ensuring that the controller is live; in particular, it may be the case that the system is only vacuously safe because *no* safe control action is available in some reachable states. Although it is possible to characterize the maximally permissive, always-live controller in terms

of fixed points, computing these fixed points requires quantifier elimination calls that quickly become intractable.

9.1.4 Automated Proving

We would like to automatically prove formulas of the form

$$\mathit{init} \rightarrow [\{\{\cup_i ?g_i; u_i\}; \mathit{plant}\}^*]\mathit{safe},$$

where *plant* is generated by the model identification algorithm, the guards g_i are generated by the synthesis algorithm, each assignment u_i is an assignment to each variable in the control vector, and i ranges over all of the finitely many choices for the control vector.

The Axiomatic ODE Solver discussed in Chapter 4 can solve the equations *plant* so the only remaining problem is to generate a loop invariant. When a globally safe control action exists, the invariant can be immediately synthesized.

9.2 Proof-Generating Synthesis for Nonlinear Systems

Generating proofs for nonlinear systems is significantly more difficult than the linear case because synthesizing controllers and automatically proving safety properties is much less systematic. For this reason, we constrain the search process using the verification-preserving model updates framework developed in Chapter 8. By only considering model updates for which safe controllers are already known, we offload the problem of synthesizing proofs and models to the problem of identifying a robust set of model updates. Naturally, this remains a difficult model engineering problem.

The algorithm is much simpler to the linear algorithm: starting with a set of initial models, we successively apply all available model updates, use regression to fit parameters, judge the fitness of each model, exclude models below a threshold, and continue until reaching an arbitrary horizon.

9.3 Conclusion

Formal verification techniques typically focus on providing guarantees in situations where a model is designed by a human. This chapter considers the problem of obtaining formal proofs of safety even in cases where models are partially constructed using design-time data, such as high-fidelity simulations or executions of previous system runs. Providing safety guarantees for systems designed in this way is an important problem because engineers are increasingly turning toward data on previous system executions to build robust autonomy into cyber-physical systems. This chapter explains how to construct explainable and provably correct models from data available at design time. The approach in this chapter leverages the insight that hybrid program synthesis has both an inductive phase and a deductive phase. This insight allows us to construct, from data, provably correct control software.

Chapter 10

Conclusion

Autonomous cyber-physical systems that use reinforcement learning for control are amenable to formal verification. High-level models may be verified in a hybrid systems theorem prover such as KeYmaera X. Once verified, these models can be used to sandbox model-based learning algorithms. The JSC algorithm demonstrates that verification results also provide a powerful signal for directing off-model learning toward safe actions. Even when no initial model is available, formal verification provides a unique perspective for guiding the model identification and controller synthesis process.

This thesis introduced several conceptual tools for combining learning and reasoning to build robust control systems that come with explainable and verifiable safety guarantees. These frameworks – tactical hybrid systems theorem proving with proof term generation, justified speculation, and hybrid program induction guided by verification preserving model updates – provide a powerful foundation for safe learning and lay the groundwork for tackling many important remaining problems. How can we incorporate safe learning for control with safe learning for perception? How can hybrid program synthesis be combined with off-model policy learning? And perhaps most importantly, do the techniques developed in this scale to real industrial systems?

Perhaps the most important insight from Part II of this thesis is that, when it comes to safe learning, logic has much more to offer than mere sandboxing. Although sandboxing is important, it is only the beginning of logic's role in the safe learning story. JSC with quantified model monitors, μlearning, and hybrid program induction together demonstrate several ways in which programming languages and their verification logics provide a promising semantic target for explainable and verifiable machine learning. An obvious direction for future work is to continue pulling on this thread by finding additional ways in which combinations of theorem proving and machine learning can leverage each others' strengths. This vision goes far beyond controls and dynamic logics.

A significant gap in current work toward safe control is the construction of systems that can build their own justifications for self-improvement. This thesis focused only on enabling learning in a way that is justified (i.e., satisfies existing safety constraints assuming a priori modeling assumptions, as in Chapter 7), but did not tackle the much more difficult problem of building systems capable of constructing their own justifications or learning the successes and failures of their own attempts at reasoning. One very concrete avenue for future work would be to extend the

paradigm of hybrid program induction to the setting of the $\text{LP}_{\text{d}\mathcal{L}}$ logic introduced in Chapter 5, so that systems may learn how to construct new proofs instead of relying, as in Chapter 9, on pre-existing proof search techniques.

By explaining how to obtain trustworthy correctness proofs for on-model control and how to leverage these proofs during off-model control, this thesis demonstrates that autonomous cyber-physical systems that use reinforcement learning for control are amenable to formal verification. Future work in this direction should focus on scaling current techniques to industrial systems and pushing the boundaries of justifiable autonomy by allowing systems to autonomously learn new models, new control strategies, and new proof techniques for establishing the correctness of new control strategies.

Bibliography

[1] *Conference on Automated Deduction - CADE-25*, volume 9195 of *LNCS*, 2015. Springer. 10

[2] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, volume 70 of *Proceedings of Machine Learning Research*, pages 22–31. PMLR, 2017. 6.2.1, 6.2.1, 8.6

[3] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In *Verified Software: Theories, Tools and Experiments - 6th International Conference (VSTTE 2014)*. 1, 3.3

[4] Régis Alenda, Nicola Olivetti, and Gian Luca Pozzato. Nested sequent calculi for conditional logics. In Luis Fariñas del Cerro, Andreas Herzig, and Jérôme Mengin, editors, *Logics in Artificial Intelligence*, volume 7519 of *LNCS*, pages 14–27. Springer-Verlag, 2012. 5.2

[5] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*. AAAI Press, 2018. 6.2.1, 6.2.1, 6.1, 8, 8.6

[6] Eitan Altman. Constrained Markov Decision Processes, 1999. 6.2.1, 6.2.1

[7] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Grossman et al. [77], pages 209–229. 1, 2.1

[8] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the Real-Time Systems Symposium*, pages 2–11. IEEE Computer Society, 1993. 3.3

[9] Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors. *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA*, volume 1066 of *LNCS*, 1996. Springer. 10

[10] Rajeev Alur, Radu Grosu, and Bow-Yaw Wang. Automated refinement checking for asynchronous processes. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Meth-*

ods in Computer-Aided Design, Third International Conference (FMCAD 2000), volume 1954 of *LNCS*, pages 55–72. Springer, 2000. 3.3

[11] Yashwanth Annpureddy, Che Liu, Georgios E. Fainekos, and Sriram Sankaranarayanan. S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, volume 6605 of *LNCS*, pages 254–257. Springer, 2011. 3.3

[12] Sergei N. Artemov. Operational modal logic. Technical Report MSI 9529, Cornell University, 1995. 5.1, 5.2

[13] Sergei N. Artemov and Lev D. Beklemishev. Provability Logic. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd Edition*, volume 13 of *Handbook of Philosophical Logic*, pages 189–360. Springer Netherlands, 2005. 5.1, 5.3.1

[14] E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli. Effective synthesis of switching controllers for linear systems. *Proceedings of the IEEE*, 88(7):1011–1025, July 2000. 8.6

[15] J. Andrew (Drew) Bagnell, Andrew Y. Ng, and Jeff Schneider. Solving uncertain Markov decision problems. Technical Report CMU-RI-TR-01-25, Carnegie Mellon University, Pittsburgh, PA, August 2001. 6.2.1, 6.2.1, 6.2.1

[16] Bruno Barras, Lourdes Del Carmen González-Huesca, Hugo Herbelin, Yann Régis-Gianas, Enrico Tassi, Makarius Wenzel, and Burkhart Wolff. Pervasive parallelism in highly-trustable interactive theorem proving systems. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013*, volume 7961 of *LNCS*, pages 359–363. Springer, 2013. 4.6

[17] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, 2003. 8.5.4

[18] Ezio Bartocci, Jyotirmoy V. Deshmukh, Alexandre Donzé, Georgios E. Fainekos, Oded Maler, Dejan Nickovic, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 135–175. Springer, 2018. 6.3.2

[19] A. Bemporad. Hybrid Toolbox - User's Guide, 2004. 3.3

[20] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems*, number 1066 in LNCS, pages 232–243. Springer–Verlag, October 1995. 3.3

[21] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In Alur et al. [9], pages 208–219. 3.3

[22] Frdric Benhamou and Laurent Granvilliers. Continuous and interval constraints. In

Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 571 – 603. Elsevier, 2006. 3.3

[23] Luca Benvenuti, Davide Bresolin, Pieter Collins, Alberto Ferrari, Luca Geretti, and Tiziano Villa. Ariadne: Dominance checking of nonlinear hybrid automata using reachability analysis. In Alain Finkel, Jérôme Leroux, and Igor Potapov, editors, *Reachability Problems - 6th International Workshop (RP 2012)*, volume 7550 of *LNCS*, pages 79–91. Springer, 2012. 3.3

[24] P. Blanchard, R.L. Devaney, and G.R. Hall. *Differential Equations*. Brooks/ Cole, Cengage Learning, 2011. 4.5.3

[25] Brandon Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völp, and André Platzer. Formally verified differential dynamic logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*, pages 208–221. ACM, 2017. 3.1, 3.3, 4.2, 4.6, 5.1, 5.2

[26] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. VeriPhy: Verified controller executables from verified cyber-physical system models. In Dan Grossman, editor, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, pages 617–630. ACM, 2018. 3.4, 5, 5.1, 8.5, 9

[27] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015. 4.6

[28] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym. *CoRR*, abs/1606.01540, 2016. 7.5

[29] Christopher Brooks. Ptolemy II: An open-source platform for experimenting with actor-oriented design. Berkeley EECS Annual Research Symposium (BEARS), February 2016. 3.3

[30] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 150–153. Springer, 2010. 3.3

[31] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Taylor model flowpipe construction for non-linear hybrid systems. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS 2012)*, pages 183–192. IEEE Computer Society, 2012. 3.3

[32] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference (CAV 2013)*, volume 8044 of *LNCS*, pages 258–263. Springer, 2013. 3.3

[33] Xin Chen, Sriram Sankaranarayanan, and Erika Ábrahám. Flow* 1.2: More effective to play with hybrid systems. In Goran Frehse and Matthias Althoff, editors, *1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems (ARCH 2014 and ARCH 2015)*, volume 34 of *EPiC Series in Computing*, pages 152–159.

EasyChair, 2015. 3.3

[34] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. 4.6

[35] A. Church. *Introduction to Mathematical Logic*, volume 13 of *Annals of Mathematics Studies*. Princeton University Press, 1996. 2.4

[36] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. 1, 3

[37] George E. Collins and H. Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *J. Symb. Comput.*, 12(3):299–328, 1991. 4.1, 5.3.1

[38] Robert L. Constable, Stuart F. Allen, Mark Bromley, and et. al. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986. 4.6

[39] Stefano P. Coraluppi and Steven I. Marcus. Risk-sensitive and minimax control of discrete-time, finite-state Markov decision processes. *Automatica*, 35(2):301–309, 1999. 6.2.1, 6.2.1

[40] Stefano P. Coraluppi and Steven I. Marcus. Mixed risk-neutral/minimax control of discrete-time, finite-state Markov decision processes. *IEEE Transactions on Automatic Control*, 45(3):528–532, 2000. 6.2.1, 6.2.1

[41] James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *J. Symb. Comput.*, 5(1/2):29–35, 1988. 4.4

[42] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. UPPAAL SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, Aug 2015. 3.3

[43] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool KRONOS. In Alur et al. [9], pages 208–219. 3.3, 3.3

[44] N. G. de Bruijn. *AUTOMATH, a Language for Mathematics*, pages 159–200. Springer, Berlin, Heidelberg, 1983. 5.1

[45] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. 3.2

[46] Leonardo Mendonça de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *CoRR*, abs/1505.04324, 2015. 3

[47] Leonardo Mendonça de Moura et. al. The Lean theorem prover (system description). In *CADE-25* DBL [1], pages 378–388. 1, 3, 4.6, 3

[48] Manfred Deistler. System identification and time series analysis: Past, present, and future. In Bozenna Pasik-Duncan, editor, *Stochastic Theory and Control*, pages 97–109. Springer, 2002. 6.3.1

[49] Kurt Driessens and Saso Dzeroski. Integrating guidance into relational reinforcement learning. *Machine Learning*, 57(3):271–304, 2004. 6.2.2

[50] Parasara Sridhar Duggirala, Matthew Potok, Sayan Mitra, and Mahesh Viswanathan.

C2E2: a tool for verifying annotated hybrid systems. In Antoine Girard and Sriram Sankaranarayanan, editors, *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control (HSCC 2015)*, pages 307–308. ACM, 2015. 3.3

[51] Krishnamurthy Dvijotham, Sven Gowal, Robert Stanforth, Relja Arandjelovic, Brendan O'Donoghue, Jonathan Uesato, and Pushmeet Kohli. Training verified learners with learned verifiers. *CoRR*, abs/1805.10265, 2018. 6.2.3

[52] Johan Eker, Jorn W Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003. 3.3

[53] Matthew England and James H. Davenport. The complexity of cylindrical algebraic decomposition with respect to polynomial degree. In Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing - 18th International Workshop (CASC 2016)*, volume 9890 of *LNCS*, pages 172–192. Springer, 2016. 5.3.1

[54] Melvin Fitting. The logic of proofs, semantically. *Annals of Pure and Applied Logic*, 132 (1):1 – 25, 2005. 5.1

[55] International Organization for Standardization (ISO). ISO 26262 road vehicles functional safety. 2011. 2

[56] Simon Foster and Jim Woodcock. Towards verification of cyber-physical systems with UTP and Isabelle/HOL. In Thomas Gibson-Robinson, Philippa J. Hopcroft, and Ranko Lazic, editors, *Concurrency, Security, and Puzzles - Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday*, volume 10160 of *LNCS*, pages 39–64. Springer, 2017. 3.3

[57] Martin Fränzle and Christian Herde. Hysat: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2007. 3.3

[58] Goran Frehse. PHAVer: algorithmic verification of hybrid systems past HyTech. *STTT*, 10(3):263–279, 2008. 3.3

[59] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *23rd CAV, 2011*, volume 6806 of *LNCS*, pages 379–395. Springer, 2011. 3.3, 4.1

[60] David Fridovich-Keil, Sylvia L. Herbert, Jaime F. Fisac, Sampada Deglurkar, and Claire J. Tomlin. Planning, fast and slow: A framework for adaptive real-time safe trajectory planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 387–394, 2018. 8.6

[61] Nathan Fulton and André Platzer. A logic of proofs for differential dynamic logic: toward independently checkable proof certificates for dynamic logics. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, 2016*, pages 110–121, 2016. 3.1, 1

[62] Nathan Fulton and André Platzer. Safe reinforcement learning via formal methods: To-

ward safe control through proof and learning. In Sheila McIlraith and Kilian Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, pages 6485–6492. AAAI Press, 2018. 6.1, 1, 7.7, 8, 8.1, 8.5.1, 8.6

[63] Nathan Fulton and André Platzer. Safe AI for CPS (invited paper). In *IEEE International Test Conference (ITC)*, 2018. 1

[64] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In Amy P. Felty and Aart Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015. 1, 2.4, 3.1, 1, 4.1, 4.1, 5.3.3, 6.2.1, 9

[65] Nathan Fulton, Stefan Mitsch, Brandon Bohrer, and André Platzer. Bellerophon: Tactical theorem proving for hybrid systems. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference (ITP 2017)*, volume 10499 of *LNCS*, pages 207–224. Springer, 2017. 3.2, 3.3, 1

[66] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. Delta-decidability over the reals. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 305–314. IEEE Computer Society, 2012. 3.3

[67] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *LNCS*, pages 208–214. Springer, 2013. 3.3

[68] Sicun Gao, Soonho Kong, Wei Chen, and Edmund M. Clarke. Delta-complete analysis for bounded reachability of hybrid systems. *CoRR*, abs/1404.7171, 2014. 3.3

[69] Javier García and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16:1437–1480, 2015. 6.1, 6.2, 6.2.1

[70] A. Garg, K. Tai, and B. N. Panda. System identification: Survey on modeling methods and models. In Subhransu Sekhar Dash, K. Vijayakumar, Bijaya Ketan Panigrahi, and Swagatam Das, editors, *Artificial Intelligence and Evolutionary Computations in Engineering Systems*, pages 607–615. Springer, 2017. 6.3.1

[71] Chris Gaskett. Reinforcement learning under circumstances beyond its control. In *Proceedings of the International Conference on Computational Intelligence for Modelling Control and Automation*, 2003. 6.2.1

[72] Peter Geibel. Reinforcement learning for MDPs with constraints. In *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, pages 646–653, 2006. 6.2.1, 6.2.1

[73] Shromona Ghosh, Felix Berkenkamp, Gireeja Ranade, Shaz Qadeer, and Ashish Kapoor. Verifying Controllers Against Adversarial Examples with Bayesian Optimization. *CoRR*, abs/1802.08678, 2018. 6.3.2, 8

[74] Antoine Girard. Reachability of uncertain linear systems using zonotopes. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control, 8th Inter-*

national Workshop, HSCC 2005, Zurich, Switzerland, March 9-11, 2005, Proceedings, volume 3414 of *LNCS*, pages 291–305. Springer, 2005. 3.3

[75] Andrea Giusti, Martijn J. A. Zeestraten, Esra Icer, Aaron Pereira, Darwin G. Caldwell, Sylvain Calinon, and Matthias Althoff. Flexible automation driven by demonstration: Leveraging strategies that simplify robotics. *IEEE Robot. Automat. Mag.*, 25(2):18–27, 2018. 6.3.2

[76] Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000. 4.3.1

[77] Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors. *Hybrid Systems*, volume 736 of *LNCS*, 1993. Springer. 1, 10

[78] Colas Le Guernic and Antoine Girard. Reachability analysis of hybrid systems using support functions. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference (CAV 2009)*, volume 5643 of *LNCS*, pages 540–554. Springer, 2009. 3.3

[79] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *J. ACM*, 40(1):143–184, January 1993. 5.2

[80] John Harrison. HOL light: A tutorial introduction. In *Formal Methods in Computer-Aided Design, First International Conference (FMCAD 1996)*, pages 265–269, 1996. 3

[81] John Harrison. A HOL Theory of Euclidean Space. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *LNCS*, pages 114–129. Springer, 2005. 4.6

[82] John Harrison, Konrad Slind, and Rob Arthan. HOL. In Freek Wiedijk, editor, *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *LNCS*, pages 11–19. Springer, 2006. 3

[83] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Logically-correct reinforcement learning. *CoRR*, abs/1801.08099, 2018. 6.2.1, 6.2.1, 6.1, 8, 8.6

[84] Matthias Heger. Consideration of risk in reinformance learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning, Proceedings of the Eleventh International Conference, Rutgers University, New Brunswick, NJ, USA, July 10-13, 1994*, pages 105–111. Morgan Kaufmann, 1994. 6.2.1, 6.2.1

[85] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society, 1996. 3.3

[86] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *STTT*, 1(1-2):110–122, 1997. 3.3

[87] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94 – 124, 1998. 6.2.1

[88] Sylvia L. Herbert, Mo Chen, SooJean Han, Somil Bansal, Jaime F. Fisac, and Claire J. Tomlin. FaSTrack: A modular framework for fast and guaranteed safe motion planning.

In *IEEE Annual Conference on Decision and Control (CDC)*. 8.6

[89] Christian Herde, Andreas Eggers, Martin Fränzle, and Tino Teige. Analysis of hybrid systems using HySAT. In *The Third International Conference on Systems (ICONS 2008)*, pages 196–201. IEEE Computer Society, 2008. 3.3

[90] A. Heyting. *Mathematische Grundlagenforschung: Intuitionismus, Beweistheorie*. Number v. 3, no. 4 in Ergebnisse der mathematik und ihrer grenzgebiete 3. bd.,4. Julius Springer, 1934. 5.2

[91] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL - A modular logical environment. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *LNCS*, pages 287–303. Springer, 2003. 4.6

[92] D. Hilbert and W. Ackermann. *Grundzüge der theoretischen Logik*. Grundlehren der mathematischen Wissenschaften. Springer, 1938. 2.4

[93] D. Hilbert, W. Ackermann, R.E. Luce, and L.M. Hammond. *Principles of Mathematical Logic*. AMS Chelsea Publishing Series. American Mathematical Society, 1999. 2.4

[94] Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Fourth Conference on Interactive Theorem Proving (ITP 2013)*, volume 7998 of *LNCS*, pages 279–294. Springer, 2013. 4.6

[95] Jannik Hüls, Stefan Schupp, Anne Remke, and Erika Abraham. Analyzing hybrid petri nets with multiple stochastic firings using HyPro. In *Proc. of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'17)*, 2017. 3.3

[96] Fabian Immler and Johannes Hölzl. Numerical analysis of ordinary differential equations in Isabelle/HOL. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving - Third International Conference (ITP 2012)*, volume 7406 of *LNCS*, pages 377–392. Springer, 2012. 3.3

[97] Fabian Immler and Johannes Hölzl. Ordinary differential equations. *Archive of Formal Proofs*, 2012, 2012. 3.3

[98] Fabian Immler and Christoph Traut. The flow of ODEs. In Jasmin Christian Blanchette and Stephan Merz, editors, *ITP 2016*, pages 184–199, 2016. 4.6

[99] Wolfram Research, Inc. Mathematica, Version 11.3. Champaign, IL, 2018. 3.1, 3.2

[100] Nils Jansen, Bettina Könighofer, Sebastian Junges, and Roderick Bloem. Shielded decision-making in MDPs. *CoRR*, abs/1807.06096, 2018. 8.6

[101] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Ryan Gardner, Aurora Schmidt, Erik Zawadzki, and André Platzer. A formally verified hybrid system for the next-generation airborne collision avoidance system. In Christel Baier and Cesare Tinelli, editors, *TACAS*, volume 9035 of *LNCS*, pages 21–36. Springer, 2015. 1

[102] Taylor T. Johnson, Stanley Bak, Marco Caccamo, and Lui Sha. Real-time reachability

for verified simplex design. *ACM Transactions on Embedded Computing Systems*, 15(2): 26:1–26:27, 2016. 6.3.2

[103] Aleksandar Lj. Juloski, W. P. M. H. Heemels, Giancarlo Ferrari-Trecate, René Vidal, Simone Paoletti, and J. H. G. Niessen. Comparison of four procedures for the identification of hybrid systems. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*, pages 354–369. Springer, 2005. 6.3.1

[104] Yoshinobu Kadota, Masami Kurano, and Masami Yasuda. Discounted Markov decision processes with utility constraints. *Computers and Mathematics with Applications*, 51(2): 279 – 284, 2006. 6.2.1

[105] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification - 29th International Conference (CAV 2017)*, pages 97–117, 2017. 6.2.3, 6.1

[106] Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming*, pages 50–73. Springer, 2010. 8.6

[107] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009 (SOSP 2009)*, pages 207–220. ACM, 2009. 5

[108] Wolf Kohn, Anil Nerode, and Jeffrey B. Remmel. Hybrid systems as finsler manifolds: Finite state control as approximation to connections. In Panos J. Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry, editors, *Proceedings of the Third International Workshop on Hybrid Systems*, volume 999 of *LNCS*, pages 294–321. Springer, 1994. 6.3.2

[109] A. Kolmogoroff. Zur deutung der intuitionistischen logik. *Mathematische Zeitschrift*, 35 (1):58–65, Dec 1932. 5.2

[110] Soonho Kong, Sicun Gao, Wei Chen, and Edmund M. Clarke. dReach: δ-reachability analysis for hybrid systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015*, volume 9035 of *LNCS*, pages 200–205. Springer, 2015. 3.3

[111] Robbert Krebbers and Bas Spitters. Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science*, 9(1), 2011. 4.6

[112] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. Formal C semantics: Compcert and the C standard. In Gerwin Klein and Ruben Gamboa, editors, *5th International Conference on Interactive Theorem Proving (ITP 2014)*, volume 8558 of *LNCS*, pages 543–548. Springer, 2014. 5

[113] Peeyush Kumar, Wolf Kohn, and Zelda B. Zabinsky. Near optimal hamiltonian-control and learning via chattering. *CoRR*, abs/1703.06485, 2017. 6.3.2

[114] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a

verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. 5

[115] Alexander B. Kurzhanski and Pravin Varaiya. On ellipsoidal techniques for reachability analysis. part I: external approximations. *Optimization Methods and Software*, 17(2):177–206, 2002. 3.3

[116] Alexander B. Kurzhanski and Pravin Varaiya. On ellipsoidal techniques for reachability analysis. part II: internal approximations box-valued constraints. *Optimization Methods and Software*, 17(2):207–237, 2002. 3.3

[117] Taisa Kushner, David Bortz, David M. Maahs, and Sriram Sankaranarayanan. A data-driven approach to artificial pancreas verification and synthesis. In Chris Gill, Bruno Sinopoli, Xue Liu, and Paulo Tabuada, editors, *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS 2018)*, pages 242–252. IEEE / ACM, 2018. 6.3.2

[118] Yuri A. Kuznetsov. *Elements of Applied Bifurcation Theory (2Nd Ed.)*. Springer-Verlag, 1998. 4.5.3

[119] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL: Status and developments. In Orna Grumberg, editor, *Computer Aided Verification CAV 1997)*, number 1254 in LNCS, pages 456–459. Springer–Verlag, June 1997. 3.3

[120] Xuan-Bach D. Le, Quang Loc Le, David Lo, and Claire Le Goues. Enhancing automated program repair with deductive verification. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME 2016)*, pages 428–432. IEEE Computer Society, 2016. 6.3.2

[121] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012. 6.3.2, 8.6

[122] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a Mechanized Metatheory of Standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, pages 173–184. ACM, 2007. 5.2

[123] K. Rustan M. Leino and Valentin Wüstholz. The dafny integrated development environment. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment (F-IDE 2014)*, volume 149 of *EPTCS*, pages 3–15, 2014. 1

[124] Shiau Hong Lim, Huan Xu, and Shie Mannor. Reinforcement learning in robust Markov decision processes. In *Neural Information Processing Systems*, pages 701–709, 2013. 6.2.1, 6.2.1

[125] Ernest Lindelöf. Sur lapplication de la méthode des approximations successives aux équations différentielles ordinaires du premier ordre. *Comptes rendus hebdomadaires des séances de lAcadémie des sciences*, 116(3):454–457, 1894. 2.3

[126] Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. A calculus for hybrid CSP. In Kazunori Ueda, editor, *Programming Languages*

and Systems - 8th Asian Symposium (APLAS 2010), volume 6461 of *LNCS*, pages 1–15. Springer, 2010. 3.3

[127] Sarah M. Loos and André Platzer. Differential refinement logic. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2016*, pages 505–514. ACM, 2016. 6.3.2

[128] Sarah M. Loos, André Platzer, and Ligia Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pages 42–56, 2011. 1, 7.5

[129] Assia Mahboubi. Programming and certifying the CAD algorithm inside the Coq system. In *Mathematics, Algorithms, Proofs, volume 05021 of Dagstuhl Seminar Proceedings, Schloss Dagstuhl*, 2005. 5.3.1

[130] Ibtissem Ben Makhlouf, Norman Hansen, and Stefan Kowalewski. HyReach: A reachability tool for linear hybrid systems based on support functions. In Goran Frehse and Matthias Althoff, editors, *3rd International Workshop on Applied Verification for Continuous and Hybrid Systems (ARCH 2016)*, volume 43 of *EPiC Series in Computing*, pages 68–79. EasyChair, 2016. 3.3

[131] Per Martin-Löf. An Intuitionistic Theory of Types. 1998. 5.1

[132] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0. 1, 3, 4.1, 4.2, 4.6, 5.2, 5.2

[133] Stefan Mitsch and André Platzer. ModelPlex: Verified runtime validation of verified cyber-physical system models. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *RV*, volume 8734 of *LNCS*, pages 199–214. Springer, 2014. 6.2.1, 8.5, 9

[134] Stefan Mitsch and André Platzer. The KeYmaera X proof IDE - concepts on usability in hybrid systems theorem proving. In Catherine Dubois, Paolo Masci, and Dominique Méry, editors, *Proceedings of the Third Workshop on Formal Integrated Development Environment, F-IDE@FM 2016, Limassol, Cyprus, November 8, 2016.*, volume 240 of *EPTCS*, pages 67–81, 2016. 3.3, 4.2, 4.6

[135] Stefan Mitsch and André Platzer. ModelPlex: Verified runtime validation of verified cyber-physical system models. *Form. Methods Syst. Des.*, 49(1):33–74, 2016. Special issue of selected papers from RV'14. 4.5.2, 4.5.3, 7.1

[136] Allen Newell and Herbert A. Simon. The logic theory machine-a complex information processing system. *IRE Transactions on Information Theory*, 2(3):61–79, 1956. 5.1

[137] Markus Nick, Sören Schneickert, Jürgen Grotepaß, and Ingo Heine. CheckMATE. *KI*, 21 (4):34–37, 2007. 3.3

[138] Arnab Nilim and Laurent El Ghaoui. Robust control of Markov decision processes with uncertain transition matrices. *Operations Research*, 53(5):780–798, September-October 2005. 6.2.1, 6.2.1

[139] Arnab Nilim and Laurent El Ghaoui. Robustness in Markov decision problems with uncertain transition matrices. In *Neural Information Processing Systems*, pages 839–846.

MIT Press, 2003. 6.2.1

[140] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 3, 5.2

[141] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. 1, 4.1

[142] Illah Reza Nourbakhsh. The coming robot dystopia: All to inhuman. *Foreign Affairs*, 94 (4), July/August 2015. 2

[143] Martin Odersky. Essentials of Scala. In Bernard Carré and Olivier Zendra, editors, *Langages et Modèles à Objets (LMO 2009)*, volume L-3 of *RNTI*, page 2. Cépaduès-Éditions, 2009. 4.3.1

[144] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Towards practical verification of machine learning: The case of computer vision systems. *CoRR*, abs/1712.01785, 2017. 6.2.3

[145] Frank Pfenning and Carsten Schürmann. System description: Twelf a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE 2016)*, volume 1632 of *LNCS*, pages 202–206. Springer, 1999. 5.2

[146] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *International Joint Conference on Automated Reasoning (IJCAR 2010)*, pages 15–21, July 2010. 5.2

[147] André Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2): 143–189, 2008. 2.1, 2.2, 4.2, 5.3.3

[148] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010. 2.1, 2.2, 4.1

[149] André Platzer. Logics of dynamical systems. In *Proceedings of the 27th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2012)*, pages 13–24. IEEE, 2012. 2.1, 2.2

[150] André Platzer. The complete proof theory of hybrid systems. In *LICS*, pages 541–550. IEEE, 2012. 7.3

[151] André Platzer. A uniform substitution calculus for differential dynamic logic. In *CADE* DBL [1]. 2.4, 2.4, 5.3, 5.3.2, 8, 5.3.2, 9, 5.4, 5.4, 5.4, 5.4, 5.5

[152] André Platzer. A complete uniform substitution calculus for differential dynamic logic. *Jounral of Automated Reasoning*, 59(2):219–266, 2017. 2.2, 2.3, 2.3, 2.4, 2.1, 2.4, 4.1, 4.1, 4.2, 4.2, 4.4, 4.4, 4.5.2, 4.5.4

[153] André Platzer and Edmund M. Clarke. Computing differential invariants of hybrid systems as fixedpoints. *Formal Methods in System Design*, 35(1):98–120, 2009. 4.5.4, 4.5.5

[154] André Platzer and Jan-David Quesel. KeYmaera: A hybrid theorem prover for hybrid systems (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR 2008*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008. 1,

3.3, 4.1, 4.2, 4.5.4, 5.1

[155] André Platzer and Yong Kiam Tan. Differential equation axiomatization: The impressive power of differential ghosts. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2018)*, pages 819–828, New York, 2018. ACM. 3.4

[156] André Platzer, Jan-David Quesel, and Philipp Rümmer. Real world verification. In Renate A. Schmidt, editor, *CADE'22*, volume 5663 of *LNCS*, pages 485–501. Springer, 2009. 3.1, 5.3.1

[157] H. Poincaré. Sur l'quilibre d'une masse fluide anime d'un mouvement de rotation. *Acta Math.*, 7:259–380, 1885. 4.5.3

[158] Jan-David Quesel, Stefan Mitsch, Sarah M. Loos, Nikos Arechiga, and André Platzer. How to model and prove hybrid systems with KeYmaera: a tutorial on safety. *STTT*, 18 (1):67–91, 2016. 5.2

[159] Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *Formal Methods - 21st International Symposium (FM 2016)*, volume 9995 of *LNCS*, pages 593–611, 2016. 6.3.2, 8.6

[160] RTCA. DO-178C software considerations in airborne systems and equipment certification. 2012. 2

[161] Stuart J. Russell, Daniel Dewey, and Max Tegmark. Research priorities for robust and beneficial artificial intelligence. *CoRR*, abs/1602.03506, 2016. 6.1

[162] Sadra Sadraddini and Calin Belta. Formal guarantees in data-driven model identification and control synthesis. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (HSCC 2018)*, pages 147–156, 2018. 8.6

[163] John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1889–1897, 2015. 6.2.1, 8.6

[164] Stefan Schupp, Erika Abraham, Ibtissem Ben Makhlouf, and Stefan Kowalewski. HyPro: A C++ library for state set representations for hybrid systems reachability analysis. In *Proceedings of the 9th NASA Formal Methods Symposium (NFM 2017)*, volume 10227 of *LNCS*, pages 288–294. Springer International Publishing, 2017. 3.3

[165] Dana S. Scott. A type-theoretical alternative to iswim, cuch, OWHY. *Theoretical Computer Science*, 121(1&2):411–440, 1993. 4.2, 4.3.1

[166] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference (TPHOLS 2000)*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008. 3

[167] Alexey Solovyev and Thomas C. Hales. Formal Verification of Nonlinear Inequalities with Taylor Interval Approximations. In Guillaume Brat, Neha Rungta, and Arnaud Venet,

editors, *NASA Formal Methods*, volume 7871 of *LNCS*, pages 383–397. Springer, 2013. 4.6

[168] A. S. Troelstra. *History of Constructivism in the 20th Century Vol. Ml-91-05*. University of Amsterdam, 1991. 2

[169] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, volume 80 of *JMLR Workshop and Conference Proceedings*, pages 5052–5061, 2018. 6.3.2

[170] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. *CoRR*, abs/1804.10829, 2018. 6.2.3, 6.1

[171] Eric W. Weisstein. Quantifier elimination. MathWorld–A Wolfram Web Resource. URL `http://mathworld.wolfram.com/QuantifierElimination.html`. 3.1

[172] Bruno Woltzenlogel Paleo. Contextual natural deduction. In Sergei Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science*, volume 7734 of *LNCS*, pages 372–386. Springer, 2013. 5.2