Thesis Proposal
Practical End-to-End Verification of Cyber-Physical Systems

Brandon Bohrer

September 12, 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
André Platzer, Chair
Stefan Mitsch
Frank Pfenning
Tobias Nipkow, TU Munich
Fifth Member

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

Keywords: hybrid systems, theorem proving, end-to-end verification, hybrid games, hybrid logic

Abstract

Cyber-physical systems (CPSs) combining discrete control and continuous physical dynamics are pervasive in modern society: examples include driver assistance in cars, industrial robotics, airborne collision avoidance systems, and the electrical grid. Many of these systems are safety-critical because they operate in close proximity to humans. Formal safety verification of these systems is important because it is a key tool for attaining the strongest possible safety guarantees.

Hybrid systems models, in particular, are a successful formalism for CPS. Hybrid systems theorem-proving in differential dynamic logic (dL) and its generalization differential *game* logic (dGL) is notable for strong logical foundations and successful application in case studies using the theorem provers KeYmaera and KeYmaera X. However, safety verification of models does not imply safety of *implementations*, which might not be faithful to the model. Further still, a formal proof means nothing if the proof-checking software is *unsound*.

This thesis addresses implementation and soundness gaps by using constructive logic and programming languages as the foundation of an end-to-end verification toolchain. That is: *Constructive Differential Game Logic (CdGL) enables practical, end-to-end verification of cyber-physical systems.* CdGL enables synthesis of implementations with bulletproof theoretical foundations. Logic is the keystone of the end-to-end connection from high-level proofs and foundations to implementations. Our pursuit of practical proving includes innovations in the proof scripting language itself.

CdGL proofs, in contrast to dGL, are suitable for synthesizing *controllers* which determine safe actions for a CPS and *monitors* which check the compliance of the external environment with model assumptions. The synthesized code is automatically proven correct down to machine-code level. The foundations are also strengthened by our formalization of dL's soundness in Isabelle/HOL, allowing proofs in the fragment dL to be exported and rechecked. We evaluate the toolchain on a 2D robot which follows arcs. The model and implementation cross-validate each other: monitors catch incorrect code and assumptions, while testing with monitors enabled allows us to assess the realism of the model.

Chapter 1

Introduction

Cyber-physical systems (CPSs) combining discrete control and continuous physical dynamics are pervasive in modern society: examples include driver assistance in cars, industrial robotics, airborne collision avoidance systems, the electrical grid, and medical devices. Many of these systems are safety-critical or even life-critical because they operate in close proximity to humans and in some cases perform life-sustaining functions. Formal safety verification of these systems is a key tool for attaining the strongest possible guarantees that they meet their safety objectives. Therefore, as the importance of CPS in society grows, so does the importance of their formal verification. *Hybrid systems* models combining discrete transitions with continuous differential equations (ODEs), in particular, have succeeded in providing a common formalism for the discrete and continuous aspects of a CPS. Of the available approaches to verification, hybrid systems theorem-proving in differential dynamic logic (dL) (Platzer, 2018a, 2008, 2017a, 2011) is notable for its strong logical foundations and successful application in a number of case studies (Jeannin et al., 2015; Loos et al., 2011; Mitsch et al., 2013; Platzer & Quesel, 2008b) using the theorem provers KeYmaera (Platzer & Quesel, 2008a) and KeYmaera X (Fulton et al., 2015). Other notable approaches include model-checking of automata-theoretic (Henzinger, 1996) models, control synthesis, and runtime monitoring.

While approaches differ, the fundamental motivation for CPS is always the same: to promote the safety and correctness of real systems. The end-to-end philosophy recognizes that hybrid systems are only an abstraction of reality, so achieving safety in practice requires bridging the abstraction gaps between models and implementations. As discussed in Section 1.1.2, this is the common feature between our work and other end-to-end approaches. However, our work is set apart by formal proof artifacts at every step, verification to machine-level, and a productive and expressive source language for modeling and proof. It is because of this combination of features that we call our work end-to-end.

In this thesis, we extend the dL tradition to an end-to-end verification technique. We use VeriPhy as a name both for the resulting software toolchain and the general approach. The VeriPhy approach says that to achieve end-to-end guarantees, every stage of verification should be founded in logics and programming languages with well-defined formal semantics. By verifying systems *constructively*, we enable synthesis of correct implementations from proofs in the general case. Because the source, implementation, and proof languages all have formal semantics, we can formally show that implementations are refinements of source models and thus are safe.

The Cast. End-to-end verification is hard in large part because it must balance competing theoretical and practical needs. Throughout this thesis, we argue that VeriPhy meets these needs better than existing approaches by providing a dialogue between three characters: the Logician, the Logic-User, and the Engineer. These three metaphorical characters are avatars for different perspectives on CPS, namely formal logic, interactive theorem proving, and CPS implementation. While contrived by the author, the characters are a stylized way to emphasize and resolve differences of perspective between theorists and practitioners. We now introduce each character.

The Logician follows in the formalist tradition of David Hilbert: to know something is to have a proof of it. Today's Logician goes further than Hilbert and says to have a proof is to have a machine-checkable derivation in a sound, formal proof calculus, and to have a sound proof calculus is to have a formal semantics against which the rules have a formal proof of soundness. The Logician holds mathematics to the highest standard possible, and they do so with good reason. CPS are often life-critical, and if all the Logician's paranoia can prevent defects in CPS, it has been worthwhile. Not only have flaws been found in designs of unverified CPS, but bugs in informal proofs are commonplace. Even bugs in theorem provers, while less common, have been uncovered through careful verification (Bohrer et al., 2017). Considering the existence of such bugs, the Logician is justified in their paranoia.

The Engineer, however, might see these efforts as misguided. The Engineer is the one tasked with designing, building, and delivering a production CPS under time and budget constraints. The Engineer will gladly use formal modeling and verification, but only if it can show concrete safety benefits on realistic systems in a short timeframe. Techniques that appeal to the Logician might appall the Engineer because they are time-consuming or only guarantee safety of an ideal model. The Engineer would rather fix one bug in the implementation than ten bugs in an ideal model, since fixing a bug in the model is not guaranteed to improve the quality of shipped code.

The Logic-User is oft-forgotten, and sits between the Logician and Engineer on the spectrum from theory to practice. The Logic-User (called the Proof Engineer by some authors), is the person tasked with employing verification tools *at scale*. Unlike the Logician, the Logic-User does not obsess with the soundness of proof rules, because they trust that the Logician has implemented the verification tool correctly. The Logic-User believes in the value of a verified model, but sympathizes with the Engineer's plea that only a nuanced model could hope to capture the difficulties faced in practice. Verification takes time, but the Logic-User expects their time to be spent wisely: time should not be wasted on verification tasks that could easily be automated away, and no unnecessary barriers to learning the tool should be erected. As authors have noted (Ringer et al., 2019), the productivity of the Logic-User takes on growing significance today with the growth of large-scale verification efforts.

As shown in Table 1.1, no prior approach satisfies all characters. General-purpose interactive theorem provers (GPITP) such as the HOL family and Coq satisfy the Logician's desire for solid logical foundations because their foundations have been extensively formalized (Barras, 2010; Kumar et al., 2016). However, the Logic-User will be dissatisfied with the laborious proofs required when specialized support for CPS is absent, and the Engineer will be dissatisfied that practical artifacts like controllers and runtime monitors must still be implemented manually. Synthesis tools based on hybrid automata can produce useful control code for the Engineer, but the Logician may be disappointed with the paper-only foundations and the Logic-User frustrated by the conceptual gap between automata and programs with which they are familiar. Of the

available choices, the Logic-User prefers domain-specific logics like dL: a program-like syntax helps avoid modeling mistakes, while proof is not as laborious as with GPITP. This comes at cost to the Engineer because state-of-the-art synthesis tools for dL (Mitsch & Platzer, 2016) only synthesize monitors, and sometimes fail to synthesize monitors for complicated models in practice. Prior to this thesis, the Logician also paid the price that dL's foundations were developed only informally on paper.

Nor is the Logician truly satisfied with any existing approach: the Logician may have trouble convincing themself that they modeled the system correctly, as the models of CPS in GPITP are ad-hoc by comparison to automata and dL's hybrid programs. Merely generating code from the model does not resolve the issue of validating the model, either. Moreover, while GPITP's support code extraction, the extracted code in practice is not provably correct. Despite recent work on proof-producing code extraction for Coq's Gallina (Mullen et al., 2018; Anand et al., 2017; Ioannidis et al., 2019) and for Isabelle/HOL (Hupel & Nipkow, 2018), these works are not sufficient in practice because they support limited feature sets.

Approach	Logician	Engineer	Logic-User
GPITP	formal	manual effort	labor-intensive
Automata	paper	monitors, controls	error-prone
dL before	paper	some monitors	less error-prone, less labor-intensive
dL after	formal	some monitors	less error-prone, less labor-intensive
CdGL	formal	monitors, controls	least error-prone, less labor-intensive

Table 1.1: Comparison of Verification Approaches

The goal of this thesis is to satisfy the needs of all the Logician, Engineer, and Logic-User at once, a goal we call "Practical, End-to-End Verification of CPS". The terms "practical" and "end-to-end" are notoriously vague; this thesis defines end-to-end verification as resolving the competing needs of the Logician and Engineer. Specifically, we define verification as end-to-end if it connects the formal end to the implementation end. We ensure the implementation of the CPS obeys a precise formal safety guarantee in terms of the system model, which itself rests on a bulletproof mathematical foundation. We say that verification is practical if it also meets the needs of the Logic-User, by simplifying models, minimizing the risk of serious modeling errors, and simplifying proofs. As with all verification, ours leaves some assumptions, for example on sensing and actuation. However, we deem it less important to eliminate every possible assumption, and more important that our approach can scale to minimize these assumptions in future work as discussed in Section 4.1.

Fig. 1.1 lays out the relationships between the chapters of the thesis. The already-completed works in this thesis connect implementation artifacts to foundations by way of dL: the monitor synthesis tool VeriPhy (Section 4.1) connects an implementation (for example, of a robot: Section 4.2) to a formal model in dL. Chapter 3 then satisfies the Logician by putting dL (as implemented in KeYmaera X) on firm footing in Isabelle/HOL (Section 3.1). Section 3.2 generalizes the foundations of dL to survive future development. Chapter 5 develops a constructive game logic for discrete games. The proposed work extends our logical foundation by developing CdGL (Constructive dGL) and Kaisar, a principled language for CdGL proofs. We show that

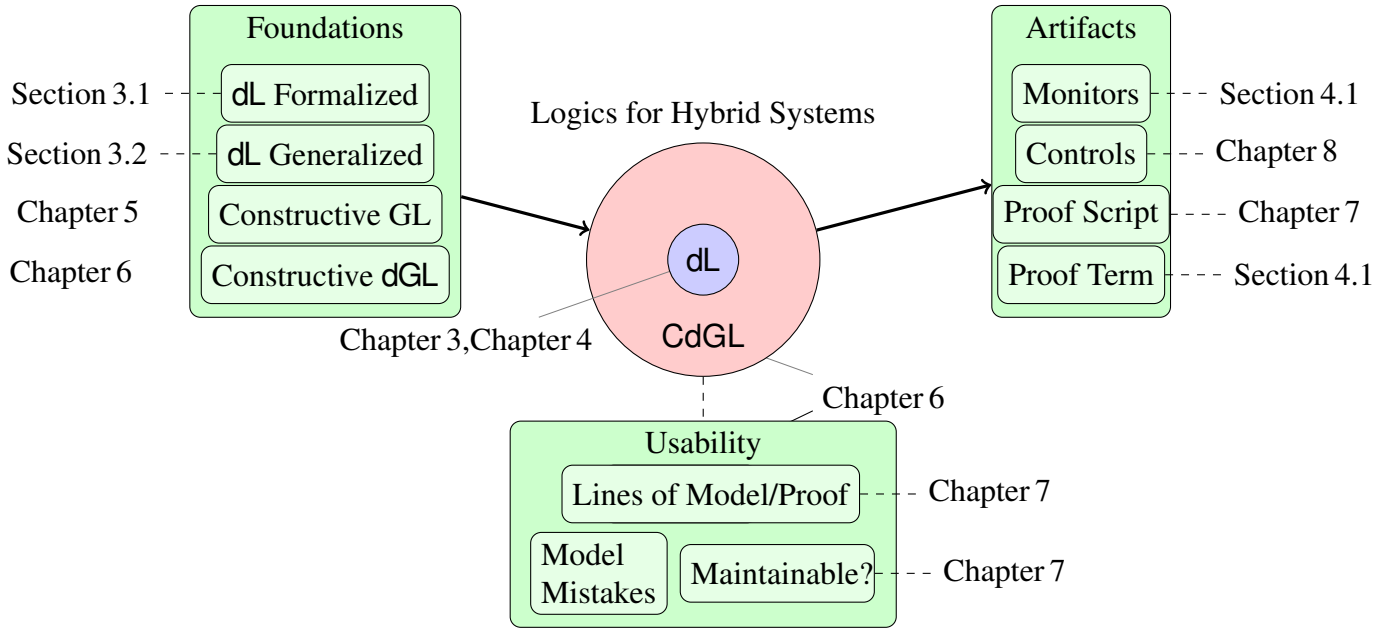


Figure 1.1: Goals of the thesis

this foundation promotes simpler models (Chapter 6) and supports synthesis of both controllers and monitors in the general case. These additions meet the practical needs of the Engineer and Logic-User while providing a higher-level input language for the VeriPhy pipeline and generating an additional output: verified controllers.

At last, we arrive at a concise thesis statement for the thesis as a whole:

> Constructive Differential Game Logic (CdGL) enables practical, end-to-end verification of cyber-physical systems.

In Section 1.1, we will compare the VeriPhy approach in detail with other works that strive for an end-to-end philosophy. Our take on end-to-end verification is that implementation-level guarantees must come by formally connecting implementation to models, else the Logician will not accept the system as verified. For practicality, we transport guarantees to implementation level automatically, with machine-code artifacts that the Engineer can use. In Section 4.2, our 2D ground robotics case study serves to show our approach scales to non-trivial scenarios. The Logic-User must be able to perform their proofs in a high-level, productive language. Viewed through this lens, Table 1.1 shows that the VeriPhy approach, with its strong foundation in logic, marries practical end-to-end verification foundation with rigorous foundations on a level far beyond the state-of-the-art.

1.1 Related Work

Here, we discuss works that are broadly related to the thesis as a whole. Works which are relevant primarily to one chapter are discussed in that chapter.

1.1.1 Verification of Theorem Provers

Because theorem-provers are used to ascertain the correctness of critical systems, their own correctness is essential. This thesis includes a formalization of dL in Isabelle/HOL (Section 3.1) which has been used to generate a verified proof-term checker for dL (Section 4.1). We compare our contributions to other works on theorem prover verification. In general, the differences among formalizations mirror the differences among the underlying logics, so the unique features of our work include real analysis proofs, ordinary differential equations (ODEs), and an explicit treatment of substitution.

Barras and Werner (Barras & Werner, 1997) verified a typechecker for a fragment of Coq in Coq. Harrison (Harrison, 2006b) verified (1) a weaker version of HOL Light's kernel in HOL Light and (2) HOL Light in a stronger variant of HOL Light. Myreen et al. have extended this work, verifying HOL Light in HOL4 (Myreen et al., 2013; Kumar et al., 2016) and using their verified compiler CakeML (Kumar et al., 2014) to ensure these guarantees apply at the machine-code level. Myreen and Davis proved the soundness of the ACL2-like theorem prover Mitawa in HOL4 (Myreen & Davis, 2014). Anand, Bickford, and Rahli (Anand & Rahli, 2014; Rahli & Bickford, 2016) proved the relative consistency of Nuprl's type theory (Constable et al., 1986; Allen et al., 2006) in Coq with the goal of generating a verified prover core. Twelf (Pfenning & Schürmann, 1999) was used to formalize the LF (Harper et al., 1993) logical framework on which it is based (Martens & Crary, 2012).

Like the above works, the goal of Section 3.1 is to verify the correctness of a theorem prover (KeYmaera X). This goal is one part of the thesis' broader goal of bulletproof foundations: formal proofs are only as trustworthy as their foundations. Since proof in dL is a lynchpin of the VeriPhy approach, it was crucial to show the soundness of dL's foundations. Section 3.1 exposed unique technical challenges because the foundations of dL differ greatly from those of the other provers. Soundness for the differential equations of dL involved significant proofs in real analysis. The KeYmaera X core is based on a uniform substitution calculus (Platzer, 2017a), so the formalization also includes significant proofs about substitution.

Our verified checker supports a significant fragment of the KeYmaera X core, including enough to recheck proofs of safety for monitor-based synthesized controllers ($\approx$100K steps). While the Isabelle/HOL formalization is based directly on the calculus supported in KeYmaera X, another contribution of the thesis (Section 3.2) is to develop a foundation for dL which is more amenable to the practical extensions used in KeYmaera X in practice. Together, these two pieces provide a multifaceted argument how we can trust hybrid systems proofs as done in KeYmaera X. Recently, an independent formalization of classical dGL has been made (Platzer, 2019), which emphasizes uniform substitution and simplicity of the formalization over suitability for extraction of a verified checker.

1.1.2 Verification of CPS

Hybrid systems verification is a broad and actively studied field. The field can be subdivided by the guarantees provided, artifacts generated, and methods employed. Methods can be broadly divided into model-checking (by reachability analysis) and theorem proving. Model-checkers typically support a higher degree of automation, while theorem-provers are typically more expressive and often have smaller trusted cores. Verification of safety, liveness, and other properties of models is often treated in isolation from synthesis, but our end-to-end approach emphasizes the use of proof to drive synthesis. Synthesis tools also vary in how, and how thoroughly, they justify the correctness of the synthesized artifacts. When the artifact is expressed in a high-level language, correctness depends on the correct compilation of that language. Additionally, justifying the correctness of the artifact entails justifying the model used as a source, which poses an additional challenge for approaches using coarse, discrete models. We now discuss the above classes of works in greater detail.

Formalisms for Hybrid Systems

In the verification of hybrid systems, a preliminary question is which formalism should be used to represent a system. While the expressive power of different formalisms is largely equivalent, differing representations can be a practical barrier to implementing verification technology. Throughout this thesis we use *hybrid programs* (Platzer, 2018a), which represent hybrid systems as programs in a programming language featuring differential equation evolution as a statement. A deterministic counterpart to hybrid programs has been used to investigate hybrid systems information flow (Prabhakar & Köpf, 2013). WHILE$^{\mathtt{dt}}$ (Suenaga & Hasuo, 2011) also represents hybrid systems as programs, but uses infinitesimals from non-standard-analysis to express differential equations as loops. Automata-theoretic representations (Henzinger, 1996) are most widely-used for model-checking. Process-calculi such as Hybrid CSP (Zhou et al., 1995; Liu et al., 2010), HyPA (Cuijpers & Reniers, 2005), and Hybrid χ (Schiffelers et al., 2003) have been used for model-checking and theorem-proving. IPL (Ruchkin et al., 2018) was introduced to promote interoperation between heterogeneous modeling languages such as these, especially those which target different levels of abstraction.

Reachability Analysis

Model-checking for hybrid systems by reachability analysis is thoroughly studied. Well-known model-checkers include SpaceEx (Frehse et al., 2011) and Flow* (X. Chen et al., 2013), as well as C2E2 (Duggirala et al., 2015). Compared to theorem-provers, their trusted bases can be large ($\approx$100,000 SLOC for SpaceEx). No model-checker is as expressive as the theorem-provers: often, model-checkers are restricted to compact state spaces and bounded-time safety, and only a few support non-linear differential equations.

The primary use of reachability analysis today is offline verification of safety properties of hybrid systems. Reachability has been applied to models from many application domains, including ground robotics (X. Chen et al., 2015), which is an application area of focus for this thesis. Reachability tools excel at such case studies because in return for their large trusted bases

and limited expressivity, they gain efficient automation. However, other uses have received some attention as well:

- Runtime reachability analysis has been performed in real time on cars (Althoff & Dolan, 2014), which serves as the basis of a fail-safe control method (Pereira & Althoff, 2015). Our work shares the basic fail-safe approach, but our work has formal proof artifacts, enforces model compliance, and does not require trusting a reachability analysis algorithm. Thus, prior reachability-based runtime verification approaches do not amount to end-to-end verification in our sense of the term.
- While safety analysis is the most widely-implemented, some solvers (Cimatti et al., 2015) implement liveness analysis (Cimatti et al., 2014) for hybrid systems as well.

In response to the use of reachability analysis algorithm in trusted contexts, (Immler, 2015) verified a set-based reachability analysis for ODEs in Isabelle, using the Isabelle differential equations library (Immler & Traut, 2016). While it is an important first step, it supports only numerical solutions of ODEs, not hybrid systems.

Theorem-provers

Interactive theorem-proving is one of the main approaches to hybrid systems verification, and is characterized by high expressivity, which can come at the cost of increased verification effort. Domain-specific logics such as dGL and Hybrid Hoare Logic are implemented in special-purpose theorem provers such as KeYmaera X (Fulton et al., 2015) and Hybrid Hoare Prover (Liu et al., 2010). Special-purpose provers enable a tailored experience for the Logic-User, but cannot verify systems outside their chosen logic. GPITP's such as Coq and Isabelle have also been used to verify hybrid systems (Rouhling, 2018), related models of CPS's (Rizaldi et al., 2018, 2017; Chan et al., 2016), and related topics such as geometry (Affeldt & Cohen, 2017) and dynamical systems (Cohen & Rouhling, 2017). GPITP's provide access to large libraries of general-purpose mathematics, but do not provide a tailored experience.

Differential Dynamic Logics This thesis is part of a long line of work using differential dynamic logic (dL) (Platzer, 2008, 2012c, 2017a) and logics descended from it. Compared to reachability analysis approaches, deductive verification in dL is symbolic and expressive: all polynomial differential equations are supported, and discrete tests and assignments are also polynomial. Guarantees are typically unbounded-time, and the plant (i.e., physics) is modeled faithfully as a continuous differential equation, not a discrete difference equation. The tradeoff is that verification is undecidable, and serious case studies usually require significant manual interaction from the user. The KeYmaera X (Fulton et al., 2015) theorem prover implements dGL and provides significant automation to minimize the number of interactions required. dL and KeYmaera X have been applied in a number of case studies (Platzer, 2016).

This thesis includes a case study which we use to assess VeriPhy's ability to verify realistic systems. Our case study generalizes dL proofs of 1D straight-line motion (with direct velocity control) (Bohrer et al., 2018), of 2D obstacle avoidance, and 1D liveness (Mitsch et al., 2013, 2017). A paper proof of liveness using dL rules assumed perfect sensing and constant speed (Martin et al., 2017). Their controllers, like ours, are closely related to the classic DYNAMIC-

WINDOW (Fox et al., 1997) control algorithm. In contrast to the prior dL effort, we offer stronger results, including 2D liveness, waypoint-following, and end-to-end correctness.

Synthesis for Planning and Control

Low-level controllers are an important part of any CPS, so their correctness is an important part of CPS correctness. Our approach to end-to-end verification (Section 4.1) uses correct-by-construction synthesis of monitors to remove low-level controllers from the trusted base for safety. However, correctness of controllers is still important to ensure *liveness*, and our case study (Section 4.2) assumes high-level plans as an input, while correctness of high-level plans is an interesting problem in its own right (Rizaldi et al., 2018). In proposed work (Chapter 8), we will complement our monitor synthesis with correct-by-construction low-level controller synthesis. Related works have explored planning and control synthesis at the high level, with correctness assumptions on the kinematic models:

- The tools LTLMoP (Finucane et al., 2010) and TuLiP (Filippidis et al., 2016) can synthesize robot controls that satisfy a high-level temporal logic specification. Their strength is that they provide easy-to-learn user interfaces for modeling and verification. Their weaknesses are that their models discretize space and time, and there is not a chain of formal artifacts for each step of synthesis. Thus, even though they share our goal of end-to-end verification, their work does not satisfy our conception of end-to-end.
- High-level plans have been synthesized (Bhatia et al., 2010; Fainekos et al., 2009) which satisfy specifications over a hybrid-dynamical system. Bisimulation arguments have been used (Alur et al., 2000) to show that the implementation refined the specification. However, these works are not end-to-end because the hybrid models are entirely trusted and low-level control is not considered.
- Controllers have been synthesized: *i)* from temporal logic specifications for linear systems (Kloetzer & Belta, 2008), *ii)* for adaptive cruise control (Nilsson et al., 2016), tested in simulation and on hardware, and *iii)* from safety proofs (Taly & Tiwari, 2010) for switched systems using templates.

However, the above techniques are limited to simplistic ODEs and do not provide an end-to-end chain of formal evidence.

Online Verification

The approaches above are *offline*: i.e., they are used before-the-fact to ascertain the correctness of a model of a CPS. In contrast, *online verification* (or runtime verification) uses checks at runtime to enforce correctness. Online verification excels at treating systems that are not fully understood before runtime (such as sensors and actuators) and systems which we simply prefer not to model (such as complex untrusted control systems). However, because behaviors are checked only at runtime, any approach which uses runtime verification *alone* cannot ensure correctness of the untrusted control, nor the ability of the monitored system to achieve its mission objectives in practice (liveness). While some prior works place runtime verification on a rigorous formal foundation, no work until this thesis has connected the foundations end-to-end.

- The basis of online verification is the SIMPLEX (Seto et al., 1998) method, which uses a trusted monitor to decide between an untrusted controller and trusted fallback. SIMPLEX is most helpful when untrusted controllers are complex, but monitors and fallback controllers are simple. The key insight of SIMPLEX is that the controller need not be trusted because safety depends only on the control decision which it outputs, and the output is only executed if the monitor approves it, else the trusted fallback controller is executed.
- ModelPlex is a tool for synthesizing dL monitor formulas (for both the controller and plant) from proven-safe dL models in KeYmaera X. However, it simply synthesizes a dL formula describing the monitored conditions, not an executable program. Thus ModelPlex is not end-to-end when taken on its own. Additionally, until extensions concurrent with this thesis (Bohrer et al., 2018), ModelPlex could not exploit proof insights during synthesis. Thus, prior releases of ModelPlex struggled to produce plant monitors which ever passed when executed in practice, or any meaningful plant monitor for most differential equations whose solutions are non-polynomial. One motivation for the proposed works is elaborate the recent advance of exploiting proof information in synthesis.
- A contribution of this thesis is the VeriPhy (Bohrer et al., 2018) toolchain for dL. VeriPhy extends SIMPLEX by ensuring the monitor is correct-by-construction (via ModelPlex), formally proving the safety of the resulting system, and automatically maintaining safety down to machine code implementation using CakeML (Kumar et al., 2014).
- High-Assurance SPIRAL (Franchetti et al., 2017) is a pragmatic compilation toolchain for ModelPlex-synthesized monitors for dL, but struggles to provide formal end-to-end guarantees because it relies on difficult-to-verify compiler optimizations. It has not been validated with an end-to-end case study on free-range 2D driving.
- Runtime reachability analysis has been used for car control (Althoff & Dolan, 2014), but relies on correctness of the model and of the analysis implementation. Since reachability tools rely on large, complicated codebases and models are challenging to get right, these assumptions present a correctness gap which stands in the way of an end-to-end argument.
- MOP (F. Chen & Rosu, 2007) is a framework which makes monitoring a key part of the software development cycle. Its implementation in ROS (Huang et al., 2014) has been used for industrial robotics case studies at Runtime Verification, Inc.

In conclusion, while runtime verification is a key part of our end-to-end approach, prior works have not provided our end-to-end proof artifacts. Even at this time, it is an active research challenge to effectively exploit proof information in the synthesis process to provide monitors that are safe, but still satisfiable in practice.

End-to-End Approaches

End-to-end verification refers to any verification approach which attempts to verify CPS as they exist in reality while maintaining a rigorous foundation. Notably, none of these approaches show correctness unconditionally: all have a trusted base and make simplifying assumptions about reality.

- High-Assurance SPIRAL (Franchetti et al., 2017) generates executable x64 machine code

based on dL model, and has been applied to geo-fencing and dynamic-window control (Low & Franchetti, 2017). However, because SPIRAL itself emphasizes esoteric compiler optimizations, High-Assurance SPIRAL must always play catchup with SPIRAL, and will not support every optimization in practice. It also stops before machine code (Zaliva & Franchetti, 2018). Moreover, VeriPhy adopts a deeper understanding of hybrid systems: several stages of VeriPhy exploit language-level semantic proofs that link the initial hybrid system model to a low-level executable semantics. High-Assurance SPIRAL, in contrast, discards the semantic understanding of hybrid system, which then precludes the clear connection between the system model and compiled code as provided by VeriPhy.

- ROSCoq (Anand & Knepper, 2015) is a framework based on the Logic of Events for reasoning about distributed CPS in Coq with constructive reals and generating verified controllers. Their use of constructive reals is prescient, and inspires the design choices in Chapter 8. Their model does provide ODEs for the physics of the robot, but because they provide limited support for reasoning about derivatives, extensive manual proofs are required by the user. They use Coq's default, unverified code extraction for synthesis of controllers (but not monitors). It is not clear whether their formalization is in the fragment of Coq supported by verified code extraction efforts.
- VeriDrone (Ricketts et al., 2015) is a framework for verifying hybrid systems in Coq that relies on a discrete-time temporal logic called RTLA, inspired by TLA (Lamport, 1992). They prove an analog of DI, but not the other ODE axioms, thus other ODE proofs, as well as arithmetic proofs, are likely laborious. They do not automatically verify monitors, nor is their machine code verified. Their model uses discrete time, which is a significant gap between model and reality, complicating any end-to-end effort. They use Coq's default, unverified code extraction for synthesis. It is not clear whether their formalization is in the fragment of Coq supported by verified code extraction efforts.

In short, while all of the above are motivated by an end-to-end philosophy, none have succeeded in providing end-to-end results in our sense.

Chapter 2

Background: Differential Game Logic dGL

Differential dynamic logic (dL) (Platzer, 2018a, 2008, 2017a, 2012b) is a dynamic logic for the formal verification of hybrid systems, which combine discrete transitions with differential equations (ODE's) to model CPS's. *Differential game logic* (dGL) (Platzer, 2015, 2017b, 2018b) extends the programs α of dL with a turn-taking operator α^d to enable representing and verifying hybrid *games* which extend hybrid systems with *adversarial* dynamics.

While some thesis chapters (Chapter 3, Section 4.1) employ dL, we present full dGL since i) Chapter 5 (completed) relies on games, ii) the proposed works (Chapter 6,Chapter 8) will require an understanding of full dGL, and iii) an introductory understanding of dGL will also provide an introductory understanding of dL. While the syntax of dGL and dL are nearly identical, their semantic differences are deep: dL is a regular modal logic whose denotational semantics can be and are given in the forward-chaining, relational Kripke style, while dGL is a subregular logic with a backward-chaining winning-region semantics.

2.1 Syntax

We introduce the syntax and informal meaning of dGL here, then introduce the winning-region semantics formally in Section 2.2. The syntax of dGL consists of three syntactic classes: terms, formulas, and hybrid games.

Definition 1 (Terms of dGL)**.** Terms θ, η of dGL are defined recursively according to the following grammar:

$$\theta, \eta ::= q \mid x \mid \theta + \eta \mid \theta \cdot \eta \mid (\theta)'$$

Where $q \in \mathbb{Q}$ is a rational constant, $x \in \mathcal{V}$ is a program variable and $\mathcal{V}$ is the (at most countable) set of all base variable names. For every base variable $x \in \mathcal{V}$, there is a differential variable $x' \in \mathcal{V}'$ standing for the differential of x. Terms $\theta + \eta$ and $\theta \cdot \eta$ are the sum and product of θ and η, and $(\theta)'$ is the *differential* of term θ. It is worth noting that the dGL term language is restrictive, and for good reason: all terms are polynomials, so all terms are defined in every state and even C^∞-smooth. Because dGL terms are well-behaved, the theory of dGL is simplified as a result. In Section 3.2, we show how to remove these simplifying assumptions in the context of dL, enabling many more term constructs that are essential in practical proving, the most common constructs including quotients, roots, trigonometric functions, and conditionals.

Definition 2 (Formulas of dGL)**.** Formulas are defined by the following grammar:

$$\phi, \psi ::= \theta \sim \eta \mid \phi \wedge \psi \mid \phi \vee \psi \mid \neg\phi \mid \phi \to \psi \mid \phi \leftrightarrow \psi \mid \forall x\, \phi \mid \exists x\, \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi$$

where $\sim$ stands for any comparison operator $\sim \quad \in \{\leq, <, =, \neq, >, \geq\}$. We write conjunctions $\phi \wedge \psi$, negations $\phi \vee \psi$, disjunctions $\phi \vee \psi$, implications $\phi \to \psi$, biimplications $\phi \leftrightarrow \psi$, and quantifiers over real numbers $\exists x\, \phi$ and $\forall x\, \phi$. The diamond modality $\langle\alpha\rangle\phi$ says that the player who is actively making decisions (typically named Angel) has a strategy for the game α which ensures postcondition ϕ. The box modality $[\alpha]\phi$ says that the player who is not currently making decisions (typically named Demon) has a strategy for the game α which ensures postcondition ϕ. This generalizes the meaning of the same modalities in dL, where $[\alpha]\phi$ says postcondition ϕ holds for *all* runs of α, while $\langle\alpha\rangle\phi$ says ϕ holds as a postcondition in *some* run. Because dGL is a classical logic, it is worth noting that the formula syntax is not minimal: many constructs such as implication, equivalence, disjunction, universal quantifiers, and the box modality are all definable classically. This will *not* be the case in the proposed logic CdGL (Chapter 5) since some of the definitions are valid only classically, not constructively.

Definition 3 (Hybrid games)**.** Games are defined by the following grammar:

$$\alpha, \beta ::= x' = \theta \,\&\, Q \mid x := \theta \mid \alpha := * \mid ?\phi \mid \alpha \cup \beta \mid \alpha;\ \beta \mid \alpha^* \mid \alpha^d$$

where $x := \theta$ stores the current value of term θ in program variable x and test $?\phi$ makes Angel lose if formula ϕ does not hold in the current state. In nondeterministic assignment $x := *$, Angel chooses a value $r \in \mathbb{R}$ to assign to x. Program $x' = \theta \,\&\, Q$ evolves x continuously according to the differential equation (ODE) $x' = \theta$ for a duration $d \geq 0$ of Angel's choosing, but Angel must ensure that Q is true throughout the evolution of $x' = \theta$, else they lose. In game $\alpha \cup \beta$, Angel chooses whether to play game α or game β. In game $\alpha;\ \beta$ the players must first play α and then play β; this case is largely responsible for the need to employ backward-chaining semantics in dGL, as it is most natural to first ask from what region X is there a winning strategy for β, then ask from what region does Angel have a strategy to reach X when playing α. In the iterated game α^*, Angel decides at the end of each loop iteration whether to continue playing α another time. All plays must be finite (Angel must stop eventually), but Angel need not decide a-priori *when* to stop, let alone announce that decision to Demon beforehand. If Angel cannot reach the goal region without repeating α infinitely, then Angel loses. To play the dual game α^d, Angel and Demon switch roles then play game α in their new roles. That is, the player previously known as Demon makes the decisions in α, at least until another dual operator is encountered. We parenthesize hybrid games $\{\alpha\}$ for clarity and disambiguation as needed. We will sometimes speak of *demonic* loops, choices, etc., which are derived from the angelic forms as follows:

$$\alpha^\times \equiv \{\{\alpha^d\}^*\}^d \qquad ?\phi^d \equiv \{?\phi\}^d$$
$$\alpha \cap \beta \equiv \left\{\alpha^d \cup \beta^d\right\}^d \qquad x' = \theta \,\&\, Q^d \equiv \{x' = \theta \,\&\, Q\}^d$$
$$x := \theta^d \equiv x := \theta \qquad \{\alpha;\ \beta\}^d \equiv \alpha^d;\ \beta^d$$

In a Demonic loop, Demon controls *only* the duration of the loop, then returns control to Angel during the body. Likewise, Demon controls only which branch is taken in a demonic choice. In

Demonic tests and differential equations, Demon is responsible for passing the test or satisfying the domain constraint. In the latter, Demon also chooses the duration. Deterministic assignment and sequential competition are self-dual, so their Demonic forms are identical to the Angelic.

We now give an example hybrid game and example safety and liveness properties. The game in Example 1 is a hybrid game generalization of a standard hybrid systems model. The game α_s is a model of adversarial driving where the opponent Demon controls the duration ($^\times$). We will write α_l to denote the counterpart where Angel controls the duration.

Example 1 (Acceleration-Controlled 1D Driving).

$$\alpha_s \equiv \tag{2.1}$$

$$v := 0;\ x := 0; (\mathsf{obs} := *; T := *; A := *; B := *; ?\mathsf{obs} > 0 \land T > 0 \land A > 0 \land B > 0)^d; \tag{2.2}$$

$$\big\{a := *;\ ? - B \leq a \land a \leq A; \tag{2.3}$$

$$t := 0;\ \{x' = v, v' = a, t' = 1 \,\&\, t \leq T \land v \geq 0\}^d\big\}^\times \tag{2.4}$$

The first line (2.2) is an initialization phase: we (Angel) call the current position $x = 0$ and are initially stopped ($v = 0$). The opponent then picks how far away our obstacle (obs) is and how long Angel will have to wait between control decisions (T). They also pick Angel's maximum braking rate B and maximum acceleration rate A. One could argue on principle that neither player really picks the timestep T, in practice this is determined by the speed of the robot's processors, sensors, and actuators, as well as the speed of its control program. However, what's important is that robot's controller (the Angel player) is *not* the one who picks, so a conservative model should assume the worst, which is that their opponent (the environment) gets to pick. The same goes for the rates A and B. It would also be eminently unreasonable if the obstacle already collided with the robot at the start of the game, and would be nonsensical if the control cycle lasted zero time or less, or if the robot could not accelerate or brake, thus the environment is responsible for picking positive $\mathsf{obs}, \mathsf{T}, \mathsf{A}$, and B, else they lose by default and Angel wins immediately. The remainder of the game is a *demonic loop* $\cdot^\times$ where the Demon player controls the duration. Angel's turn consists of making a control decision by setting the acceleration a. Angel has near complete freedom to set the acceleration: they may set any value ($a := *$) so long as it is within the physical limits of the car ($? - B \leq a \land a \leq A$). It is Angel's responsibility to do so, and they lose the game if the acceleration is out of bounds. Next, the car moves according to ideal Newtonian physics: the velocity v continuously changes according to acceleration a while position changes continuously according to the velocity. Demon (the environment) controls the duration of the ODE, but is constrained to the timestep $t \leq T$: i.e., Angel's control action must be safe even if Demon chooses not to use the full time budget, but need not be safe past time T. In that case, Demon is responsible for breaking the rules, so Angel wins. The equation $t' = 1$ simply says that t represents the time elapsed in the current ODE run. We also write α_l for α_s with an Angelic loop.

We give examples of safety and liveness theorems:

Example 2 (1D Driving Game Safety and Liveness).

$$\textit{safe} \equiv \langle \alpha_s \rangle \mathsf{x} \leq \mathsf{obs}$$

$$\textit{live} \equiv \langle \alpha_l \rangle \left(\mathsf{x} \geq \mathsf{obs} \lor t \leq \frac{T}{2} \right)$$

The safety theorem *safe* says that Angel (the robot) has a strategy to ensure $\mathsf{x} \leq \mathsf{obs}$ (i.e. a strategy not to hit the obstacle) regardless how long Demon runs the loop and the ODE. The liveness theorem *live* says that Angel (the robot) has a strategy to reach the obstacle, *so long as* Demon always runs the ODE long enough ($t \geq \frac{T}{2}$). The requirement on running the ODE long enough is to rule out *Zeno* behaviors by Demon: without this restriction, Demon can run the ODE for shorter and shorter durations, with infinitely many iterations in finite time, which is not physically realizable but would falsify the liveness theorem.

It is worth noting that with the exception of (Quesel & Platzer, 2012), prior case studies (Jeannin et al., 2015; Loos et al., 2011; Mitsch et al., 2017; Platzer & Quesel, 2008b) used vanilla dL for hybrid *systems* in KeYmaera X (Section 4.1,Section 4.2). In Chapter 6 of this thesis, we will advocate for making (constructive) games proofs the standard. We foreshadow the motivations of Chapter 6 here.

Consider a hybrid *system* version of Example 1:

Example 3 (Acceleration-Controlled 1D Driving System).

$$\text{Pre} \equiv v = 0 \land x = 0 \land \mathsf{obs} > 0 \land T > 0 \land A > 0 \land B > 0 \tag{2.5}$$

$$\alpha_{\text{Sys}} \equiv \tag{2.6}$$

$$\Big\{\Big\{\ a := *;\ ?-B \leq a \land a \leq A;\ ?\frac{(v+aT)^2}{2\mathsf{B}} \leq \mathsf{obs} - \left(\mathsf{x} + \mathsf{vT} + \mathsf{a}\frac{\mathsf{T}^2}{2}\right) \tag{2.7}$$

$$\cup\ ?v = 0;\ a := 0 \tag{2.8}$$

$$\cup\ ?v \geq 0;\ a := -B\Big\}; \tag{2.9}$$

$$\{x' = v, v' = a, t' = 1 \,\&\, t \leq T \land v \geq 0\}\Big\}^* \tag{2.10}$$

Note α_{Sys} of Example 3 is much more complex and explicit than Example 1. This is because the *strategy* by which acceleration is chosen must be explicitly given in the model as three cases: (2.7) allows any physically-possible choice that is safe for time T, (2.8) lets a robot stay stopped, and (2.9) allows braking.

Why are Example 1 and game models in general simpler? A typical game model has shape $(\mathsf{ctrl}^d;\ \mathsf{plant})^*$, where the turn-taking (or dual) operator α^d has the effect of switching players. Most often, we wish to show that there *exists* a control choice where *all* behaviors of the plant (physical environment) are safe and live, and so on for all iterations of the system. Such iterated modality alternations are precisely the one pattern expressible in dGL but not dL (Platzer, 2015). In vanilla dL, safety is usually expressed in the form $\phi \to [(\mathsf{ctrl};\ \mathsf{plant})^*]\psi$ and the only forms of liveness which can be expressed are those with finitely-many player alternations, e.g., $\phi \to \langle(\mathsf{ctrl};\ \mathsf{plant})^*\rangle\psi$ or $\phi \to [(\mathsf{ctrl};\ \mathsf{plant})^*]\langle(\mathsf{ctrl};\ \mathsf{plant})^*\rangle\psi$. dGL models are simpler because in typical dL usage, *all* cases of a controller must be safe, whereas a safe choice need only *exist* in a game. As a result, operational information (what control decision is made in which circumstances) is present in dL models, whereas it need only appear in *proofs* in dGL.

As discussed further in Chapter 6, relegating control choices to the proof has advantages for synthesis in CdGL as well as simplicity advantages. That being said, the maturity of KeYmaera X and the many available case studies show the utility of the thesis chapters which target vanilla dL as well. The connections between vanilla dL and CdGL will be further investigated in Chapter 6.

2.2 Semantics

We give the denotational semantics for classical hybrid game logic dGL, which is divided into a semantics for terms, formulas, and games. Throughout the semantics, states are written $\omega, \nu, \mu : \mathcal{S}$ where the set of states $\mathcal{S}$ is in bijection to $\mathbb{R}^n$ for $n = |\mathcal{V} \cup \mathcal{V}'|$ where $\mathcal{V}$ is the set of base variables and $\mathcal{V}'$ the set of differential variables. Regions are sets of states and are ranged over by X, Y, Z.

Definition 4 (Classical term semantics)**.** The semantics of a term $[\![\theta]\!]\omega : \mathbb{R}$ is the value of term θ in state ω, and is defined inductively on θ:

$$[\![q]\!]\omega = q \qquad [\![\theta+\eta]\!]\omega = [\![\theta]\!]\omega + [\![\eta]\!]\omega \qquad [\![(\theta)']\!]\omega = \textstyle\sum_{x\in\mathcal{V}} \frac{\partial [\![\theta]\!]\omega}{\partial x} \cdot \omega(x')$$
$$[\![x]\!]\omega = \omega(x) \qquad [\![\theta\cdot\eta]\!]\omega = [\![\theta]\!]\omega \cdot [\![\eta]\!]\omega$$

That is, literals denote themselves, variables x take their meaning from the state ω, sums and products denote the sum and product of the denotation of their operands respectively, and the differential $(\theta)'$ denotes the sum of every partial derivative of the denotation of θ, each scaled by the value of the corresponding differential variable x'. Because every term mentions at most a finite number of variables and the partial derivative with respect to an unmentioned variable is uniformly zero, this sum always has finite support. An advantage of this semantics is that it is defined in every state while agreeing with the intuitive meaning of "time derivative of $[\![\theta]\!]\omega$" whenever a differential appears in the postcondition of an ODE (Platzer, 2017a).

Definition 5 (Classical formula semantics)**.** The semantics of a formula ϕ is given by the relation $[\![\phi]\!]\omega$, which is defined inductively on ϕ

$$\begin{aligned}
[\![\theta \sim \eta]\!] &= \{\omega \mid [\![\theta]\!]\omega \sim [\![\eta]\!]\omega\} \quad \text{for } \sim \ \in \{\leq, <, =, \neq, >, \geq\} \\
[\![\neg\phi]\!] &= [\![\phi]\!]^{\complement} \\
[\![\phi \wedge \psi]\!] &= [\![\phi]\!] \cap [\![\psi]\!] \\
[\![\phi \to \psi]\!] &= \{\omega \mid \omega \in [\![\phi]\!] \text{ implies } \omega \in [\![\psi]\!]\} \\
[\![\phi \leftrightarrow \psi]\!] &= \{\omega \mid \omega \in [\![\phi]\!] \text{ iff } \omega \in [\![\psi]\!]\} \\
[\![\exists x\, \phi]\!] &= \{\omega \mid \omega^r_x \in [\![\phi]\!]\} \quad \text{for some } x \in \mathbb{R} \\
[\![\forall x\, \phi]\!] &= \{\omega \mid \omega^r_x \in [\![\phi]\!]\} \quad \text{for all } x \in \mathbb{R} \\
[\![\langle\alpha\rangle\phi]\!] &= \varsigma_\alpha([\![\phi]\!]) \\
[\![[\alpha]\phi]\!] &= \delta_\alpha([\![\phi]\!])
\end{aligned}$$

where $\varsigma_\alpha(X)$ and $\delta_\alpha(X)$ are the regions from which Angel (or Demon, respectively) has a strategy to reach a region X.

Definition 6 (Classical game semantics)**.** For any game α and goal region $X \subseteq \mathcal{S}$, we inductively define the *winning region* $\varsigma_\alpha(X)$ for Angel, i.e., the region from which they have a strategy to

enter region X after playing game α:

$$\begin{aligned}
\varsigma_{x:=\theta}(X) &= \{\omega \in \mathcal{S} \mid \omega_x^{[\![\theta]\!]\omega} \in X\} \\
\varsigma_{x:=*}(X) &= \{\omega \in \mathcal{S} \mid \omega_r^{[\![\theta]\!]\omega} \in X \text{ for some } r \in \mathbb{R}\} \\
\varsigma_{x'=\theta\,\&\,Q}(X) &= \{\varphi(0) \in \mathcal{S} \mid \varphi(r) \in X \text{ for some } r \in \mathbb{R}_{\geq 0} \text{ and (differentiable) } \varphi : [0,r] \to \mathcal{S} \\
&\qquad \text{such that } \varphi(\zeta) \in [\![Q]\!] \text{ and } \frac{\partial \varphi(t)(x)}{\partial t}(\zeta) = [\![\theta]\!]\varphi(\zeta) \text{ for all } 0 \leq \zeta \leq r\} \\
\varsigma_{?\phi}(X) &= [\![\phi]\!] \cap X \\
\varsigma_{\alpha\cup\beta}(X) &= \varsigma_\alpha(X) \cup \varsigma_\beta(X) \\
\varsigma_{\alpha;\beta}(X) &= \varsigma_\alpha(\varsigma_\beta(X)) \\
\varsigma_{\alpha^*}(X) &= \bigcap\{Z \subseteq \mathcal{S} \mid X \cup \varsigma_\alpha(Z) \subseteq Z\} \\
\varsigma_{\alpha^d}(X) &= (\varsigma_\alpha(X^{\complement}))^{\complement}
\end{aligned}$$

Likewise, we define the winning regions $\delta_\alpha(X)$ for *Demon*, which are dual to those for Angel:

$$\begin{aligned}
\delta_{x:=\theta}(X) &= \{\omega \in \mathcal{S} \mid \omega_x^{[\![\theta]\!]\omega} \in X\} \\
\delta_{x:=*}(X) &= \{\omega \in \mathcal{S} \mid \omega_r^{[\![\theta]\!]\omega} \text{ for all } r \in \mathbb{R} \in X\} \\
\delta_{x'=\theta\,\&\,Q}(X) &= \{\varphi(0) \in \mathcal{S} \mid \varphi(r) \in X \text{ for all } r \in \mathbb{R}_{\geq 0} \text{ and (differentiable) } \varphi : [0,r] \to \mathcal{S} \\
&\qquad \text{such that } \varphi(\zeta) \in [\![Q]\!] \text{ and } \frac{\partial \varphi(t)(x)}{\partial t}(\zeta) = [\![\theta]\!]\varphi(\zeta) \text{ for all } 0 \leq \zeta \leq r\} \\
\delta_{?\phi}(X) &= ([\![\phi]\!])^{\complement} \cup X \\
\delta_{\alpha\cup\beta}(X) &= \delta_\alpha(X) \cap \delta_\beta(X) \\
\delta_{\alpha;\beta}(X) &= \delta_\alpha(\delta_\beta(X)) \\
\delta_{\alpha^*}(X) &= \bigcup\{Z \subseteq \mathcal{S} \mid Z \subseteq X \cap \delta_\alpha(Z)\} \\
\delta_{\alpha^d}(X) &= (\delta_\alpha(X^{\complement}))^{\complement}
\end{aligned}$$

Note that this winning-region semantics is backward-chaining, which corresponds to the construction of winning regions by *backward induction* in game theory. The semantics are backward-chaining in the sense that we first commit to a goal region X, from which we can construct an winning region containing the winning *initial* states. This is in contrast to dL (Platzer, 2018a) and CGL (Chapter 5), both of which have a forward-chaining Kripke-style presentation, where the final region is constructed for a given initial region.

2.3 Proof Calculus

We recall a representative fragment of the proof calculus for dGL in Fig. 2.1. The proof calculus is given as a Hilbert system: axioms are repeatedly applied to recursively decompose hybrid games. While many basic axioms are the same in dL and dGL, some are necessarily different because dGL is subregular: for example, Kripke's axiom K fails in dGL and is in practice subsumed by rule M.

$$([\cdot]) \quad \neg\langle\alpha\rangle\neg\phi \leftrightarrow [\alpha]\phi$$

$$(\langle:=\rangle) \quad \phi(\theta) \leftrightarrow \langle x := \theta\rangle\phi(x)$$

$$(\langle:*\rangle) \quad \exists x\, \phi(x) \leftrightarrow \langle x := \theta\rangle\phi(x)$$

$$(\langle'\rangle) \quad \exists t{\geq}0\, \langle x := y(t)\rangle\phi \leftrightarrow \langle x' = \theta\rangle\phi$$

$$(\langle?\rangle) \quad \langle?\psi\rangle\phi \leftrightarrow (\phi \wedge \psi)$$

$$(\langle\cup\rangle) \quad \langle\alpha \cup \beta\rangle\phi \leftrightarrow \langle\alpha\rangle\phi \vee \langle\beta\rangle\phi$$

$$(\langle;\rangle) \quad \langle\alpha;\beta\rangle\phi \leftrightarrow \langle\alpha\rangle\langle\beta\rangle\phi$$

$$(\langle*\rangle) \quad \phi \vee \langle\alpha\rangle\langle\alpha^*\rangle\phi \rightarrow \langle\alpha^*\rangle\phi$$

$$(\langle^d\rangle) \quad \neg\langle\alpha\rangle\neg\phi \leftrightarrow \langle\alpha^d\rangle\phi$$

$$\text{(M)} \quad \frac{\phi \rightarrow \psi}{\langle\alpha\rangle\phi \rightarrow \langle\alpha\rangle\psi}$$

$$\text{(FP)} \quad \frac{\phi \vee \langle\alpha\rangle\psi \rightarrow \psi}{\langle\alpha^*\rangle\phi \rightarrow \psi}$$

Figure 2.1: Selected axioms and rules for dGL

We first discuss the individual axioms, then discuss how the axioms are used together in proofs. Axiom $[\cdot]$ says that the modalities $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ are dual to each other for arbitrary games α, whereas the axiom $\langle^d\rangle$ says game α^d is dual to α. In practical proofs it is often easier to say that the α^d operation switches players and then plays α, as in the derived axiom:

$$(\langle\cdot\rangle^d) \quad \langle\alpha^d\rangle\phi \leftrightarrow [\alpha]\phi$$

Axiom $\langle:=\rangle$ says assignments can be eliminated by substituting in the postcondition, while $\langle:*\rangle$ says random assignments can be converted to quantifiers. Axiom $\langle'\rangle$ says that if there exists a function $y(t)$ which solves an ODE, then the ODE can be eliminated by substituting in $y(t)$. This axiom gives the case where there is no domain constraint because ODEs with domain constraints are derivable from those without them. The practicality of this axiom depends on the complexity of $y(t)$. Especially when $y(t)$ is not a polynomial, it is preferable to reason with *differential invariants* (Platzer, 2018a). Axiom $\langle?\rangle$ says a test passes when the test condition holds. Axiom $\langle\cup\rangle$ says Angel chooses which game to play. Axiom $\langle;\rangle$ says game $\alpha;\beta$ is played by playing α first, then β in the resulting state. Axiom $\langle*\rangle$ says a repetition game can be played by choosing between playing zero rounds or playing some strategy for the first round followed by some strategy for the repetition. Axiom M says hybrid games satisfy monotonicity. Axiom FP says repetition α^* is a fixed point of α.

In practice, dGL proofs are done in a backward-chaining style by decomposing goal formulas. To this end, KeYmaera X implements dGL as a mix of Hilbert axioms and sequent-calculus, which is interderivable with a pure Hilbert-style system as in Fig. 2.1. Many constructs, such as tests, sequential compositions, choices, and deterministic assignments, can be decomposed without any special insight because there is a single canonical introduction or elimination rule. Human insight is required to supply invariants for loops and differential equations, and when instantiating first-order quantifiers, or equivalently non-deterministic assignments.

Once the game has been decomposed, the remaining goals are typically first-order logic over the reals. While this fragment of dGL is decidable in theory (Tarski, 1951, reprinting 1998), it has doubly-exponential complexity (Davenport & Heintz, 1988; Weispfenning, 1997), and so human insight is necessary in practice. In addition to instantiating quantifiers, typical interactive methods include weakening unnecessary assumptions, simplifying away arithmetic terms whose details are not needed for the proof, or applying a valid axiom of real arithmetic.

Chapter 3

Completed Work: Logical Foundations

A bulletproof theoretical foundation is essential to our approach to end-to-end verification, as it allows the Logician to conclude that our models are fully understood and our proofs sound. Logics for verification of hybrid systems and hybrid games are the main formal tool with which we wish to develop correct CPS's, and as such it is doubly important that the logics themselves are developed with the absolute highest degree of certainty in their correctness. In this chapter, we discuss two completed works by the author. These works not only attest that an adequate theoretical foundation has been laid for the remaining chapters of the thesis, but also that the author is adequately experienced with developing logical foundations for the proposed works to succeed.

3.1 Formalization of dL in Isabelle/HOL

When we perform proofs in dL (or likewise in dGL), the Logician wishes to know that the theorem we believe we have proved is indeed true with respect to the denotational semantics of dL, which are considered ground truth. This property is called soundness, and paper proofs of soundness are provided in the papers which developed a sequent calculus (Platzer, 2008) and uniform substitution (Platzer, 2017a) calculus for dL, of which the latter is the basis for the implementation of dGL in the theorem prover KeYmaera X (Fulton et al., 2015). However, proofs on paper are notoriously susceptible both to i) soundness errors by human logical blunder and ii) even more often, completeness errors by omission: paper proofs often focus on the simplest cases of the proof, providing no evidence that the implementation will continue to be correct in the most difficult cases, exactly when it is most in need of formal assurance. Errors of both kinds can be most robustly addressed by formalizing the entire development of the proof system in a general-purpose proof assistant such as Isabelle (Nipkow et al., 2002) or Coq (*Coq Proof Assistant*, 1989), starting from syntax and semantics up to axioms and ultimately a soundness theorem for an entire proof-checking procedure. Errors of the first kind are the easiest to address: assuming we have not accidentally exploited a bug in the host prover, an erroneous logical step in the mechanized proof will simply be rejected by the prover. To totally eliminate errors of the second kind requires a more ambitious effort: once we have formalized dL in another prover, how do we know that we have formalized every important feature and that we have not forgotten

one? After all, if we forget to include a certain feature in our paper proof, we might very easily forget it in a mechanized proof as well, and because this is not a soundness error, we will not catch it simply by attempting to prove soundness in the prover.

Errors of the second kind can be addressed experimentally by a change of perspective: instead of asking whether our proof matches the existing implementation, we can use a theorem prover's code generator to synthesize a proof checker from a formalization of the proof system. If that proof checker is willing to accept the dL proofs that are constructed in practice, then there is little question to the exhaustiveness of the development, because no matter whether the paper proof and mechanized proof forgot to account for some feature from the implementation, the implementation itself certainly cannot forget. While the work involved in bringing a mechanization up to speed with the practical implementation can be significant, there is also a second payoff: the resulting proof checker is guaranteed sound by construction, and can be used as a backup for the standard KeYmaera X prover core, effectively removing the core from the trusted computing base.

The author has formalized the dL uniform substitution calculus (Platzer, 2017a) using Isabelle (Nipkow et al., 2002) as a host prover, then validated the resulting verified checker. The initial formalization is documented in (Bohrer et al., 2017) and follow-up work in (Bohrer et al., 2018). In all, the formalization includes the syntax, static and dynamic semantics, proof calculus, and soundness theorem for dL. The full formalization is ≈25,000 lines of Isabelle proof scripts. In addition, a proof checker was synthesized with the Isabelle code generation mechanism and evaluated by instrumenting KeYmaera X to output complete proof terms as it performs a proof. In (Bohrer et al., 2018) this proof checker was evaluated on a proof that a velocity-controlled car driving in a straight line is safe under the supervision of a controller generated by VeriPhy. The proof term in question is non-trivial: ≈100,000 proof steps, which directly match the KeYmaera X proof. While the proof checker was built primarily to validate the dL formalization, it can also be used by Logic-User to ensure that the theorem-prover did not encounter buggy behavior in practical use.

These results contribute to the overall goals of the thesis in several ways. Firstly, the proof checker has been used to check the correctness of dL proofs, and dL proofs are used in this thesis to show that end-to-end verified systems are actually safe. In effect, this helped bridged the high-level, practical foundation given by dL with the lower-level foundations of Isabelle. Secondly, the work requires intimate knowledge of the foundations of dL, including edge cases which were not addressed in prior works such as i) subtle variable-renaming side conditions on rules for systems of (multiple) differential equations and ii) a bound renaming rule, which was in fact unsound in KeYmaera X until the bug was discovered by doing the mechanized proof. Validation with the synthesized proof checker ensured that the formalization was thorough. For example, this revealed the need to formalize operations like division, absolute value, and min and max, which are not covered in previous presentations. For these reasons, this chapter has prepared the author for the foundational work that remains in the proposed thesis chapters.

Note that we considered formalizing dL in a proof system with built-in support for higher-order abstract syntax (Pfenning & Elliott, 1988) (HOAS) such as Twelf (Pfenning & Schürmann, 1999), but decided against this because HOAS alone would only provide a free implementation of syntactic operations such as renaming and substitution, and would not solve our main challenge of showing that those operations are sound with respect to our denotational semantics. Our

denotational semantics then requires a formalization of real analysis. Moreover, the uniform substitution operation of dL allows substituting for an arbitrary free rigid symbol of an expression, whereas HOAS systems typically apply substitutions to some designated "argument" symbol.

3.2 Definite Descriptions in dL

A significant gap between theory presentations of dL and its implementation in KeYmaera X is that theory presentations of dL assume polynomial or (some) rational terms, while KeYmaera X allows extended terms featuring arbitrary division, exponents, and certain interpreted functions. The formalization summarized in Section 3.1 approached this gap with brute force: division by constants, absolute value, min, and were are added to the language of terms. However, the brute-force approach is brittle in multiple ways. Because every new construct comes with its own cases in the substitution algorithm and its own axioms, each new construct requires non-trivial changes to the Isabelle/HOL formalization of dL. This can only be done by people familiar with that formalization. The Logic-User would be disappointed that they must wait for the Logician to design, investigate, and implement new constructs in the theorem-prover rather than implementing new features themself. The extended constructs are not supported in their full generality, either, because dL expects that all terms are total (defined everywhere) and that continuous systems contain only (C^1-smooth) continuous terms, since the soundness proofs for dL axioms are simplest in that case.

Recent work by the author (Bohrer, Fernández, & Platzer, 2019) develops another approach which both provides these constructs in their full generality and provides a more promising foundation for future extensions, *without* increasing the size of the core when adding new constructs. We expect that defining new terms outside the core will enable extension work by the Logic-User and not only the Logician. We introduce the logic dL_ι (Fig. 3.1) which extends dL with a definite description construct $\iota x\ \phi(x)$ meaning the unique x (if a unique one exists) such that $\phi(x)$ is true. Term $\iota x\ \phi(x)$ takes on a special undefined denotation (written $\bot$) if no (or multiple) such x exists. Definite descriptions make terms as expressive as formulas, including non-polynomial terms which can be discontinuous or partial. Combined with dL_ι's pairing construct (θ, η), vector terms are also supported.

As illustrated in Fig. 3.1 in blue, dL_ι includes interesting foundational changes:

- Because terms are partial, dL_ι is a free logic with a 3-valued semantics
- The axioms and proof rules of dL are revised to be valid for dL_ι semantics. When an axiom requires terms to be continuous or total, the axioms now state these requirements explicitly. Significant theorems can be proven for non-Lipschitz ODEs, which cannot be expressed with dL's ODE constructor.
- The uniform substitution calculus of dL is generalized to dL_ι with modest changes, despite major semantic differences. This shows a clear path forward for any future implementation effort.

In the context of the thesis, this section shows that the extended term language used in practice is amenable to a foundational treatment like those provided by theory. This section also qualifies the author for the remaining foundational tasks: whereas Section 3.1 focused on a detailed re-

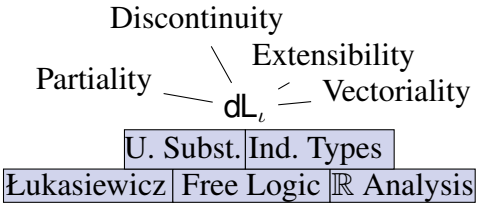


Figure 3.1: dL$_\iota$ features and technical challenges

production of an existing proof calculus, dL$_\iota$ required a wholesale revision of dL semantics, just as combining CGL and dGL into CdGL will. The theoretical developments for dL$_\iota$ culminate in soundness and equi-expressiveness proofs with respect to dL. Soundness validates the basic correctness of the calculus while equi-expressiveness says that decidability results for dL also apply to dL$_\iota$ because provable dL$_\iota$ formulas can be translated to provable dL formulas. Moreover, the theoretical results were good practice because these are very much the same kind of results we aim to achieve for the CdGL theory.

3.2.1 Related Work

As shown in Fig. 3.2, there are formalizations of KeYmaera X (Bohrer et al., 2018), Coq (Barras, 2010), Nuprl (Anand & Rahli, 2014), and HOL4 (Kumar et al., 2016), connecting each to its logical foundation. With the possible exception of HOL4, each formalization omits features which are theoretically challenging for their specific logic: discontinuous and partial terms in KeYmaera X, termination-checking in Coq, or context management in Nuprl. Thus only a subset of the implementation is guaranteed sound. When formalizations of theorem provers *do* succeed in reflecting the implementation (Kumar et al., 2016), they owe a credit to the generality of the underlying theory: it is much more feasible to formalize a general base theory than to formalize ad-hoc extensions as they arise. Our calculus, as with modern implementations (Fulton et al., 2015) and machine-checked correctness proofs (Bohrer et al., 2017) for dL, is based on uniform substitution (Church, 1956, §35,§40): symbols ranging over predicates, programs, etc. are explicitly represented in the syntax. This improves the ease with which dL$_\iota$ can be implemented and its soundness proof checked mechanically in future work. We show that in contrast to dL, dL$_\iota$ can directly represent a hybrid system featuring non-Lipschitz ODEs. The full paper studies an example based on Hubbard's leaky bucket (Hubbard & West, 2012, §4.2), a textbook example of a non-Lipschitz ODE. In contrast to the indexed variables of QdL (Platzer, 2012a), our non-scalar terms are equipped with an induction operator, making it easier to write sophisticated computations.

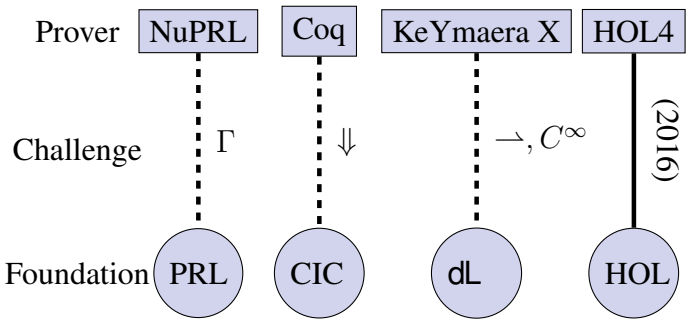


Figure 3.2: Prover formalizations and challenges

Chapter 4

Completed Work: End-to-End Robot Verification

This chapter describes works by the author and collaborators which introduce a general approach that enables end-to-end verification while maintaining solid logical foundations and then apply the approach in a case study on the verification of 2D free-driving ground robots. The robots follow sequences of arcs and lines (known as Dubins paths (Dubins, 1957)) with the additional feature of enforcing speed limits. These works already constitute a major component of the goals of the thesis: what remains in the proposed work is to extend this beyond only verification and monitor synthesis to include controller synthesis while maintaining rigorous foundations. We begin by motivating end-to-end verification and defining it, then give our approach and case study results in greater detail.

4.1 VeriPhy Pipeline for End-to-End Verification

Recall our notion of end-to-end verification from Chapter 1: we wish to produce formal guarantees that apply to practical implementations of a CPS by proving that the implementation refines a provably safe hybrid system model. We wish for the refinement process to be fully automatic for the sake of the Engineer who builds the CPS and does not want to do formal proofs. Granted, the Engineer will need the Logic-User's help to verify the hybrid system model, but this is done only once: it is more important that we need not redo manual proofs each time the Engineer changes their code slightly. At the same time, each refinement step should provide machine-checkable evidence, in order to satisfy the Logician that the proof is correct. Our approach, which we will refer to as the VeriPhy approach (as it is implemented in our tool VeriPhy), achieves all of the above, as we will now describe.

In light of the fact that a real Engineer might use languages such as Matlab where formal verification is difficult, and that their code may change hundreds of times per day, the VeriPhy approach employs runtime monitoring so that the Engineer's controller is untrusted. Specifically, we monitor whether the implementation complies to the source-model. The Logic-User wants source-model verification to be as simple as possible and the Logician wants safety of the model to closely match an intuitive notion of real-world safety. For these reasons, the source-model is

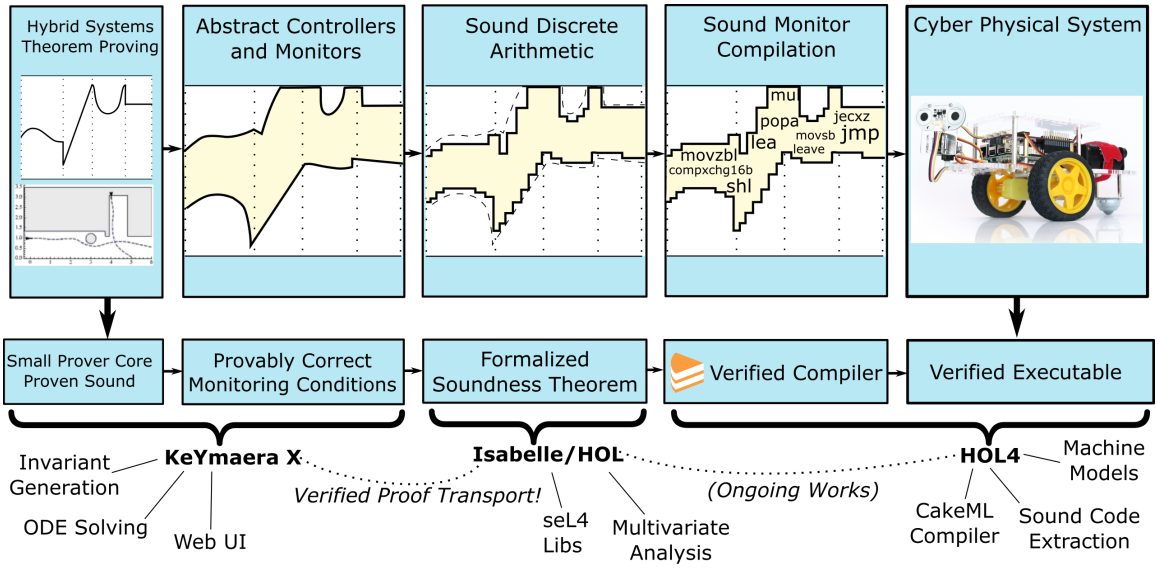


Figure 4.1: High assurance artifacts and steps in the VeriPhy verification pipeline

a hybrid program, verified in KeYmaera X with dL.

- The ModelPlex (Mitsch & Platzer, 2016) tool, which is implemented as a feature of KeYmaera X, is invoked to synthesize a *control monitor formula* and *plant monitor formula*, which determine whether control decisions and physical evolutions respectively satisfy the assumptions made of them by the model.
- The monitors are combined with a proven-safe *fallback controller* (which is provided in the input model) to form a *sandbox controller*. The sandbox controller applies control decisions provided from an untrusted source (henceforth called the *external controller*) whenever those decisions satisfy the controller monitor, and otherwise invokes the fallback controller.
- The sandbox controller is automatically proven safe in KeYmaera X, in part by reusing the contents of the hybrid system safety proof.
- The sandbox safety proof is optionally rechecked in the verified proof checker (Section 3.1) to eliminate the KeYmaera X core from the trusted base. As discussed in Section 3.1, this has also been used to validate the dL formalization for exhaustiveness.
- The exact real arithmetic from hybrid systems semantics ($[\![\cdot]\!]$) is conservatively recast as fixed-point interval arithmetic ($[\![\cdot]\!]$), and this recasting is formally proven in Isabelle to be sound.
- The resulting program is automatically proven equivalent (in HOL4) to a CakeML (Kumar et al., 2016) program. The heart of the proof says that sensing and actuation in the model can be modeled by a CakeML state machine (Ho et al., 2018).
- The equivalent CakeML program ($[\![\texttt{cmlSandbox}]\!]$) is compiled to machine code in the CakeML verified compiler

- The machine code ($\{|\texttt{CML(cmlSandbox)}|\}$) is linked against the user's trusted implementation of sensors and actuators

Each step is provably a refinement (Fig. 4.2) of the step before it, culminating in the fact that any execution of the machine-code running on the actual CPS corresponds to some execution of the source hybrid system. Because all executions of the source hybrid system were proven safe already, then the execution of the actual CPS is also known to be safe. The differing brackets at each level represent that states at each level have differing types, e.g. reals vs. intervals, and these types are converted accordingly in each refinement theorem. The CakeML sandbox (`cmlSandbox`) is generated by importing the monitor conditions and fallback used in the dL sandbox.

Note that the sensor and actuator drivers, which are provided by the users of VeriPhy, are trusted. We justify this by noting that verification of sensing and actuation is a moving target: their hardware changes frequently, and their verification would require rewriting models and their proofs every time that hardware changes. Because models of hardware still require assumptions on the physical behavior of the hardware's components, even this would not eliminate the trusted hardware base. For these reasons, VeriPhy prioritizes providing a simple and well-specified interface for sensing and actuation, which leaves open the possibility of verifying these components in future work. This interface consists of a handful of CakeML foreign-functions (Ho et al., 2018) for reading from and writing to the external world, which have precise specifications in HOL4 and can then be implemented in the VeriPhy user's language of choice, usually C. Fig. 4.2 shows the chain of formal safety properties proven for the artifacts given in Fig. 4.1. One

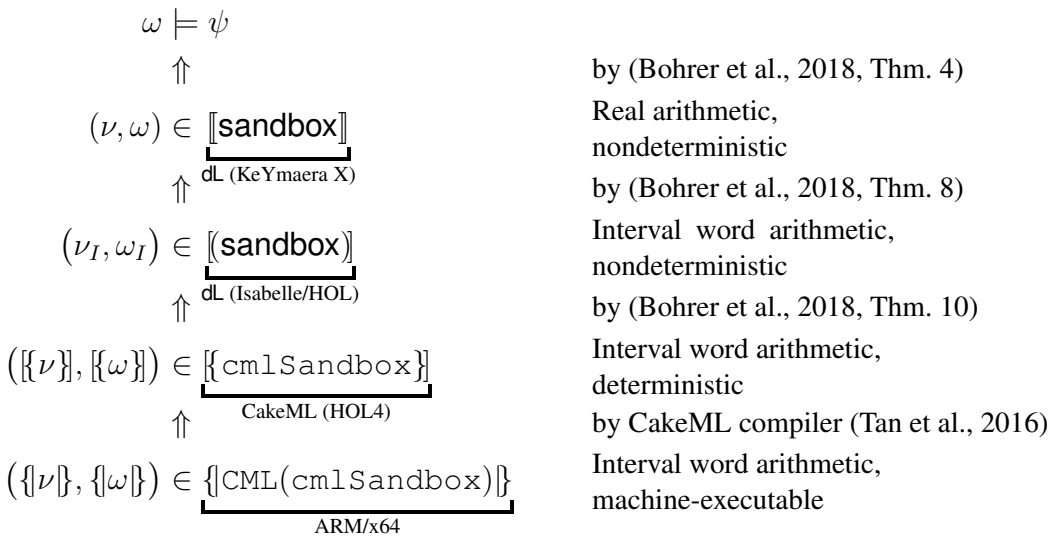


Figure 4.2: End-to-end proof chain for end-to-end result

major challenge is that our notion of end-to-end verification demands crossing multiple levels of abstraction, from hybrid systems down to discrete arithmetic and finally machine-code. For this reason, we incorporated several provers: KeYmaera X is used for the source-model proof, Isabelle/HOL is used for the translation into discrete arithmetic, and HOL4 with its verified

CakeML compiler is used for final compilation to machine-code. If we wish to verify sensing and actuation in the future, this could be done by proving in HOL4 that the hardware drivers implement their specifications.

4.1.1 Related Work

Verified Compilation. We use the CakeML (Tan et al., 2016) verified ML compiler and its associated verification tools (Myreen & Owens, 2012; Guéneau et al., 2017). Recent work (Hupel & Nipkow, 2018) also provides verified extraction of CakeML code from Isabelle/HOL proofs, but its present iteration does not support all features needed in this work, such as locales, inductively defined predicates, and the built-in set type. CakeML is higher-level than other languages that have verified compilers, such as CompCert (Leroy, 2006) for C (for floating-point support, see (Boldo et al., 2013)) or Jinja (Klein & Nipkow, 2006) for a Java-like language. The smaller gap between CakeML source and Isabelle/HOL definitions makes the verification of sandbox implementation in CakeML against hybrid systems semantics painless. We chose CakeML compilation over, e.g., translation validation with unverified compilers (Sewell et al., 2013) since translation validation can be brittle.

Lustre (Halbwachs et al., 1992) is a reactive, synchronous language intended for use in safety-critical CPS. It has static analyzers based on abstract interpretation such as NBac (Jeannet, 2003), which eliminate many bugs in practice, but are fundamentally conservative and also consider only the control code, not the physics in the plant. Lustre has a verified compiler, Vélus (Bourke et al., 2017), but verified compilation *alone* does not negate the fact that verification for CPS must account for physics. We could have chosen Lustre as an intermediate compilation target instead of CakeML, but the functional style of CakeML is a much closer match to Isabelle/HOL definitions, while Lustre's reactive model is an entirely different paradigm from both HOL and hybrid programs.

Machine Arithmetic Verification. Machine arithmetic correctness is a major VeriPhy component. We verify arithmetic soundness foundationally. This is an active research area with libraries available in HOL Light (Harrison, 2006a), Coq (Boldo & Melquiond, 2011; Daumas et al., 2001; Boldo et al., 2009; Melquiond, 2012), Isabelle/HOL (Yu, 2013), etc. The main results we need are results saying that basic arithmetic operations round in the direction specified by the rounding mode. While it is possible others have proved such results, only PFF in Coq (Daumas et al., 2001) has documented such results explicitly. For this reason, we proved rounding results ourselves in Isabelle/HOL using the seL4 (Klein et al., 2010) machine word library. We chose Isabelle/HOL and HOL4 over Coq because their combination of cutting-edge analysis libraries (Immler & Traut, 2016), mature formalization of dL (Bohrer et al., 2017), proof-producing code extraction (Myreen & Owens, 2012), and classical foundations positions them well for our end-to-end pipeline. Standalone programs (Beyer & Huisman, 2018) have also been verified in HOL and Coq (Becker et al., 2018) which certify error-bounds on the outputs of arithmetic expressions. Static analysis programs have been written (Martinez et al., 2010; Majumdar et al., 2012; Bouissou et al., 2009) to detect arithmetic errors in the context of hybrid systems, but these programs do not have foundational correctness guarantees and support only

limited classes of ODEs.

4.2 Case Study: End-to-End Ground Robotics Verification

To validate our claim that VeriPhy enables end-to-end verification for dL, we have applied VeriPhy to software and hardware implementations of robots (Bohrer, Tan, et al., 2019). Some goals of this chapter are to: i) provide proof-of-concept that accurate simulations and hardware implementations of robots safely achieve mission objectives under VeriPhy control, ii) identify limitations of the present VeriPhy implementation from the perspective of the Logician, Logic-User, and Engineer, and iii) propose steps to counter any limitations. The initial validation work (Bohrer et al., 2018) considered only verified a simple 1D velocity-controlled car (Example 1), which we ran on consumer-grade robot hardware. In this section, we develop a vehicle model which follows piecewise curved (i.e., Dubins (Dubins, 1957)) paths with speed limits. This is far more representative of realistic driving scenarios than the initial work, because speed limits and steering are fundamental to driving. By simple virtue of requiring a larger model and more complicated invariants, it has stress-tested the VeriPhy implementation. Already, this work has helped us identify and fix a number of "silly" bugs, e.g., where VeriPhy made overly strict assumptions on the input model or used excessively brute-force, slow proof approaches. Looking forward, this work has highlighted the challenges of debugging a controller supervised by synthesized software: when a monitor fails, the Engineer needs feedback as to why and how it failed, so they may assess whether the controller implementation or the model is to blame. The Logic-User also needs feedback on the correctness of the model. In our case, the Engineer's implementation provided this feedback, but in future work simulations could be synthesized which demonstrate the model's behavior for the Logic-User. Our contributions in this case study include dL models, proofs, a simulation which employs the sandbox controller generated by VeriPhy, and test environments for the simulation. One possible extension for the thesis would be to evaluate this case study on a full-size MRZR (Polaris Industries, 2019) vehicle owned by collaborators.

4.2.1 Model

This section introduces our 2D robot model in dL. The circular Dubins dynamics are bloated by a tolerance, which accounts for imperfect actuation and for discrepancies between the Dubins dynamics and real dynamics of the implementation. That is, because realistic robots never follow a path perfectly, we will bloat each arc to an annulus section which is more easily followed, shown in Fig. 4.3. Each waypoint is specified by coordinates (x, y), a curvature k, and a speed limit $[\mathsf{vl}, \mathsf{vh}]$ for the robot's velocity v which is to be met by the time the waypoint is reached. Curvature parameter $\mathsf{k} \neq 0$ yields curved arc sections while $\mathsf{k} = 0$ yields line segments. Together, these are the primitives of Dubins paths.

Our hybrid program α is a time-triggered *control-plant* loop: $\alpha \equiv (\mathsf{ctrl}; \mathsf{plant})^*$. We use *relative* coordinates: the robot's position is always the origin, from which perspective the waypoint "moves toward" the vehicle (Fig. 4.3). This simplifies proofs (fewer variables) and implementation (real sensors and actuators are vehicle-centric).

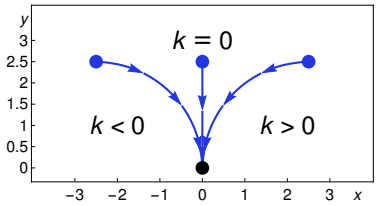


Figure 4.3: Trajectories of dynamics for different choices of k.

The plant is an ODE describing the robot kinematics:

$$\mathsf{plant} \equiv \{\mathsf{x}' = -\mathsf{v}\,\mathsf{k}\,\mathsf{y},\ \mathsf{y}' = \mathsf{v}\,(\mathsf{k}\,\mathsf{x} - 1),\ \mathsf{v}' = \mathsf{a},\ \mathsf{t}' = 1 \\ \&\ \mathsf{t} \leq \mathsf{T}\ \wedge\ \mathsf{v} \geq 0\}$$

Here, a is an input from the controller describing the acceleration with which the robot is to follow the arc of curvature k to waypoint (x, y). In the equations for x', y': *i)* The v factor models (x, y) moving at linear velocity v, *ii)* The $\mathsf{k}, \mathsf{x}, \mathsf{y}$ factors model circular motion with curvature k, with $\mathsf{k} > 0$ corresponding to counter-clockwise rotation of the waypoint, $\mathsf{k} < 0$ to clockwise rotation, and $\mathsf{k} = 0$ to straight-line motion, *iii)* The additional -1 term in the y' equation shifts the center of rotation to $\left(\frac{1}{\mathsf{k}}, 0\right)$. The equations $\mathsf{v}' = \mathsf{a}$ and $\mathsf{t}' = 1$ make acceleration the derivative of velocity and t stand for current time. The domain constraint $\mathsf{t} \leq \mathsf{T} \wedge \mathsf{v} \geq 0$ says that the duration of one control cycle shall never exceed a timestep parameter $\mathsf{T} > 0$ representing the maximum delay between control cycles and that the robot never drives in reverse. Fig. 4.4 illustrates a goal of size $\varepsilon = 1$ around the origin and several trajectories which pass through the goal.

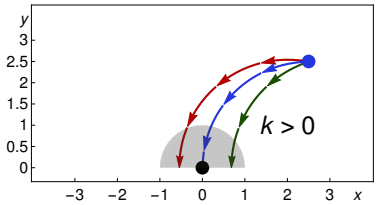


Figure 4.4: Trajectories of plant for choices of $\mathsf{k} < 0$ when $\varepsilon = 1$.

The controller's task is to compute an acceleration a which slows down (or speeds up) soon enough that the speed limit $\mathsf{v} \in [\mathsf{vl}, \mathsf{vh}]$ is ensured *by the time* the robot reaches the goal. The

controller is written:

$$\begin{aligned}
\mathsf{Ann} &\equiv |\mathsf{k}|\varepsilon \leq 1 \wedge \left|\frac{\mathsf{k}\ (\mathsf{x}^2+\mathsf{y}^2-\varepsilon^2)}{2} - \mathsf{x}\right| < \varepsilon \\
\mathsf{Feas} &\equiv \mathsf{Ann} \wedge \mathsf{y}{>}0 \wedge 0{\leq}\mathsf{vl}{<}\mathsf{vh} \wedge \mathsf{AT}{\leq}\mathsf{vh}-\mathsf{vl} \wedge \mathsf{BT}{\leq}\mathsf{vh}-\mathsf{vl} \\
\mathsf{ctrl} &\equiv (\mathsf{x},\mathsf{y}) := *;\ [\mathsf{vl},\mathsf{vh}] := *;\ \mathsf{k} := *;\ ?\mathsf{Feas};\ \underbrace{\mathsf{a} := *;\ ?\mathsf{Go}}_{\mathsf{ctrl}_\mathsf{a}}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Go} &\equiv -\mathsf{B} \leq \mathsf{a} \leq \mathsf{A} \wedge \mathsf{v}+\mathsf{aT} \geq 0 \\
&\wedge \Big(\mathsf{v} \leq \mathsf{vh} \wedge \mathsf{v}+\mathsf{aT} \leq \mathsf{vh} \vee \\
&\quad (1+|\mathsf{k}|\varepsilon)^2\Big(\mathsf{vT}+\frac{\mathsf{a}}{2}\mathsf{T}^2+\frac{(\mathsf{v}+\mathsf{aT})^2-\mathsf{vh}^2}{2\mathsf{B}}\Big)+\varepsilon{\leq}\|(\mathsf{x},\mathsf{y})\|_\infty\Big) \\
&\wedge \Big(\mathsf{vl} \leq \mathsf{v} \wedge \mathsf{vl} \leq \mathsf{v}+\mathsf{aT} \vee \\
&\quad (1+|\mathsf{k}|\varepsilon)^2\Big(\mathsf{vT}+\frac{\mathsf{a}}{2}\mathsf{T}^2+\frac{\mathsf{vl}^2-(\mathsf{v}+\mathsf{aT})^2}{2\mathsf{A}}\Big)+\varepsilon{\leq}\|(\mathsf{x},\mathsf{y})\|_\infty\Big)
\end{aligned}$$

where the plan assignment $(\mathsf{x},\mathsf{y}) := *$ chooses the next 2D waypoint, the assignment $[\mathsf{vl},\mathsf{vh}] := *$ chooses the speed limit interval, and $\mathsf{k} := *$ chooses any curvature. The *feasibility* test $?\mathsf{Feas}$ determines whether or not the chosen waypoint, speed limit, and curvature are *physically* attainable in the current state under the plant dynamics (e.g., it checks that there is enough remaining distance to get within the speed limit interval). We also simplify plans so all waypoints satisfy $\mathsf{y} > 0$, by subdividing any violating path segments automatically. Bisecting the arc once always suffices because any arc is at most magnitude 2π, so the bisected arc is at most magnitude π and thus satisfies $\mathsf{y} > 0$. It is worth this effort to gain the assumption $\mathsf{y} > 0$ because it simplifies the external controller and proofs. The abbreviation ctrl_a names just the control code responsible for deciding *how* the waypoint is followed rather than *which* waypoint is followed.

In Feas, formula Ann says we are within the *annulus section* (Fig. 4.5) ending at the waypoint (x,y) with the specified curvature k and width ϵ. A larger choice of ε yields more error tolerance in the sensed position and followed curvature at the cost of an enlarged goal region. Formula Ann also contains a simplifying assumption that the radius of the annulus is at least ϵ. Feas also says the speed limits are assumed distinct and large enough to not be crossed in one control cycle.

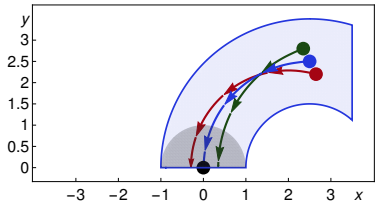


Figure 4.5: Annular section through the (blue) waypoint $(2.5, 2.5)$. Trajectories from the displaced green and red waypoints with slightly different curvatures remain within the annulus.

The *admissibility* test $?\mathsf{Go}$ checks that the chosen a will take the robot to its goal with a safe speed limit, by *predicting future motion* of the robot. We illustrate this with the upper bound

conditions. The bound will be satisfiable after one cycle if either the chosen acceleration a already maintains speed limit bounds ($\mathsf{v} \leq \mathsf{vh} \wedge \mathsf{v}+\mathsf{a}\mathsf{T} \leq \mathsf{vh}$) or when there is enough distance left to restore the limits before reaching the goal. For straight line motion ($\mathsf{k} = 0$), the required distance can simply be found by integrating acceleration and speed:

$$\underbrace{\mathsf{v}\mathsf{T} + \frac{\mathsf{a}}{2}\mathsf{T}^2}_{\text{distance in time } \mathsf{T}} + \frac{\overbrace{(\mathsf{v}+\mathsf{a}\mathsf{T})}^{\text{speed in time } \mathsf{T}})^2 - \mathsf{vh}^2}{2\mathsf{B}} + \varepsilon \leq \|(\mathsf{x},\mathsf{y})\|_\infty$$

where $\mathsf{a} \in [-\mathsf{B}, \mathsf{A}]$. The extra factor of $(1 + |k|\varepsilon)^2$ in formula Go accounts for the fact that an arc along the inner side of the annulus is shorter than one along the outside (Fig. 4.5).

4.2.2 Proofs

In contrast to prior works (Mitsch et al., 2017), this work contains both safety and liveness proofs for the full 2D dynamics.[1] In many ways, this lays a foundation for the proposed work because a proof of winnability for a hybrid game will contain both liveness-like (strategy for our player) and safety-like components (strategy for the opponent). Taken at surface value, the safety theorem says that speed limits are obeyed whenever the robot reaches a waypoint

Theorem 1 (Safety)**.** *The following* dL *formula is valid:*

$$\begin{aligned}&\mathsf{A}{>}0 \wedge \mathsf{B}{>}0 \wedge \mathsf{T}{>}0 \wedge \varepsilon{>}0 \wedge J \rightarrow \\ &[(\mathit{ctrl};\mathit{plant})^*]\big(\|(\mathsf{x},\mathsf{y})\| \leq \varepsilon \rightarrow \mathsf{v} \in [\mathsf{vl},\mathsf{vh}]\big)\end{aligned}$$

Taken more broadly, safety also includes the fact that the robot always remains within an ϵ annulus around the arc leading to the waypoint, which arises as part of the loop invariant used to prove Theorem 1. The first four assumptions ($\mathsf{A}{>}0 \wedge \cdots \wedge \varepsilon{>}0$) are basic sign conditions on the symbolic constants used in the model. The final assumption, J, is the loop invariant. The full definition of J is deferred to Section 4.2. We write $\|(\mathsf{x},\mathsf{y})\|$ for the Euclidean norm $\sqrt{\mathsf{x}^2+\mathsf{y}^2}$ and consider the robot "close enough" to the waypoint when $\|(\mathsf{x},\mathsf{y})\| \leq \varepsilon$ for our chosen goal size ε. While this captures the desired notion of safety, it does not prove that the robot can actually reach the goal, which is a *liveness* property:

Theorem 2 (Liveness)**.** *The following* dL *formula is valid:*

$$\begin{aligned}&\mathsf{A}{>}0 \wedge \mathsf{B}{>}0 \wedge \mathsf{T}{>}0 \wedge \varepsilon{>}0 \wedge J \rightarrow \\ &[(\mathit{ctrl};\mathit{plant})^*]\Big(\mathsf{v}{>}0 \wedge \mathsf{y}{>}0 \rightarrow \\ &\quad\langle(\mathit{ctrl}_a;\mathit{plant})^*\rangle\big(\|(\mathsf{x},\mathsf{y})\| \leq \varepsilon \ \wedge\ \mathsf{v} \in [\mathsf{vl},\mathsf{vh}]\big)\Big)\end{aligned}$$

This theorem has the same assumptions as Theorem 1. It says that no matter how long the robot has been running ($[(\mathsf{ctrl};\mathsf{plant})^*]$), then if some simplifying assumptions still hold ($\mathsf{v}{>}0 \wedge$

[1] In the interest of full disclosure, the modeling and proof sections were largely performed by collaborators in this work. The present author provided simulations, writing, integration with VeriPhy, and advice on the modeling and proof tasks. In contrast, the modeling and proofs in the proposed work are to be performed by the present author.

$\mathsf{y}>0$) the controller can be continually run ($\langle(\mathsf{ctrl}_a;\mathsf{plant})^*\rangle$) with admissible acceleration choices (ctrl_a) to reach the present goal ($\|(\mathsf{x},\mathsf{y})\| \leq \varepsilon$) within the desired speed limits ($\mathsf{v} \in [\mathsf{vl},\mathsf{vh}]$). The simplifying assumptions $\mathsf{v}>0 \wedge \mathsf{y}>0$ say the robot is still moving forward and the waypoint is still in the upper half-plane, i.e., we have not run *past* the waypoint.

The liveness theorem is invaluable because it validates the dL model: if it were not even possible at the level of the dL model to achieve liveness, then it would be absolutely impossible at the level of implementation to achieve liveness. Because it is undesirable to change the model drastically once implementation has started (i.e., delays in modeling and verification will trickle down to greater delays in implementation and testing), it is valuable to have this confidence in the model before implementation even begins. Moreover, the combination of safety and liveness proofs prepare us for the proposed work of generalizing this case study to a hybrid game: proofs about an Angelic controller proceed like the liveness proof, while proofs about a Demonic plant proceed like the safety proof.

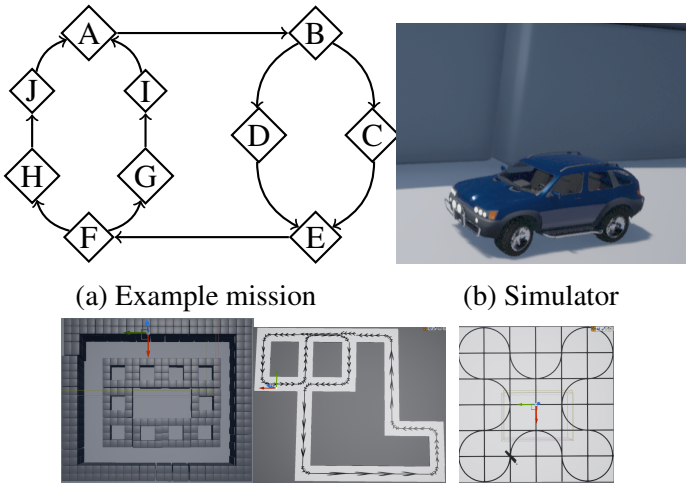
4.2.3 Simulations

In order to claim that VeriPhy has been evaluated on a realistic system implementation, it is essential that we choose a simulation platform that is reasonably faithful to the physics of actual autonomous cars. For this purpose we chose the autonomous driving simulation feature of the AirSim (Shah et al., 2018) simulator, which is notable for combining top-notch visuals, reasonable physics, and an open-source code base that is particularly friendly to modification. While some aspects of the underlying physics engine are closed-source, many aspects are open to customization, including for example the details of wheels and suspensions, and the center of gravity. For this reason and because AirSim has been successfully applied in dozens of projects, we have good reason to believe its physics model is more faithful than any purpose-built one-off simulation would be.

In AirSim, the author implemented several basic control algorithms known as *bang-bang* and *proportional-derivative*. The author developed several driving environments for testing, shown in Fig. 4.6 The simulations showed that while it is rare to achieve zero errors, failure rates in the single digits are attainable, which suffice to complete all environments. Moreover, a major challenge we solved was to develop a simulation which worked seamlessly with the monitors, themselves synthesized from the model. The model and proof were iteratively developed using feedback from the resulting monitors. Our failure rates were as low as 0.1% for the control monitor and 4.0% for the plant monitor, which attested that our simple model was still general enough to express control decisions and physical dynamics that arise in some practical scenarios. Thus our formal link from dL to CPS execution can be made to work in practice for nontrivial driving scenarios.

4.2.4 Related Work

Synthesis and verification for robots as well as end-to-end verification have been discussed at length in the main related work section (Section 1.1). This chapter is unique compared to prior works in its use of a verified-safe sandbox controller to enforce compliance between the implementation and formal model for a realistic robot simulation with an expressive correctness guar-



(a) Example mission (b) Simulator

(c) Rectangle (d) Tight turns (e) Large clover

Figure 4.6: Implementation and environments built in AirSim

antee. In this section, we discuss only related works on simulation, which were not discussed in the broader section.

Simulation is an essential part of evaluating models and designs for any robotic system. Multiple simulation platforms are available, of which AirSim (Shah et al., 2018) is a recent platform for UAVs and autonomous cars. Other simulators would likely have worked as well, but we chose AirSim because it is configured with high-fidelity physical and visual models out of the box, reducing the chances of introducing modeling errors. An additional benefit of using an established simulation platform is that reduces the risk of "overfitting" the simulation to our model or code in any way that might yield more positive results than an independent evaluation would warrant.

Chapter 5

Completed Work: CGL: Constructive Game Logic

Throughout the previous chapters, the glue that unified the Logician's high-level abstract models with the Engineer's low-level implementations has been dL for *classical* hybrid *systems*. While Chapter 4 showed that the connection between dL models and implementations holds even in realistic case studies, it has also revealed the room that dL leaves for improvement as such a glue. We argue that the restrictions to *systems* and to *classical* reasoning pose limitations for the Engineer and Logic-User. The previous sections of the thesis have revealed several weaknesses from the practitioner's perspective:

- VeriPhy synthesizes only monitors, not controllers. The Logician and some Engineers would prefer to synthesize both, because the synthesized controller is guaranteed to satisfy the controller monitor, while a hand-written controller is not. Constructivity is important for synthesis because classical instances of the law of excluded middle allow uncomputable case analyses, for which synthesizing an implementation is a non-starter. In practice, such case analyses arise in almost every diamond proof for almost every controller: when a controller has multiple cases, a case analysis is performed to select a case. Thus the distinction between classical and constructive is not a mere curiosity, but arises in almost every control synthesis application.
- Experience suggests the current implementation of VeriPhy is not robust to complex models, and that the key to providing robustness is to make the key insights from the hybrid *proof* available to the synthesizer. The classical Hilbert calculus of dL complicates this because interpretations of classical logics and Hilbert calculi as computation are complex and indirect. We wish to provide a constructive logic with a natural-deduction calculus, so that proofs can be understood as programs that *implement* monitoring and control, so that the synthesizer simply compiles proofs to executable form. We show that proofs for *games* (systems + adversarial dynamics) elegantly allow a single model and proof to provide both the monitor and the control.
- VeriPhy uses fixed-point intervals to implement arithmetic computations. In the present implementation, limited arithmetic precision has proven to be a significant hurdle for users. Of the many arithmetic options available, constructive logic provides a conclusive solution:

implement arithmetic with computable reals. The cost of this decision is that computable reals have a higher space and time complexity than interval arithmetic, and the payoff is the elimination of mysterious monitor failures at runtime caused by insufficient arithmetic precision.

- Games allow several model simplifications over systems, which help the Logic-User. Game logic internalizes modality alternation into the language of games. This promotes a proof style where controller proofs are Angelic ($\langle\alpha\rangle\phi$) while plant proofs are Demonic ($[\alpha]\phi$). In contrast, a systems safety proof treats both the controller and plant as Demonic. If we recall the contrast between Example 3 and Example 1, we see that Demonic controller models are typically much more complicated than their Angelic counterparts, because they explicitly give an envelope of safe control choices, which is left implicit in the Angelic model. In non-trivial models such as Chapter 4, this control envelope is a significant contributor to overall model complexity. Furthermore, practical experience shows that modeling mistakes involving *totality* are common: it is easy to write a controller which has nonexhaustive case analyses, making universal safety theorems *vacuous*. Existential controllers obviate this issue, because safety will *fail to prove* if an existential controller were not total.

Looking beyond the core theme of end-to-end verification, the Logician and Logic-User both have additional reasons to pursue constructive proof. All the following concerns are addressed by the combination of games and constructivity:

- Many users find formal proof more intimidating to learn than code. A computational interpretation opens the possibility of writing CGL proofs in a familiar programming syntax, by writing code that implements a strategy. We suggest that this could ease adoption of game proofs.
- As discussed in Section 5.1, constructive program logics are nearly unexplored. This is surprising because the Curry-Howard correspondence, wherein propositions are interpreted types and proofs are interpreted as programs, has for many years been the backbone of cross-pollination between programming languages and logic research. CGL gives the first Curry-Howard interpretation of Game Logic, and the most exhaustive for any dynamic logic. Proofs of $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ are understood computationally as winning strategies for a computational player who moves first or second, respectively. They are respectively understood as *controlling* program α to achieve ϕ or *monitoring* an execution for compliance with α, which ensures ϕ as postcondition. This substantiates the claim that such proofs suffice for synthesis, and we posit that this correspondence is applicable to *any* nondeterministic program logic. CGL generalizes First-order Dynamic Logic and (Parikh's) Game Logic, which itself generalizes (Pratt's) PDL and (Peleg's) Concurrent Dynamic Logic. Thus CGL includes the first serious Curry-Howard treatment for all the above logics as well.

  Specifically, we are the first to feature a full proof-theoretic treatment based on proof terms and their normalization. We then prove meta-theoretical properties such as the Existence Property (Lemma 11) which attest to CGL's constructivity by showing all provable existential formulas have computable witnesses.

This chapter covers the discrete first-order CGL (Bohrer & Platzer, 2020), which will be extended to *hybrid* games (CdGL) in Chapter 6. We treat the discrete fragment first in large part

because there is such a significant gap in the literature, and the discrete result is a contribution of broader interest to the constructive logic community, which stands on its own.

5.1 Related Work

This work is at the intersection of game logic and constructive modal logics. Individually, they have a rich literature, but little work has been done at their intersection. Of these, we are the first for GL and the first with a proofs-as-programs interpretation for a full first-order program logic.

Games in Logic Propositional GL was introduced by (Parikh, 1983), followed by coalitional GL (Pauly, 2002). A first-order version of GL is the basis of differential game logic dGL (Platzer, 2015) for hybrid games. Logics with explicit strategies are forerunners to CGL, such as Strategy Logic (Chatterjee et al., 2007), Alternating-Time Temporal Logic (ATL) (Alur et al., 2002), CATL (van der Hoek et al., 2005), Ghosh's SDGL (Ghosh, 2008), and Ramanujam's structured strategies (Ramanujam & Simon, 2008). Dynamic-epistemic (van Benthem, 2015; van Benthem et al., 2011; Van Benthem, 2001) and evidence logics (van Benthem & Pacuit, 2011) are also used for strategic reasoning with an emphasis on knowledge, and Hoare calculi, while more restrictive, have been equipped with Angelic nondeterminism (Mamouras, 2016). In the above, strategies are reified in the *formula* language, if at all. In contrast, we reify them in the *proof* language, which crucially preserves the simplicity of the GL formula language. We argue the simple GL-style formula language promotes practical proving, while our clean separation between formulas and strategies aids our theoretical developments by which we justify that CGL is constructive. Prior logical explorations of *hybrid* games (Quesel & Platzer, 2012; Platzer, 2015, 2017b, 2018b) are classical.

Constructive Modal Logics A major contribution of CGL is our constructive semantics for games, not to be confused with game semantics (Abramsky et al., 2000), which are used to give programs a semantics *in terms of* games. We draw on work in semantics for constructive modal logics, of which two main approaches are intuitionistic Kripke semantics and realizability semantics.

An overview of Intuitionistic Kripke semantics is given by (Wijesekera, 1990). Intuitionistic Kripke semantics are parameterized over worlds, but in contrast to classical Kripke semantics, possible worlds represent what is currently *known* of the world. Worlds are preordered by $w_1 \geq w_2$ when w_1 contains at least the knowledge in w_2. Kripke semantics were used in Constructive Concurrent DL (Wijesekera & Nerode, 2005), where both the world and knowledge of it change during execution. A key advantage of realizability semantics (van Oosten, 2002; Lipton, 1992) is their explicit interpretation of constructivity as computability (Bauer, 2005) by giving a *realizer*, a program which witnesses a fact. Our semantics combine elements of both: Strategies are represented by realizers, while the game state is a Kripke world. Constructive set theory (Aczel & Gambino, 2006) aids in understanding which set operations are permissible in constructive semantics.

Modal semantics have also exploited mathematical structures such as: i) Neighborhood models (van Benthem et al., 2017), topological models for spatial logics (van Benthem & Bezhan-

ishvili, 2007), hybrid systems (Artëmov et al., 1997), and temporal logics of dynamical systems (Fernández-Duque, 2018). ii) Categorical (Alechina et al., 2001), sheaf (Hilken & Rydeheard, 2001), and pre-sheaf (Ghilardi, 1989) models. iii) Coalgebraic semantics for classical Propositional Dynamic Logic (PDL) (Doberkat, 2011). While games are known to exhibit algebraic structure (Goranko, 2003), such laws are not essential to this work. Our semantics are also notable for the seamless interaction between a constructive Angel and classical Demon.

Because CGL is first-order, a constructive approach must also address the constructivity of operations that inspect game state. The base sorts of CGL include both natural numbers, where exact equality and case-analysis are computable, and real numbers, where exact equality is not computable. Throughout, we use the reals of Bishop and Bridges (Bishop, 1967; Bridges & Vita, 2007), which contain the same uncountably-many numbers as the classical reals $\mathbb{R}$, but do not admit decidable equality. A realizability interpretation of Bishop's analysis has been explored (Schwichtenberg, 2008). Constructive reals have been implemented efficiently in theorem provers, for example the CORN (Krebbers & Spitters, 2011) library.

Constructivity and Dynamic Logic With CGL, we bring to fruition several past efforts to develop constructive dynamic logics. Prior work on PDL (Degen & Werner, 2006) sought an Existence Property for Propositional Dynamic Logic (PDL), but they questioned the practicality of their own implication introduction rule, whose side condition is non-syntactic. Furthermore, one of our results is a first-order Existence Property, which Degen cited as an open problem beyond the methods of their day. To our knowledge, only (Kamide, 2010) considers Curry-Howard or functional proof terms for a program logic. While their work is a notable precursor to ours, their logic is a weak fragment of PDL without tests, monotonicity, or unbounded iteration, while we support not only PDL but the much more powerful first-order GL. Lastly, we are preceded by Constructive Concurrent Dynamic Logic, (Wijesekera & Nerode, 2005) which gives a Kripke semantics for Concurrent Dynamic Logic (Peleg, 1987), a proper fragment of GL. Their work focuses on an epistemic interpretation of constructivity, algebraic laws, and tableaux reasoning. We differ in our use of realizability semantics and natural deduction, which were essential to developing a Curry-Howard interpretation for CGL. In summary, we are justified in claiming to be the first work which develops Curry-Howard with proof terms and an Existence Property for an *expressive* program logic, the first constructive game logic, and the only with first-order proof terms.

While constructive natural deduction calculi map most directly to functional programs, proof terms can be generated for any proof calculus, including a well-known interpretation of classical logic as continuation-passing style (Griffin, 1990). Proof terms have been developed (Fulton & Platzer, 2016) for a Hilbert calculus for dL, a dynamic logic for hybrid systems. Their work focuses on a provably correct interchange format for classical dL proofs, not constructive logics.

Intuitionistic modalities were also explored in dynamic-epistemic logic (DEL) (Frittella et al., 2016), but that work is interested primarily in proof-theoretic semantics while we employ realizability semantics to stay firmly rooted in computation. Proof-theoretic semantics for constructive, paraconsistent LTL have also been given (Kamide & Wansing, 2010). Intutionistic Kripke semantics have also been applied to multimodal System K with iteration (Celani, 2001), a weak fragment of PDL.

5.2 Syntax

We define the language of CGL, consisting of terms, games, and formulas. Terms include global, mutable *program variables* $x, y \in \mathcal{V}$ where $\mathcal{V}$ is the set of variable identifiers. We assume a signature Σ assigning a fixed base sort $\Sigma(x) = \tau$ to each x, and we assume the base sorts consist of at least the integers $\mathbb{Z}$, the constructive (Bishop) reals $\mathbb{R}$, and Booleans $\mathbb{B}$, which we use in our examples and semantics. The Bishop reals are equinumerous with classical reals, but admit only computable operations.

Definition 7 (Terms)**.** A *term* f, g is an arbitrary Type-2 (Weihrauch, 2000) computable function over the game state. That is, in the case that f and g are real-valued, they must be computable when the program variables are represented as streams of bits. This is in contrast to dGL, which is restricted to polynomial terms. We support arbitrary computable terms both because many non-polynomial terms have proven useful in practice and because for CGL we want terms to serve as witnesses to existentials. The theory of polynomial terms gives rise to existentials whose witnesses are non-polynomial, whereas the theory of computable terms has computable terms as witnesses. Thus arbitrary computable terms can serve as witnesses to existentials, while polynomial terms cannot.

We give a nonexhaustive grammar of terms, specifically those used in our examples:

$$f, g \ ::= \ \cdots \mid q \mid x \mid f + g \mid f \cdot g \mid f \ \mathsf{div} \ g \mid f \ \mathsf{mod} \ g$$

where $q \in \mathbb{Q}$ is a rational literal, x a program variable, $f + g$ a sum, $f \cdot g$ a product, and $f \ \mathsf{div} \ g$ and $f \ \mathsf{mod} \ g$ are the quotient and remainder in integer division of f by g, respectively. We leave the implementation language of f unspecified for the standard reason that different implementers might use different languages to implement realizers.

Quotient and remainder are integer operations only, and ill-typed for arguments of other types. A game in CGL is played between a constructive player named Angel and a classical player named Demon. We say that whichever player moves next is the *active* player while their opponent is *dormant*. We emphasize this terminology differs subtly from standard descriptions of classical GL: when both players are classical one identifies Angel with the active player and Demon with the dormant player.

In CGL, hybrid games follow the same syntax as in dGL, except that all formulas appearing in games are CdGL formulas. As with $\phi \vee \psi$, The Angel player must make choices computably in $\alpha \cup \beta$, in α^*, and in $x := *$. We parenthesize games with braces $\{\alpha\}$ when necessary. Sequential and nondeterministic composition both associate to the right, i.e., $\alpha \cup \beta \cup \gamma \equiv \{\alpha \cup \{\beta \cup \gamma\}\}$. This does not affect their semantics as both operators are associative, but aids in reading proof terms.

Definition 8 (CGL Formulas)**.** The set of CGL *formulas* ϕ (also ψ, ρ) is given recursively by the grammar:

$$\phi \ ::= \ \langle\alpha\rangle\phi \mid [\alpha]\phi \mid f \sim_\tau g$$

The defining constructs in CGL (and GL) are the modalities $\langle\alpha\rangle\phi$ and $[\alpha]\phi$. In CGL, in subtle contrast to GL, these modalities say that Angel can construct a proof of ϕ in every end state of game α, when starting as the active or dormant player, respectively. One could also imagine

modalities $\text{aD}(\alpha, \phi)$ and $\text{dD}(\alpha, \phi)$ which hold when an active or dormant Demon has a strategy to show ϕ as a postcondition of α. The modal formula $[\alpha]\phi$ of classical GL corresponds to what we call $\text{dD}(\alpha, \phi)$, as Angel is always active and Demon always dormant. We depart from the classical tradition because a constructive player (Angel) competing against a classical player (Demon) is fundamentally asymmetric. The constructive modalities $\text{aD}(\alpha, \phi)$ and $\text{dD}(\alpha, \phi)$ could be of independent theoretical interest, but we focus on $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ because they are the modalities useful for synthesis, whereas $\text{aD}(\alpha, \phi)$ and $\text{dD}(\alpha, \phi)$ are merely used to show that the constructive player cannot possibly win.

We assume the presence of interpreted comparison predicates $\sim_\tau \in \{\leq, <, =, \neq, >, \geq\}$ for every numeric sort τ. Comparisons are typed, i.e., they are ill-typed for arguments of different sorts.

The standard connectives of first-order constructive logic can be derived from games and comparisons. Verum ($\texttt{tt}$) is defined $1 > 0$ and falsum ($\texttt{ff}$) is $0 > 1$. Conjunction $\phi \wedge \psi$ is defined $\langle ?\phi\rangle\psi$, disjunction $\phi \vee \psi$ is defined $\langle ?\phi \cup ?\psi\rangle\top$, implication $\phi \to \psi$ is defined $[?\phi]\psi$, universal quantification $\forall x\, \phi$ is defined $[x := *]\phi$, and existential quantification $\exists x\, \phi$ is defined $\langle x := *\rangle\phi$. As usual in logic, equivalence $\phi \leftrightarrow \psi$ can also be defined $(\phi \to \psi) \wedge (\psi \to \phi)$. As usual in constructive logics, negation $\neg\phi$ is defined $\phi \to \texttt{ff}$, and inequality is defined by equivalence $f \neq g \equiv \neg(f = g)$, known as Heyting's axiom in the context of constructive reals. Likewise, we define $\theta \geq \eta \equiv \neg(\eta > \theta)$. We will use the derived constructs freely but present semantics and proof rules only for the core constructs to minimize duplication. Indeed, it will aid in understanding of the proof term language to keep the definitions above in mind, because the proof terms for many first-order programs follow those from first-order constructive logic. Note that in CGL, classical duality axioms such as $\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$ do not hold.

For convenience, we also write derived operators where the dormant player is given control of a single choice before returning control to the active player. The *dormant choice* $\alpha \cap \beta$, defined $(\alpha^d \cup \beta^d)^d$, says the dormant player chooses which branch to take, but the active player controls the subgames. *Dormant repetition* $\alpha^\times$ is defined likewise by $\{\{\alpha^d\}^*\}^d$.

We write ϕ_x^y (likewise for α and f) for the *renaming* of x for y and vice versa in formula ϕ, and write ϕ_x^f for the *substitution* of term f for program variable x in ϕ, if the substitution is admissible.

5.2.1 Example Games

We demonstrate the meaning and usage of the CGL constructs by introducing two classic games.

Nim. Nim (Fig. 5.1) is the standard introductory example of a discrete, 2-player, zero-sum, perfect-information game. The constant NIM defines the body of the game Nim, i.e., one turn for each player. The game state consists of single counter $c : \mathbb{N}$, which each player chooses ($\cup$) to reduce by 1, 2, or 3 ($c := c - k$). The counter is non-negative, so setting a negative value is a violation of the rules ($?c \geq 0$), and the game repeats until some player cannot make a move, at which point that player is declared the loser ($?c > 0$).

Proposition 3 (Dormant winning region)**.** *Suppose* $c \equiv 0 \pmod 4$, *Angel controls the loop duration, and Demon acts first. Then dormant Angel has a strategy to ensure* $c = 0$. *That is, the*

$$\textsc{Nim} = \Big\{\big\{?c > 0; \{c := c - 1 \cup c := c - 2 \cup c := c - 3\}; ?c \geq 0\big\};$$
$$\big\{?c > 0; \{c := c - 1 \cup c := c - 2 \cup c := c - 3\}; ?c \geq 0\big\}^d\Big\}$$

Figure 5.1: Example game: Nim

$$\text{CC} = x := *; ?0 \leq x \leq 1; y := 1 - x;$$
$$\{a := x; d := y \cap a := y; d := x\}$$

Figure 5.2: Example game: cake-cutting

following CGL *formula is valid (true in every state, using a common strategy):*

$$c > 0 \to c \mathbin{\mathsf{mod}} 4 = 0 \to [\textsc{Nim}^\times]\, c = 0$$

Proposition 3 says a dormant Angel has a strategy to reduce c to exactly 0 after enough iterations. This implies Angel wins the game because Demon violate the rules once $c = 0$ and no move is valid.. We now state the winning region for an active Angel player. Here the postcondition `ff`; because `ff` is not true in any state, Proposition 4 indicates that Angel forces Demon to break the rules by failing a test.

Proposition 4 (Active winning region)**.** *Suppose* $c \not\equiv 1 \pmod 4$ *initially, Angel controls the loop duration, and Angel acts first. Then Angel wins by forcing a rule violation by Demon, i.e, the following* CGL *formula is valid:*

$$c > 0 \to c \mathbin{\mathsf{mod}} 4 \neq 1 \to \langle \textsc{Nim}^* \rangle \mathit{ff}$$

Cake-cutting. Another classic 2-player game, from the study of equitable division, is the cake-cutting problem (Pauly & Parikh, 2003): The active player cuts the cake in two, then the (initially-)dormant player gets first choice of a piece. This is an optimal protocol for splitting the cake in the sense that the active player is incentivized to split the cake evenly, else the dormant player could take the larger piece. Cake-cutting is also one of the simplest examples of a game with real-valued variables. The constant CC in Fig. 5.2 defines the cake-cutting game. Here x is the relative size (from 0 to 1) of the first piece, y is the size of the second piece, a is the size of the active player's piece, and d is the size of dormant player's piece. The game is played only once. The active player picks the division of the cake, which must be a fraction $0 \leq x \leq 1$. The dormant player then picks which slice goes to whom.

An active Angel has a tight strategy to achieve a 0.5 cake share, as stated in Proposition 5.

Proposition 5 (Active winning region)**.** *The following formula is valid:*

$$\langle \text{CC} \rangle\, a \geq 0.5$$

Surprisingly, a dormant Angel does not have a computable strategy to achieve exactly 0.5 share of the cake, but rather for any $k < 0.5$ has a strategy to achieve share k, as shown in Proposition 6.

Proposition 6 (Dormant winning region)**.** *The following formula is valid:*

$$0 < k < 0.5 \rightarrow [\text{CC}]\, d > 0.5 - k$$

This reveals the subtleties of computations on reals, which we now briefly discuss.

Computability and reals. Exact comparisons on constructive reals are not decidable, e.g. formulas such as $x = 0.5 \vee x \neq 0.5$ and $x > y \vee x \leq y$ are not valid constructively. The standard comparison principle for constructive reals, which will appear as a rule in Section 5.4, is the *approximate splitting principle*: for any $\epsilon > 0$, the formula $x > y \vee x < y + \epsilon$ is valid, because approximate comparisons require only finitely-many digits, making them Type-2 computable (Weihrauch, 2000). Because disjunction is defined as Angelic choice, it is no surprise that the same challenge arises when choosing which path to take in a game, e.g. Angel is forced to fail a test in $?x < 0 \cup ?x \geq 0$ for real-valued x because unlike in classical GL Angel cannot apply the law of excluded middle. Sometimes both branches of an approximate split are satisfied at once, e.g., $y < x < x + \epsilon$. In this case Angel's strategy might suggest *multiple* correct moves, but the *collection axiom* of constructive set theory (Schwichtenberg, 2008; Bishop, 1967; Bridges & Vita, 2007) says such strategies are still constructive: intuitively, Angel's move is determined by which of $y < x$ or $x < x + \epsilon$ happened to be demonstrated first.

Because dormant Angel can choose k arbitrarily close to 0.5, we argue this is not a serious practical limitation, rather we take it as confirmation that the difference between constructive and classical games, while simple, is real and arises even in simple games. The need to use inexact comparisons is entirely reflective of the challenges that arise any time we wish to compare real (or even floating-point) numbers in software. Certainly, Angel will not bemoan the loss of ϵ slices of cake.

5.3 Semantics

We now develop the semantics of CGL. The major challenge for CGL is capturing the competition between a *constructive* Angel and *classical* Demon. We use realizability semantics (van Oosten, 2002; Lipton, 1992) to make the relationship between constructive proofs and programs particularly clear. Our realizers give Angel's computable strategy, while the opponent Demon may act classically. Because Angel is computable but Demon is classical, our semantics has the flavor both of realizability semantics and of a traditional Kripke semantics for programs.

The semantic functions are indexed by a *game state* $\omega \in \mathcal{S}$ where we write $\mathcal{S}$ for the set of all states. We additionally write $\top, \bot \in \mathcal{S}$ for the pseudo-states $\top$ and $\bot$ indicating that Angel or Demon respectively has won the game early by forcing the other to fail a test. For a fixed signature Σ, each $\omega \in \mathcal{S}$ maps each $x \in \mathcal{V}$ to a value $\omega(x) \in \mathfrak{V}(\Sigma(x))$, where $\mathfrak{V}(\tau)$ is the set of values of type τ, i.e., $\mathfrak{V}(\mathbb{Z}) = \mathbb{Z}$ and $\mathfrak{V}(\mathbb{R}) = \mathbb{R}$. We write ω_x^v for the state that agrees with ω except that x is assigned value v where $v \in \mathfrak{V}(\tau)$.

Definition 9 (Arithmetic term semantics)**.** A term f is simply a computable function of the state, so the interpretation $[\![f]\!]\omega$ of term f in state ω is $f(\omega)$.

Definition 10 (Realizers)**.** Realizers $a, b, c \in \mathcal{R}\mathbf{z}$ (where $\mathcal{R}\mathbf{z}$ is the set of all realizers) are defined coinductively:

$$a, b, c \ ::= \ \epsilon \mid \lambda\omega : \mathcal{S}.\ f \mid \Lambda x : \tau.\ a \mid \Lambda x : \phi\, \mathcal{R}\mathbf{z}.\ a \mid (a, b) \mid a\, v \mid a\, b \mid a\, \omega$$

where f is a term over the state ω. Here ϵ is the unit realizer. Lambda realizers, for example $\lambda\omega : \mathcal{S}.\ f$,consume argument ν and compute the term f with ν substituted for ω. We write (a, b) for a pair of realizers, and a_0 and a_1 for its components. Realizers for sequential compositions $\alpha; \beta$ follow the outer structure of an α realizer, but return a β realizer as a continuation on each branch. Realizers for Angelic and Demonic repetitions α^* are both coinductive streams of α-realizers. Realizer application is written (e.g.) $a\, \omega$.

5.3.1 Formula and game semantics

We write $[\![\phi]\!] \subseteq \mathcal{R}\mathbf{z} \times \mathcal{S}$ for the region (written X, Y, Z) of realizer-state pairs which realize formula ϕ. Because Angel commits to a computable strategy from the start, all nondeterminism in the game semantics is Demonic. We write $X\langle\!\langle\alpha\rangle\!\rangle : \wp(\mathcal{R}\mathbf{z} \times \mathcal{S})$ for the set of final realizer-state pairs (b, ν) which result from Angel playing first in α for some $(a, \omega) \in X$. Likewise, $X[\![\alpha]\!] : \wp(\mathcal{R}\mathbf{z} \times \mathcal{S})$ gives the realizer-state pairs (b, ν) resulting when Demon plays first. The realizer b paired with each final state ν is used to play Angel's strategy in any following game α (resp. any formula ϕ), and serves as a continuation. The definitions below implicitly assume $\bot, \top \notin X$, they extend to the case $\bot \in X$ (likewise $\top \in X$) using the equations $(X \cup \{\bot\})[\![\alpha]\!] = X[\![\alpha]\!] \cup \{\bot\}$ and $(X \cup \{\bot\})\langle\!\langle\alpha\rangle\!\rangle = X\langle\!\langle\alpha\rangle\!\rangle \cup \{\bot\}$. That is, if Demon has already won by forcing an Angel violation initially, any remaining game can be skipped with an immediate Demon victory, and vice-versa.

Definition 11 (Formula semantics)**.**

$$\begin{aligned}
(\epsilon, \omega) &\in [\![f \sim g]\!] \text{ iff } [\![f]\!]\omega \sim [\![g]\!]\omega \\
(a, \omega) &\in [\![\langle\alpha\rangle\phi]\!] \text{ iff } \{(a, \omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq ([\![\phi]\!] \cup \{\top\}) \\
(a, \omega) &\in [\![[\alpha]\phi]\!] \text{ iff } \{(a, \omega)\}[\![\alpha]\!] \subseteq ([\![\phi]\!] \cup \{\top\})
\end{aligned}$$

Definition 12 (Angel game forward semantics)**.**

$$\begin{aligned}
X\langle\!\langle ?\phi\rangle\!\rangle &= \{(a_1, \omega) \mid (a, \omega) \in X,\ (a_0, \omega) \in [\![\phi]\!]\} \cup \{\bot \mid (a, \omega) \in X,\ (a_0, \omega) \notin [\![\phi]\!]\} \\
X\langle\!\langle x := f\rangle\!\rangle &= \{(a, \omega_x^{[\![f]\!]\omega}) \mid (a, \omega) \in X\} \\
X\langle\!\langle x := *\rangle\!\rangle &= \{(a_1, \omega_x^{a_0(\omega)}) \mid (a, \omega) \in X\} \\
X\langle\!\langle \alpha; \beta\rangle\!\rangle &= (X\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\beta\rangle\!\rangle \\
X\langle\!\langle \alpha \cup \beta\rangle\!\rangle &= X_{\langle 0\rangle}\langle\!\langle\alpha\rangle\!\rangle \cup X_{\langle 1\rangle}\langle\!\langle\beta\rangle\!\rangle \\
X\langle\!\langle \alpha^*\rangle\!\rangle &= \bigcap\{Z_{\langle 0\rangle} \subseteq \mathcal{R}\mathbf{z} \times \mathcal{S} \mid X \cup (Z_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle) \subseteq Z\} \\
X\langle\!\langle \alpha^d\rangle\!\rangle &= (X[\![\alpha]\!])
\end{aligned}$$

Definition 13 (Demon game forward semantics)**.**

$$
\begin{aligned}
X[\![?\phi]\!] &= \{(a\,b,\omega) \mid (a,\omega) \in X, (b,\omega) \in [\![\phi]\!], b \in \mathcal{R}\mathbf{z}\} \cup \{\top \mid (a,\omega) \in X, \text{ but no } (b,\omega) \in [\![\phi]\!]\} \\
X[\![x := f]\!] &= \{(a, \omega_x^{[\![f]\!]\omega}) \mid (a,\omega) \in X\} \\
X[\![x := *]\!] &= \{(a\,r, \omega_x^r) \mid r \in \mathfrak{V}(\Sigma(x))\} \\
X[\![\alpha;\beta]\!] &= (X[\![\alpha]\!])[\![\beta]\!] \\
X[\![\alpha \cup \beta]\!] &= X_{[0]}[\![\alpha]\!] \cup X_{[1]}[\![\beta]\!] \\
X[\![\alpha^*]\!] &= \bigcap\{Z_{[0]} \subseteq \mathcal{R}\mathbf{z} \times \mathcal{S} \mid X \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\} \\
X[\![\alpha^d]\!] &= (X\langle\!\langle\alpha\rangle\!\rangle)
\end{aligned}
$$

Comparisons $f \sim g$ defer to the term semantics, so the interesting cases are the program modalities. Both $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ ask whether Angel wins α by following the given strategy, and differ only in whether Demon vs. Angel is the active player, thus in both cases *every* Demonic choice must satisfy Angel's goal, and early Demon wins are counted as Angel losses. The program cases exploit the *Angelic* $Z_{\langle 0\rangle}, Z_{\langle 1\rangle}$ and *Demonic* projections $Z_{[0]}, Z_{[1]}$, which represent binary decisions made by constructive Angel and classical Demon, respectively. The Angelic projections, which are defined $Z_{\langle 0\rangle} = \{(a_1,\omega) \mid a_0(\omega) = 0\}$ and $Z_{\langle 1\rangle} = \{(a_1,\omega) \mid a_0(\omega) = 1\}$, filter by which branch Angel chooses with $a_0(\omega) \in \mathbb{B}$, then project the remaining strategy a_1. The Demonic projections, which are defined $Z_{[0]} \equiv \{(a_0,\omega) \mid (a,\omega) \in Z\}$ and $Z_{[1]} \equiv \{(a_1,\omega)\}$, contain all the same states as Z, but left-project and right-project the realizer, respectively, to notify Angel which branch was taken. We now discuss the Angel cases, then the Demon cases.

Angelic tests $?\phi$ end in the current state ω with remaining realizer a_1 if Angel can realize ϕ with a_0. Angelic deterministic assignments consume no realizer and simply update the state, then end. Angelic nondeterministic assignments $x := *$ ask the realizer a_0 to compute a new value for x from the current state. Angelic compositions $\alpha;\beta$ first play α, then β from the resulting state using the resulting continuation.

Angelic choice games $\alpha \cup \beta$ use the Angelic projections to decide which branch is taken according to a_0. The realizer a_1 may be reused between α and β, since a_1 could just invoke a_0 if it must decide which branch has been taken. This definition of Angelic choice (corres. constructive disjunction) captures the reality that realizers in CGL, in contrast with most constructive logics, are entitled to observe a game state, but they must do so in computable fashion.

In Angelic repetition α^*, Angel has fixed a computable strategy in advance, which is invoked after each repetition to decide whether the loop continues, based on the current state.

5.3.2 Duality semantics

To play the dual game α^d, the active and dormant players switch roles, then play α. In *classical* GL, this characterization of duality is interchangeable with the definition of α^d as the game that Angel wins exactly when it is impossible for Angel to lose. The characterizations are *not* interchangeable in CGL because the Determinacy Axiom (all games have winners) of GL is not valid in CGL:

Remark 1 (Indeterminacy). The classically equivalent determinacy axiom schemata $\neg\langle\alpha\rangle\neg\phi \to [\alpha]\phi$ as well as $\langle\alpha\rangle\neg\phi \vee [\alpha]\phi$ of classical GL are not valid in CGL.

Remark 2 (Classical duality). In classical GL, Angelic dual games are characterized by the axiom schema $\langle\alpha^d\rangle\phi \leftrightarrow \neg\langle\alpha\rangle\neg\phi$, which is not valid in in CGL.

Remark 3 (Classical interdefinability). In classical GL, the equivalence $\langle\alpha^d\rangle \leftrightarrow [\alpha]\phi$ is provable from the axiom schemata $\langle\alpha^d\rangle\phi \leftrightarrow \neg\langle\alpha\rangle\neg\phi$ and $[\alpha]\phi \leftrightarrow \neg\langle\alpha\rangle\neg\phi$ (consistency & determinacy).

CGL takes $\langle\alpha^d\rangle\phi \leftrightarrow [\alpha]\phi$ and $[\alpha^d]\phi \leftrightarrow \langle\alpha\rangle\phi$ as primary, since the two axiom schemata of 3 do not hold. Determinacy is interesting in its own right, and CGL is determined in the following weak sense due to a classical existence dichotomy of strategies:

Proposition 7 (Weak determinacy)**.** *If Angel is active, every game α with goal ϕ is determined because either there exists a strategy a such that $(\{a\} \times \mathcal{S})\langle\langle\alpha\rangle\rangle \subseteq [\![\phi]\!] \cup \{\top\}$ (Angel wins for every Demon decision) or for all a, $(\{a\} \times \mathcal{S})\langle\langle\alpha\rangle\rangle \cap ([\![\phi]\!] \cup \{\top\})^{\complement} \neq \emptyset$ (Demon wins for some decision). Symmetrically if Angel is dormant, then every game α with goal ϕ is determined because either there exists a strategy a such that $(\{a\} \times \mathcal{S})[\![\alpha]\!] \subseteq [\![\phi]\!] \cup \{\top\}$ or for all a, $(\{a\} \times \mathcal{S})[\![\alpha]\!] \cap ([\![\phi]\!] \cup \{\top\})^{\complement} \neq \emptyset$.*

Note that determinacy could be stated as a double-negated axiom if we extend CGL with modalities for the winning regions of active and dormant Demons.

5.3.3 Demonic semantics

Demon wins a Demonic test by presenting a realizer b as evidence that the precondition holds. If they cannot present a realizer (i.e., because none exists), then Angel wins by default. Else Angel's higher-order realizer a consumes the evidence of the pre-condition, i.e., Angelic strategies are entitled to depend (computably) on *how* Demon demonstrated the precondition. Angel can check that Demon passed the test by executing b. In demonic nondeterministic assignment $x := *$, Demon chooses to set x to *any* value of its type. In the Demonic choice game $\alpha \cup \beta$, Demon chooses classically between α and β.

Not every syntactically-valid ϕ-strategy is sensible. In $Z\langle\langle\alpha^*\rangle\rangle$, realizers must be "effectively well-founded," meaning they lead the loop to terminate. For each a there must be a computable termination metric $\mathcal{M}$ such that $a(\omega) = \mathcal{M} = 0$ on $\omega \in X$ and $\sup_{(c,\omega)\in Y} \mathcal{M}(\omega) < \sup(c,\omega) \in Y\langle\langle\alpha\rangle\rangle$ for all Y and suitable c, where $<$ is the ordering on the metric. If a ϕ-realizer a satisfies this constraint (or if it does not even apply) then we say a is *suitable* for ϕ.

We can now formally define validity of formulas and intuitionistic sequents. A formula ϕ is *valid* iff there exists a suitable realizer a uniformly for all states, i.e., such that $\{a\} \times \mathcal{S} \subseteq [\![\phi]\!]$. An intuitionistic sequent $\Gamma \vdash \phi$ is *valid* if the formula $\bigwedge\Gamma \to \phi$ is valid, that is if there exists suitable a such that for all suitable b and ω for which $(b,\omega) \in [\![\bigwedge\Gamma]\!]$ holds it is the case that $(b\,a,\omega) \in [\![\phi]\!]$, where $\bigwedge\Gamma$ is the conjunction of all assumptions in Γ. The realizer b is a product of realizers for all components of $[\![\bigwedge\Gamma]\!]$, a conjunction which can be taken under any fixed ordering on Γ, e.g., lexicographic ordering on variable names. We also write $a \models (\bigwedge\Gamma \to \phi)$ to say a is the witness of $\Gamma \vdash \phi$.

Proof Calculus For practical use, CGL requires a syntactic proof method. For this purpose, CGL has a constructive natural deduction calculus equipped with proof terms. The use of proof terms makes it easy to study the computational content of the proofs. The main proof judgement is $\Gamma \vdash M : \phi$, which says term M is a proof of ϕ in the context Γ. The assumptions in the context

are named, so that they may be mentioned in proof terms. We give an example proof rule:

$$(\langle\cup\rangle\text{E}) \quad \frac{\Gamma \vdash A : \langle\alpha \cup \beta\rangle\phi \quad \Gamma, \ell : \langle\alpha\rangle\phi \vdash B : \psi \quad \Gamma, r : \langle\beta\rangle\phi \vdash C : \psi}{\Gamma \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \psi}$$

This says that a case analysis $\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ eliminates the choice $\langle\alpha \cup \beta\rangle\phi$ (which must be proven by A) to prove any conclusion ψ, so long as B and C prove ψ assuming the left or right branch were taken, respectively.

For practical use, a sequent calculus could be implemented "as a library" on top of the natural deduction calculus.

Operational Semantics Proof terms can be understood as pure functional programs in the sense that they reduce to normal forms. We write M normal to say M is normal, or $M \mapsto M'$ to say M reduces to M' in one step. The normal forms are as follows:

Lemma 8 (Normal Form Characterization)**.** *If $\Gamma \vdash M : \phi$ and M is normal, then the final step of proof M is either:*

- *A case-redex that inspects the state, e.g. case on (split $[\theta \sim \theta + \epsilon]\ M$), or*
- *An introduction rule for the outermost connective of ϕ. In the case that ϕ is a program modality, it is specifically an introduction rule for the outermost program construct.*

5.4 Theory Results

We prove standard theoretical results. Progress and preservation show that CGL is sound when interpreted as a *type system*. All non-normal proofs reduce to proofs of the same sequent. This is a fundamental step in establishing a Curry-Howard correspondence for games.

Lemma 9 (Progress)**.** *If $\Gamma \vdash M : \phi$, then either M is normal or $M \mapsto N$ for some N.*

Lemma 10 (Preservation)**.** *If $\Gamma \vdash M : \phi$ and $M \mapsto^* N$, then $\Gamma \vdash N : \phi$*

The following properties justify the claim that CGL is constructive by showing that every provable existential (resp. disjunctive, modal) formula has a computable witness. Not only are these properties frequently used as a test of constructivity, but they are directly tied to our motivation of using constructivity for synthesis: because provable games have computable strategies, we can also hope to synthesize those strategies.

Lemma 11 (Existence Property)**.** *If $\Gamma \vdash M : (\exists x : \tau\ \phi)$ then there exists a term f and realizer b such that for all $(a, \omega) \in [\![\bigwedge \Gamma]\!]$, we have $(b\,a, \omega_x^{f(\omega)}) \in [\![\phi]\!]$.*

Lemma 12 (Disjunction Property)**.** *When $\Gamma \vdash M : \phi \vee \psi$ there exists realizer b and computable f, s.t. for every ω and a such that $(a, \omega) \in [\![\bigwedge \Gamma]\!]$, either $f(\omega) = 0$ and $(b_0, \omega) \in [\![\phi]\!]$, else $f(\omega) = 1$ and $(b_1, \omega) \in [\![\psi]\!]$.*

Lemma 13 (Active Strategy Property)**.** *If $\Gamma \vdash M : \langle\alpha\rangle\phi$, then there exists a realizer b such that for all ω and realizers a such that $(a, \omega) \in [\![\bigwedge \Gamma]\!]$, then $\{(b\,a, \omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$.*

Lemma 14 (Dormant Strategy Property)**.** *If $\Gamma \vdash M : [\alpha]\phi$, then there exists a realizer b such that for all ω and realizers a such that $(a, \omega) \in [\![\bigwedge \Gamma]\!]$, then $\{(b\,a, \omega)\}[\![\alpha]\!] \subseteq [\![\phi]\!] \cup \{\top\}$.*

While these properties have short proofs, the deeper point is that they fall out naturally from design choices which were far-reaching, such as our use of realizability.

Arithmetic Proofs CdGL accepts constructive analysis as its foundation instead of the classical analysis used in dGL. One challenge is ensuring that such proofs are no more complex than strictly necessary, yet still sound by constructive principles.

In lieu of the law of excluded middle, the case analysis principle in constructive analysis is the Approximate Splitting Principle:

$$\frac{\Gamma \vdash M : \epsilon > 0}{\Gamma \vdash \text{split } [\theta \sim \eta]\ M : (\theta > \eta \vee \theta < \eta + \epsilon)}$$

That is, we can freely compare two terms, so long as we allow an overlap of some $\epsilon > 0$. We allow arbitrary proofs of $\epsilon > 0$ for generality, but in practice ϵ will usually be a rational literal, for which the premiss is trivially decidable.

The constructive theory of first-order arithmetic differs from the classical theory precisely because exact comparisons are not decidable. Whether the full theory is decidable is a major open question (Lombardi, 2019) beyond the scope of this thesis. However, we propose to identify simple decidable fragments.

5.5 Soundness Theorem

The soundness theorem says that CGL is sound when interpreted as a *proof calculus*. We first list standard renaming and substitution lemmas which are required for the soundness theorem.

Lemma 15 (Renaming)**.** *If* $\Gamma \vdash M : \phi$ *then* $[y/x]\Gamma \vdash [y/x]M : [y/x]\phi$.

Lemma 16 (Context substitution)**.** *If* $\Gamma, x : \psi \vdash M : \phi$ *and* $\Gamma \vdash N : \psi$ *then* $\Gamma \vdash [N/x]M : \phi$.

Lemma 17 (Variable substitution)**.** *If* $\Gamma \vdash M : \phi$ *and* $[\theta/x]\phi$ *is admissible then* $[\theta/x]\Gamma \vdash [\theta/x]M : [\theta/x]\phi$.

Given these lemmas, we can prove soundness: all provable formulas are true according to the denotational semantics which we have taken as our ground truth. This is a sine qua non for any proof calculus and justifies our use of the calculus to prove CGL theorems.

Theorem 18 (Soundness of Proof Calculus)**.** *If* $\cdot \vdash M : \phi$ *then* ϕ *is realizability-valid.*

The proof follows by structural induction on the derivation. Proofs can be assumed to be normal.

5.6 Status

The work presented in this chapter is under submission to POPL'20 (Bohrer & Platzer, 2020). All theorems listed here have been proven. Several other theorems are also of interest. In particular, we wish to explore a Gödel-Gentzen translation and compare the closure ordinals for repetition games in CGL with those for GL. We are also interested in exploring completeness of the CGL calculus with respect to the realizability semantics. The technical details of such a completeness proof would differ significantly from completeness proofs for dGL (Platzer, 2017b) in that we would now be treating a realizability semantics, not a winning-region semantics. However, these studies can be undertaken equally well in the context of CdGL, so we plan to do so in the proposed work (Chapter 6).

Chapter 6

Proposed Work: CdGL

In the proposed work, we extend CGL from Chapter 5 to support *hybrid* games so that we can model and verify adversarial CPS's. We call the resulting extension CdGL. Extending CGL to CdGL is important for the same reason that it is important to allow ODEs in hybrid systems models rather than just discrete approximations of the plant. The Logician seeks a model that is clearly in close correspondence with reality, and this correspondence is clearer for continuous physics than for discrete approximations. The Logic-User also wants reasoning about the model to be straightforward, and experience suggests direct verification of a continuous model is conceptually simpler than verification of a discrete approximation.

We propose developing the theory of CdGL, then evaluating CdGL by using the 2D planar robot of Section 4.2 as a case study.

6.1 CdGL Theory

We propose to generalize the results of Chapter 5 for CdGL and also provide several results which were not given in past publications. The first step is to extend the language of games α, β to *hybrid* games by adding differential equations:

$$\alpha, \beta ::= \cdots \mid x' = f \,\&\, Q$$

As in dGL, the differential equation $x' = f \,\&\, Q$ says that x evolves continuously according to $x' = f$ for a duration controlled by the active player, which is chosen subject to the constraint Q throughout.

One of the major hurdles in this proposed work is to give an appropriate semantics and proof calculus for constructive differential equations. However, we assert that a constructive semantics for differential equations can be given because Picard iteration gives a constructive definition of the solution of a (locally Lipschitz) differential equation, and constructive formalization of Picard iteration (Krebbers & Spitters, 2011) show that the solution of a (locally Lipschitz) differential equation is a computable function. We briefly review the dGL rules for differential equation games and suggest why a constructive interpretation should be feasible. Fig. 6.1 gives an axiomatization for differential equations in classical dL or dGL. Axiom $\langle' \rangle$ is typically derived from the others and says that we can verify a differential equation by verifying its solution. Axiom

$$(\langle' \rangle) \quad \langle x' = f \,\&\, \phi \rangle \leftrightarrow \exists t{\geq}0 \, \langle x := y(t) \rangle \phi$$

$$\text{(DW)} \quad [x' = f \,\&\, Q](Q \rightarrow \phi) \leftrightarrow [x' = f \,\&\, Q]\phi$$

$$\text{(DC)} \quad ([x' = f \,\&\, \psi]\phi \leftrightarrow [x' = f \,\&\, \psi \wedge \rho]\phi) \leftarrow [x' = f \,\&\, \psi]\rho$$

$$\text{(DI)} \quad [x' = f \,\&\, \psi]\phi \leftarrow \phi \wedge [x' = f \,\&\, \psi]\phi'$$

$$\text{(DG)} \quad \exists y \, [x' = f, y' = a(x) * y + b \,\&\, \psi]\phi \leftrightarrow [x' = f \,\&\, \psi]\phi$$

$$\text{(DS)} \quad (\forall t \, (\forall 0{\leq}s{\leq}t \, [x := x + cs]\psi) \rightarrow [x := x + ct]\phi) \leftrightarrow [x' = c \,\&\, \psi]\phi$$

$$\text{(DV)} \quad \langle x' = h \rangle f \geq g \leftarrow \exists \varepsilon > 0 \, ([x' = h](f \leq g \rightarrow f' \geq g' + \varepsilon))$$

Figure 6.1: Differential equation rules

DS, which is not derived, is the special case of $\langle' \rangle$ for constant ODEs. To show DS constructively valid, it should suffice to show constructively that $x(t) = x_0 + ct$ is the unique solution of $x' = c$. This is a special case of the Picard-Lindelöf theorem, which is true constructively (Krebbers & Spitters, 2011, 2011). Picard-Lindelöf is also the basis of validity for DG (LICS, 2018), which says it is sound to extend a system of differential equations with a new continuous variable, so long as the new equation is linear. The linearity constraint ensures the existence interval (i.e., the time before an infinite asymptote is reached, if any) of the differential equation is maintained, for soundness. Axiom DG is mainly used in its forward direction to help prove postconditions which are invariant but not inductive, and in its converse direction to implement $\langle' \rangle$. For differential equations whose solutions are infeasibly complicated, the "bread-and-butter" proofs are done with DC, DI, and DW. DI says that a postcondition ϕ is invariant if it holds initially and if its differential (e.g., $(f \geq g)' \equiv (f)' \geq (g)'$) is invariant. DC allows one DI to serve as a "lemma" to another by adding it to the domain constraint, and DW allows closing the proof by proving the postcondition from a sufficiently strengthened domain constraint. Validity of DI follows from the Mean-Value Theorem (a special case of Rolle's Theorem), which has been proven constructively (Krebbers & Spitters, 2011, 2011). The classical soundness proofs of DC and DW are direct, and we expect the same for an appropriately-chosen semantics. The axioms of Fig. 6.1 will need adaptation to the constructive setting. Even leaving constructivity aside, CdGL also supports a stronger term language than dGL, and we will need to treat continuity and differentiability carefully as was done in Section 3.2.

6.2 Example System

As a case study for CdGL, we propose reproducing the 2D planar robot of Section 4.2 as a hybrid game, and providing a constructive proof that the robot can reach its waypoints while maintaining its speed limits. In order to assess the feasibility of this task early, without access to an implementation of CdGL, we propose performing the proof in classical dGL in KeYmaera X and inspecting by hand that the proof uses only constructive reasoning. We give the hybrid game

model here. The plant is identical to that of Section 4.2: the robot is fixed at the origin facing toward the +y axis, while the waypoint moves relative to it:

$$\begin{aligned}\mathsf{plant} \equiv \{&\mathsf{x}' = -\mathsf{v}\,\mathsf{k}\,\mathsf{y},\ \mathsf{y}' = \mathsf{v}\,(\mathsf{k}\,\mathsf{x} - 1),\ \mathsf{v}' = \mathsf{a},\ \mathsf{t}' = 1 \\ &\&\ \mathsf{t} \leq \mathsf{T}\ \wedge\ \mathsf{v} \geq 0\}\end{aligned}$$

The conditions Ann and Feas, which say the waypoint and robot are on the specified annulus, and that the waypoint is achievable, respectively, are also as before:

$$\begin{aligned}\mathsf{Ann}\ &\equiv\ |\mathsf{k}|\varepsilon \leq 1 \wedge \left|\frac{\mathsf{k}\ (\mathsf{x}^2 + \mathsf{y}^2 - \varepsilon^2)}{2} - \mathsf{x}\right| < \varepsilon \\ \mathsf{Feas}\ &\equiv \mathsf{Ann} \wedge \mathsf{y}{>}0 \wedge 0{\leq}\mathsf{vl}{<}\mathsf{vh} \wedge \mathsf{AT}{\leq}\mathsf{vh} - \mathsf{vl} \wedge \mathsf{BT}{\leq}\mathsf{vh} - \mathsf{vl}\end{aligned}$$

The main change is in the controller. In the systems setting, the controller needed to express a condition Go which specified the allowable accelerations in each case. In the games setting, the acceleration is existentially quantified in the model, and choices of acceleration are merely part of the proof:

$$\mathsf{ctrl}\ \equiv\ ((\mathsf{x},\mathsf{y}) := *;\ [\mathsf{vl},\mathsf{vh}] := *;\ \mathsf{k} := *;\ ?\mathsf{Feas})^d;\ \mathsf{a} := *$$

Recall that in Section 4.2 we proved *safety* (the robot never leaves the path) and *liveness* (the robot eventually reaches its goal). We wish to show the same high-level properties here, but in the context of games we express them as Angel's ability to win a game against Demon. The following winnability theorem says the robot can always control its way to the goal region within its speed limits (liveness):

Theorem 19 (Angel Robot Winnability)**.** *The following CdGL formula is valid:*

$$\begin{aligned}&\mathsf{A}{>}0 \wedge \mathsf{B}{>}0 \wedge \mathsf{T}{>}0 \wedge \varepsilon{>}0 \wedge J \rightarrow \\ &\langle(\mathit{ctrl};(\mathit{plant})^d)^*\rangle\big(\|(\mathsf{x},\mathsf{y})\| \leq \varepsilon \wedge \mathsf{v} \in [\mathsf{vl},\mathsf{vh}]\big)\end{aligned}$$

where J is the loop invariant as defined in Section 4.2.

Winnability subsumes liveness because it says there exist control decisions which reach the goal, given arbitrary Demonic control of the plant duration. We propose to generalize Theorem 19 slightly, yielding a single theorem which subsumes both Theorem 2 (liveness) and Theorem 1 (safety) in one go.

Theorem 20 (Generalized Angel Robot Winnability)**.** *The following CdGL formula is valid:*

$$\begin{aligned}&\mathsf{A}{>}0 \wedge \mathsf{B}{>}0 \wedge \mathsf{T}{>}0 \wedge \varepsilon{>}0 \wedge J \rightarrow \\ &\langle(\mathit{ctrl};(\mathit{plant})^d;?J)^*\rangle\big(\|(\mathsf{x},\mathsf{y})\| \leq \varepsilon \wedge \mathsf{v} \in [\mathsf{vl},\mathsf{vh}]\big)\end{aligned}$$

where J is the loop invariant as defined in Section 4.2.

That is, after Demon chooses the duration of the plant, Angel must prove that the loop invariant J still holds. Recall that the invariant J includes as conjuncts the facts that the robot is on

the annulus and has sufficient distance remaining to achieve its speed limit. To win the generalized game, Angel must be able to prove that it is on track to the waypoint (both in position and velocity) at *whatever* point Demon chooses, thus at every one of the uncountably many points in time. This is exactly what safety ought to constitute. Because we simply added a test to the previous model, whenever Angel wins the modified game they have still achieved liveness. It is simply harder now, as they must never violate safety in the process of achieving liveness.

The example demonstrates several advantages of hybrid games proving generally over hybrid systems. Not only is the model shorter, waypoint-following exhibits turn-taking behavior which is challenging to characterize in a hybrid system. The Angel player must follow an arbitrary waypoint, i.e., one chosen by Demon. In order to make victory for Angel even possible, the responsibility is Demon's to choose a waypoint that is feasible. Expressing this turn-taking in Theorem 2 is cumbersome, while in Theorem 20, it falls naturally out of the fact that acceleration is controlled by Angel while all else is up to Demon. Lastly, we have the advantage that safety and liveness need not be proved separately, but can be combined as a single winnability theorem. This clearly saves us the fuss of repeating the dynamics in each theorem statement, but there is a conceptual advantage as well: in the systems setting, we must prove *all* options are safe and that *one* is live. In the games setting, we can simply prove that *one* strategy is safe-and-live, so for states our strategy will never enter, we need not even consider safety.

The winnability approach ought to apply to any *reach-avoid* problem for a control-plant loop, i.e., any theorem of form:

$$\mathsf{pre} \rightarrow \langle (\mathsf{ctrl};\ (\mathsf{plant})^d;\ ?\mathsf{safe})^* \rangle \mathsf{live}$$

Within the hybrid systems literature, and certainly within the dL literature, most systems can be expressed as control-plant loops, and the most important properties are reach-avoid properties or special cases thereof. Thus we are justified to claim that many verification tasks would benefit from the simplicity provided by modeling and proving with games.

6.2.1 Relating Classical and Constructive Games

After validating CdGL on the above case study, we propose to study the relationship between constructive and classical games, specifically CdGL vs. dGL. Typically, constructive logics support a "Gödel-Gentzen" (GG) translation, which allows the classical counterpart to be embedded in a constructive logic. We propose to develop a GG translation for CdGL for the dual reasons that a GG translation increases our confidence that we have the "right" definition of CdGL, and that it is of independent interest to know that every classical dGL property can be proved using CdGL technology. We let T stand for the translation.

Theorem 21 (GG embedding)**.** *If (and only if) ϕ is provable in* dGL*, then $\cdot \vdash M : T(\phi)$ in CdGL for some M.*

This result is useful because it allows us to embed classical statements in a constructive proof, much as if T were a "modality". Because dGL is (relatively) complete, this also yields a limited completeness theorem in CdGL.

The converse question asks whether every CdGL theorem is true in dGL, i.e.:

Theorem 22 (Converse embedding)**.** *If $\Gamma \vdash M : \phi$ in CdGL then ϕ is provable in* dGL*.*

Such an embedding theorem, when true, typically has a simple proof by induction on M. Nonetheless, this result is worth knowing because it complements the constructivity tests of Chapter 5 by saying CdGL is the constructive counterpart *of* dGL *specifically*. For any pair of classical and constructive logics, we generally expect that constructive truth implies classical truth: if not, we would have to be careful when applying our intuitions about dGL to CdGL. Lastly, embedding suggests a way to connect CdGL implementation efforts to the existing dGL implementation: A CdGL proof can be expressed as a dGL proof which happens to use only constructive rules.

6.2.2 Relaxing (Constructive) Games to (Constructive) Systems

We provide another new theoretical result in order to solidify the connection between CdGL and plain dL, which is used for the input of the VeriPhy tool. Theorem 22 already says every CdGL theorem is a theorem of dGL, and on a close reading the theorem holds true if CdGL and dGL are each restricted to systems. To complete the connection between CdGL and dL, it suffices to show that every proven game in CdGL can be refined to a system by inlining the strategy from the proof. This has long been a folk theorem for dGL, which we now make rigorous. In doing so, we show how we could automatically process CdGL models and proofs to generate systems models and proofs which are adequate as an input to the existing VeriPhy tool. This allows us to connect CdGL to VeriPhy without reimplementing VeriPhy from the ground up for constructive games.

We introduce some notation here to state the relaxations rigorously. We write $\alpha \leq \beta$ to say that α *refines* β, i.e., the behaviors of α are a subset of the behaviors of β. The refinement logic dRL (Loos & Platzer, 2016) developed refinement for dL, which can be extended to games by defining $(\alpha^d \leq \beta) \equiv (\alpha \geq \beta)$. That is, the effect of the dual operator is to flip the direction of the inclusion just as it flips modalities. Our goal is to show that there are (computable) inlining functions $\mathsf{inline}_{[\alpha]}(d)$ and $\mathsf{inline}_{\langle\alpha\rangle}(a)$ which inline Demon or Angel strategies d, a (respectively) into a game α. We demand that the inlining functions produce refinements, which guarantees by monotonicity that the original theorems hold also of the refined system. That is, safety proofs, when inlined only ever restrict the set of behaviors, which preserves any safety property. When liveness proofs are inlined, they expand the set of behaviors, which preserves any liveness property.

Theorem 23 (Angelic relaxation)**.** *If* $a \models \langle\alpha\rangle\phi$ *then* $\mathsf{inline}_{\langle\alpha\rangle}(a) \leq \alpha$ *and* $b \models \langle\mathsf{inline}_{\langle\alpha\rangle}(a)\rangle\phi$ *for some* b*.*

Theorem 24 (Demonic relaxation)**.** *If* $d \models [\alpha]\phi$ *then* $\mathsf{inline}_{[\alpha]}(d) \geq \alpha$ *and* $D \models [\mathsf{inline}_{[\alpha]}(d)]\phi$ *for some* D*.*

The Demonic relaxation of a safe system can then be fed to VeriPhy as we already know it is safe too.

Chapter 7

Ongoing Work: Kaisar, Structured Proofs for Dynamic Logics

At this point we have identified the logic CdGL as the backbone of our verification toolchain. However, CdGL will need to be implemented for practical use. Experience shows that the existing Bellerophon (Fulton et al., 2017) proof scripting language for dGL poses unnecessary barriers for the Logic-User, e.g.:

- Bellerophon's reliance on search-based and especially position-based formula selection leads to unmaintainable proof scripts.
- The main automated proof methods in Bellerophon are not easily *traceable*, i.e., they result in proof goals whose correspondence with the source model is not clear from inspection. In practice this makes it difficult to comprehend proof states, let alone a finished proof during maintenance. The best alternative presently is to write manual proof steps for each connective in the model, but this results in excessive duplication between the model and proof.
- Bellerophon takes an ad-hoc approach to proving the first-order verification conditions which arise at the leaves of proofs. The complexity of first-order logic over reals demands that these goals can not always be discharged automatically in practice, but the Logic-User needs a clear toolbox with which to address these subgoals.

Moreover, there is surprisingly little extant research on principled designs for proof languages, so we take this opportunity to investigate the design of proof languages for dynamic logics (incl. Hoare logics) generally. We will then specialize the design to CdGL and implement it, which will provide a tool to check CdGL proofs.

The first inspiration for the Kaisar language is the Isar (Wenzel, 2007) proof language from Isabelle/HOL (Nipkow et al., 2002), with its notion of *structured proof*. The defining feature of Isar is its use of block-structured statements for reasoning steps such as making an assumption, cutting in a fact, or proving a subgoal. When facts are introduced to the context, they may be assigned names, by which they are later referenced.

While the simple act of naming facts seems trivial from the perspective of a programming language, it is both deep from the perspective of dynamic logic proofs and impactful in the context of the dL family. Any well-behaved programming (or proof) language requires well-

behaved scoping rules. Preferably, the scope of a variable should always be determined lexically by the construct that introduces it. This is actually non-trivial for program logics, because a fact about the state can be invalidated by executing a program, e.g., the equation $x = 0$ will cease to hold upon assigning $x := 1$. This is one reason that Bellerophon does not have named contexts, only references by position and value, which are far more fragile and/or verbose. Even traditional proof languages, such as Coq scripts, struggle with this: while Coq script does have named facts, those facts are scoped dynamically. Our first contribution is that Kaisar has lexical scope, unlike Coq script and Bellerophon. We discuss other proof languages in full detail in Section 7.3.

The second main contribution solves an issue identified during the design and implementation of the first iteration of Kaisar (Bohrer & Platzer, 2019). In practice, most Logic-Users are not satisfied to maintain both the source code/model for a verified system *and* a separate proof artifact. While dynamic logic proofs need not follow the structure of a program in perfect lockstep, many program constructs do have a single canonical introduction rule, meaning many proof steps can be uniquely determined from the program structure. If we completely separate the proof from the model, we end up duplicating these "canonical" steps between the model and proof, which roughly doubles the amount of proof boilerplate written by the Logic-User.

Prior approaches which do not have this overhead are not structured, i.e., they do not have lexical scope, but rather work by verification-condition generation (see Section 7.3). Thus the contribution of Kaisar is not just to achieve lexical scope but to do so while avoiding proof-vs.-model duplication.

The basic trick to avoiding proof duplication is to write proofs *as* programs which have been annotated with enough insights to complete the proof. We call the combination of a program and annotations a proven-program. A proven-program contains only the minimal possible proof information beyond a program, e.g., invariant annotations for loops and witness annotations for quantifier instantiation. Kaisar is by no means the first language to prove programs by adding annotations to them, for example Dafny does the same.

We argue that Kaisar offers something unique, however. Prior annotation-based languages like Dafny (Leino, 2010) rely on incomplete proof automation based on black-box use of SMT solvers. That automation is known to be fragile: changes as simple as rewriting $x + y < z$ to $x < z - y$ can break a proof. Kaisar is interesting because it fuses the ease-of-use of annotation-based proving with the full power of structured first-order proof, a la Isar. In doing so, we dodge the scalability and transparency issues that come with past annotation-based approaches.

Fusing Dafny-like ease-of-use with Isar-like generality is technically challenging. The core challenge behind Kaisar is to manage the proof context in the face of program dynamics. From a birds-eye perspective, the context-management challenges in Kaisar are much like those faced when we developed the operational semantics of CGL proofs in Chapter 5. We will also show that (in the systems case) this context can be understood in the context of differential-dynamic hybrid logic (dHL) (Bohrer & Platzer, 2018) developed by the author, where program states can be assigned names and then referenced via *nominals*. We equip Kaisar with several constructs for nominal reasoning.

7.1 Proposed Design

We start by defining the language of proven-programs α:

$$\alpha ::= x{:=}\theta \mid x{:=}* \mid ?x : \phi \mid \alpha; \beta \mid \alpha \cup \beta \mid (\alpha)^*@\{B\} \mid \{B\} \mid L{:}\ \alpha$$

where B is a structured proof block and L is a label. In the loop case, the annotated block $\{B\}$ proves that the loop invariant follows from the current context. The invariant need not be explicitly annotated because (it is a metatheorem that) the conclusion of a proof can be inferred from the proof automatically. The proof of the inductive step is embedded in the annotated loop body α.

There is a general-purpose generalization construct $\{B\}$ which embeds a structured proof B in a program: the proof B starts with the program's current proof context, and imports the conclusion of B back into the broader proof. While we may choose to employ other constructs in structured proofs B, I give some of the natural choices here:

$$B ::= \mathsf{show}\ x : \phi\ U \mid \mathsf{have}\ x : \phi\ U\ B \mid \mathsf{note}\ x : \phi\ F\ B$$

where U is an unstructured proof according to the native language of the host prover and F is a forward-style explicit proof term. Depending on the host prover, unstructured proofs could be written in Bellerophon, Coq script, and Isabelle apply-script. Forward style proof terms correspond loosely to the keywords `of` and `OF` in Isar, or to a minimal proof term language with modens ponens in logic or type theory. Every proof ends with ($\mathsf{show}\ x : \phi\ U$) where the proof's conclusion ϕ is proven by U and given name x. Proof ($\mathsf{have}\ x : \phi\ U\ B$) cuts in fact $x : \phi$, proves it with U, then continues proving B. Proof ($\mathsf{note}\ x : \phi\ F\ B$) also cuts in $x : \phi$, with the distinction that the cut formula ϕ is proven with a *forward* proof term F.

The program construct $?(x : \phi)$ can also function like the Isar "assume" keyword for introducing assumptions since the implication $\phi \rightarrow \psi$ and box test $[?\phi]\psi$ are equivalent by axiom $\langle ? \rangle$. More generally, we can implement what Martin Davis called "Obvious Logical Inferences (Davis, 1981) ," that is proof-checking can identify formulas up to trivial syntactic identities, which Isar and Mizar do.

The first nominal construct is labeling $L : \alpha$, which introduces a label L which refers to the current state, then behaves as program α. The label L can then be used in both the term language and formula language. The term language contains (at least) literals, variables, nominals, and functions:

$$\theta ::= c \mid x \mid @_L(\theta) \mid f(\theta)$$

The nominal term $@_L(\theta)$ refers to the old value of θ in state L. One of the major technical contributions in the Kaisar implementation will be an elaboration pass that implements $@_L(\theta)$ using ghost variables to remember the state L. This can be done following the translation in (Bohrer & Platzer, 2018) and means that the Logic-User does not have to write out hybrid-logic proofs manually to handle nominals.

Formulas contain at least comparisons, propositional connectives, and nominals:

$$\phi ::= \theta \sim \eta \mid \phi \wedge \psi \mid \neg\phi \mid \forall x\, \phi \mid @_L(\phi) \mid \geq(L)$$

where $\sim$ stands for any comparison operator $\sim \in \{<, \leq, =, \neq, \geq, >\}$. The nominal $@_L(\phi)$ gives the truth value of ϕ as of label L, and can be elaborated into ϕ where are terms θ become the nominal $@_L(\theta)$, which is it turn elaborated into ghost variables and/or substitutions. The proposition $\geq(L)$ if the current program counter is "after" L, and is implemented as a binary ghost variable. This is a natural variation on the proposition L of hybrid logic which holds true exactly *at* state L, and is useful for example in Owicki-Gries (S. Owicki & Gries, 1976) reasoning, where it is known (Lamport, 1977) to be the *only* kind of ghosting necessary for completeness.

7.2 Proposed Theory

Soundness is a sine qua non for any proof language. Beyond soundness, a key property that makes the Kaisar design work is the fact that in the proof-checking judgement, the formula proved has the output modality, i.e., we can syntactically compute the model proved as a function of its proof. Rather than write the checking judgement explicitly and proving its mode, we give a strongest-postcondition semantics for Kaisar proofs, then the desired modality follows directly from the fact that strongest-postcondition is a function. We write $\mathsf{sp}(S, \Gamma)$ for the strongest postcondition of proven-program S starting in context Γ. We also write Γ_α for the *context* which results from checking $\mathsf{sp}(S, \Gamma)$. That is, if $\vec{x} = \mathrm{BV}(\alpha)$ and $\vec{y}$ is fresh with $|\vec{x}| = |\vec{y}|$,, and $\vec{z} : \vec{\psi}$ are the auxiliary facts proved during S, then $\Gamma_\alpha = (\Gamma \frac{\vec{y}}{\vec{x}}), \vec{z} : \vec{\psi}$ where $\Gamma \frac{\vec{y}}{\vec{x}}$ is the fresh renaming of each x_i to corresponding y_i.

$$
\begin{aligned}
\mathsf{sp}(\{\mathsf{show}\ x : \phi\ U\}, \Gamma) &= \phi && \text{if U proves } \phi \\
\mathsf{sp}(\{\mathsf{have}\ x : \phi\ U\ S\}, \Gamma) &= \mathsf{sp}(S, (\Gamma, x : \phi)) && \text{if U proves } \phi \\
\mathsf{sp}(\alpha; \beta, \Gamma) &= \mathsf{sp}(\beta, (\Gamma_\alpha, \mathsf{sp}(\alpha, \Gamma))) \\
\mathsf{sp}(\alpha \cup \beta, \Gamma) &= \mathsf{sp}(\alpha, \Gamma) \vee \mathsf{sp}(\beta, \Gamma) \\
\mathsf{sp}(?(x : \phi), \Gamma) &= \phi \\
\mathsf{sp}(x := \theta, \Gamma) &= (x = \theta) \\
\mathsf{sp}((\alpha)^*@\{B\}, \Gamma) &= \mathsf{sp}(\{B\}, G) && \text{if } \mathsf{sp}(\alpha, \Gamma_{\alpha\{B\}}) = \mathsf{sp}(\{B\}, \Gamma) \\
\mathsf{sp}(L\colon \alpha, \Gamma) &= \mathsf{sp}(\alpha, (\Gamma, L))
\end{aligned}
$$

That is, "show" must prove its declared goal, and "have" must prove the lemma before proceeding to S. In $\alpha; \beta$, the conclusion is given by β, but the proof of β has access to all results proven in α. In $\alpha \cup \beta$, the postcondition is the disjunction of those for α and β because either branch might be taken. Tests assume ϕ, which they also conclude by default. Every assignment to x exports a fact which by convention is also named x.

If we wish for a different postcondition than the inferred one, we insert a block-style proof. For example, the following is a proof that both 1 and 2 are positive (with $\cup$ written as U):

```
     (x:=1 {show (x > 0) using x by auto})
U   (x:= 2 {show (x > 0) using x by auto})
```

the result of checking this proven program is that the program $x{:=}1 \cup x{:=}2$ has $x > 0$ as a postcondition. We could write the block proof once instead, after the program:

```
(x:=1 U  x:=2)
```

```
{show (x > 0) using x by auto}
```

The loop case is perhaps most surprising. The invariant is "chosen" by checking the proof B which is the proof that "the invariant holds initially". Then proof-checking succeeds if the body proves that the invariant is preserved, and the postcondition is the invariant. As mentioned previously, an explicit generalization step may be added if a postcondition other than the invariant is desired. As another toy example, here is a proof that incrementing 2 repeatedly gives a positive number:

```
x:=2;
{x:=x+1}*@{show gr1:"x>1" using x by auto}
{show "x>0" using gr1 by auto}
```

Labels do not modify the postcondition, they just add themselves to the context.

We wish to provide a formal guarantee regarding the modularity provided by Kaisar's lexical scope. We take inspiration from the *adaptation completeness* (Kleymann, 1999) theorem for VDM, which says when a Hoare triple $\{P\}\alpha\{Q\}$ holds of *every* α, then a proof of any $\{P\}\alpha\{Q\}$ can be adapted to a proof of $\{P\}\beta\{Q\}$ for any β. That is, $\{P\}\beta\{Q\}$ will have a proof whenever any $\{P\}\alpha\{Q\}$ has a proof.

However, we are more concerned with adapting a proof when only one part of the program changes. We propose to define a notion of *modular proofs* which are amenable to such adaptations and a *modularity completeness* saying that every provable property has a modular proof. Informally, a modular proof is one where a change to a subprogram requires changes only to the subproof for that program.

Definition 14 (Modular proof)**.** Let C, D stand for contexts $\mathsf{Prog} \to \mathsf{Prog}$ and $\mathsf{Prog} \to \mathsf{Formula}$, respectively. Consider any α, β, Γ such that $C(\beta) = \alpha$. Then we say the pair (A, F) is a *modular proof* of $\langle C(\beta)\rangle\phi$ in context Γ if:

1. $\Gamma \vdash A : D(\beta)$
2. F is a natural transformation from programs to proofs where for every program γ we have $\Gamma \vdash F(\gamma) : (D(\gamma) \to \langle C(\gamma)\rangle\phi)$.

While this definition is subtle, the basic idea is to decompose the proven-program α as a subprogram β in a context C. A *modular proof* uses an *interface* $D(\beta)$ which specifies all the properties of β that are needed in the rest of the proof. Then the modular proof consists of a proof A that β satisfies interface D and a family of proofs F that *any* program γ which satisfies interface D can be substituted for β in context C and maintain the postcondition ϕ.

Given this definition of modular proof, a first cut at a modular completeness theorem might show that for every $\Gamma \vdash M : \langle\alpha\rangle\phi$ there is a modular proof (A, F) However, there always exist *trivial modular proofs*: if we define $\beta = \alpha, C(\cdot) = \cdot$, and $D(\cdot) = \langle\gamma\rangle\phi$ then the pair $(M, (\_ \mapsto \mathrm{id}))$ is a modular proof, where id is the identity proof of $P \to P$. From a high level, a modular proof is only interesting if the interface D is somehow simpler than the theorem $\langle\alpha\rangle\phi$ and if proofs can be adapted with respect to non-trivial contexts C.

A *strong modular completeness* theorem would say that a proof can be modularized for *every* context C which program α matches. This would mean that no matter which part of the program we choose to modify during maintenance, only the corresponding proofs would be affected. However, it is possible that strong modular completeness does not hold: first-order verification conditions can depend intimately on, e.g. the values of preceeding assignments, and Dynamic

Logic-style proofs for sequential composition $\alpha; \beta$ are less modular than the restrictive Hoare rule for compositions. For this reason, we propose to identify the strongest modular completeness theorem that can be proven, thus answering the question of "When exactly do small program changes lead to small proof changes"? We also propose to prove completeness w.r.t. the CdGL calculus, so that Kaisar can inherit the proposed CdGL completeness results.

This chapter has a significant implementation component, through which we will also implement CdGL proof checking. The Kaisar parser will produce an abstract syntax tree which is then elaborated to the intermediate representation (IR), a natural-deduction calculus based on the calculus from Chapter 6. The initial implementation of Kaisar will implement proof-checking directly on the IR. Because every constructive theorem holds classically, a second possible implementation approach is to generate classical dGL proofs from the IR, then check that the proof uses only constructive steps. A benefit of this approach is that it can connect to the verified KeYmaera X core and take advantage of our trust in that core. In Chapter 8, we will use the same IR as the input to synthesis.

7.3 Related Work

Structured proofs were introduced over 40 years ago in Mizar, which has seen continued use in the years since (Bancerek et al., 2015; Wenzel & Wiedijk, 2002). Isar (Wenzel, 2006, 1999, 2007) is one of the most mature structured proof languages today, and has been successfully applied to large-scale verification tasks (Nipkow, 2002; Klein et al., 2010; Lochbihler, 2007). In Mizar, the "unstructured" fragment consists of simple one-line commands so that the structured proof steps are the majority, whereas Isar allows arbitrarily-complex unstructured proofs using Isabelle's "apply-script" language. One of the major questions remaining in the Kaisar design is whether to employ a simplistic unstructured language as in Mizar, or whether to incorporate a full-powered unstructured language such as Bellerophon. Other examples include DECLARE (Syme, 1997) for HOL, TLAPS for TLA^+ (Cousineau et al., 2012; Lamport, 1992), SSReflect (Gonthier & Mahboubi, 2010) and declarative proofs (Corbineau, 2007) in Coq. Kaisar is inspired immediately by Isar, and reuses its **note**, **show**, and **have** keywords. Natural-language structured proof paradigms have also been developed (Lamport, 2012, 1995). While their main goal is to ensure that paper proofs are done correctly, they can also inspire the design of formal languages. Apt's *proof outlines* (K. Apt et al., 2010) are another paradigm that provides limited proof structuring. Proof outlines consist of a sequence of proof states, and are more compact than full proof-trees, but we hope for Kaisar to be even more compact.

"Mizar modes" have been used to implement structured proofs for provers without extending the core. Mizar modes are available Cambridge HOL (Harrison, 1996), Isabelle (Kaliszyk et al., 2016), and HOL Light (Wiedijk, 2001). The "mode" approach was also used to write the Iris Proof Mode (IPM) in Coq (Krebbers et al., 2017), which implements the Iris concurrent separation logic in Coq's $\mathcal{L}_{\mathtt{tac}}$ language. The "Mizar mode" approach works well in provers with small LCF-style cores, where the Mizar mode can employ the core as a "library". Coq and KeYmaera X both (Barras & Werner, 1997; Bohrer et al., 2017) have small LCF-style cores.

Two classes of languages closely related to structured proofs are tactics languages (which implement reusable automation) and unstructured proof languages (which implement concrete

proofs). Automation can often be written in the prover's implementation language: OCaml in Coq, ML in Isabelle, or Scala in KeYmaera X. Domain-specific languages for tactics include untyped $\mathcal{L}_{\texttt{tac}}$ (Delahaye, 2000), reflective Rtac (Malecha & Bengtson, 2015), and dependently typed Mtac (Ziliani et al., 2013) in Coq, Eisbach (Matichuk et al., 2016) in Isabelle, and VeriML (Stampoulis & Shao, 2010, 2012) in a standalone proof-checker. The KeY prover (Ahrendt et al., 2016) features a soundness-critical "tactlet" (Habermalz, 2000) language with limited expressivity, which can be understood as allowing user-specified inference rules. Examples of unstructured languages are the Coq (*Coq Proof Assistant*, 1989) script language and the Isabelle (Nipkow et al., 2002) apply-script language. KeYmaera X features a language named Bellerophon (Fulton et al., 2017) for unstructured proofs and tactics. Bellerophon consists of regular expression-style tactic combinators (sequential composition, repetition, etc.) and a standard tactics library featuring, e.g. sequent calculus rules and general-purpose automation. Bellerophon's strength is in tactics that compose the significant automation provided in its library. Its weakness is in performing large-scale concrete proofs. For example, assumptions in Bellerophon are unnamed and referred to by their index or by search, which can become unreadable or brittle at scale. It lacks both the nominals unique to Kaisar and structuring: the Bellerophon combinators in common use are a strict subset of the combinators in apply-script. Over time, the Bellerophon implementation has gained new features such as auxiliary tactic, function, and program definitions, which have partially offset scalability challenges. However, these features have not been studied in detail, let alone published. Thus, one of the auxiliary goals in designing Kaisar is to revisit features from recent releases of Bellerophon, develop a formal description when appropriate, and propose any improvements.

Kaisar is distinguished from prior languages by its metatheory, especially for nominals. While Isar and Coq's declarative language have formally defined semantics, we know of only one interactive proof language besides Kaisar with significant metatheoretic results: VeriML (Stampoulis & Shao, 2010, 2012), which is implemented as a standalone proof checker. We share with VeriML the goal of solving practical interactive proof problems via principled proof languages with metatheoretic guarantees. We differ in that VeriML's theory addresses staged computation with an eye toward fast proof-checking while ours addresses stateful context management with an eye toward understandable, maintainable proofs.

The theory of Kaisar explains state change with nominals, a la hybrid differential dynamic logic dHL (Bohrer & Platzer, 2018), first proposed as dL_H (Platzer, 2007). In dHL, nominal formulas enable stating and proving theorems about named states. Our goal differs: we apply named states to simplify proofs of theorems, but nominals don't appear in the theorems.

While we provide the most general treatment of historical state and its theory via nominals, there is a rich tradition of historical reference in program proofs. It has long been known that historical reference, implementable with ghost state, is an essential component of proof in many domains (K. R. Apt et al., 1979; S. Owicki & Gries, 1976; S. S. Owicki, 1975; K. Apt et al., 2010; Clint, 1973). It has been known equally long (Clarke, 1980) that manual ghost arguments can make proofs clumsy. A number of verification tools (Ahrendt et al., 2016; Fulton et al., 2015; Leino, 2010, 2008; Barnett et al., 2005; Leino et al., 2009; Leavens et al., 1999; Kleymann, 1999) offer ad-hoc historical reference constructs without theoretical justification which can reference only the initial program state. We generalize them and provide the first formal treatment of historical reference. In the hybrid systems setting specifically, *differential* ghosts

are also essential (LICS, 2018) for reasoning about differential equations.

We are broadly interested in exploiting other principled design ideas from the proof language literature. Beyond structured proof, another notable principled feature is the use of pattern-matching for formula selection, which has been investigated by Traut (Traut & Noschinski, 2014) and Gonthier (Gonthier & Tassi, 2012). Pattern-matching was also investigated in the first proposal for Kaisar (Bohrer & Platzer, 2019). Based on this experience, we plan to leave pattern matching out of the core calculus for simplicity, but experiment with pattern-matching in the implementation. When proving programs in Isar, it is possible to assign custom case labels (Noschinski, 2015) to produce more readable proofs, which gives a similar flavor to proofs in the original Kaisar design (Bohrer & Platzer, 2019).

Kaisar's structuring paradigm can be contrasted with proof by verification-condition generation (King, 1971): Many approaches to program verification work by generating a first-order logic formula (the verification condition) which, if true, implies correctness of the program. The VC is then fed to either an automatic prover or an interactive prover. A major limitation of VCG's is that in many cases it is undecidable whether the VC is valid, so any automation used is incomplete. Even when VC's fall within a decidable logic, they are often infeasible in practice. This is the case, for example, with real first-order VC's in dL, which are decidable (Tarski, 1951, reprinting 1998), but often infeasible (Davenport & Heintz, 1988; Weispfenning, 1997). Approaches which rely solely on automated backends then fail to scale to large proofs, and approaches that use interactive provers face their own scalability issues because there is a lack of *traceability*: the resulting verification condition appears cryptic and disconnected from the program text. The first-order goals the user proves in Kaisar are the same as they would prove in VCG with interactive proof, but the contribution of Kaisar is that the first-order proofs are written in context, and the present place in the program can easily be traced by reading the proof.

Dafny (Leino, 2010) is an example of a prominent system which uses VCG with SMT-solving as an incomplete backend. Dafny allows using assertions and equational reasoning to assist automation, but even then incompleteness makes debugging a failed proof attempt challenging. Ivy (Padon et al., 2016) is one system which sidesteps the opacity of backend solvers by using a decidable specification logic, so that its backend solver is complete. However, this approach does not apply to CdGL, which inherits undecidability from dGL.

Chapter 8

Proposed Work: ProofPlex, Proof-Directed Game Synthesis

In this chapter we propose a tool for the synthesis of controllers and monitors from (proven-safe and proven-live) computational hybrid games, which we name ProofPlex. We motivate ProofPlex by first considering the different artifacts one might wish to synthesize:

1. A *controller monitor*, which detects whether an untrusted external controller is compliant with the assumptions of a controller model
2. A *plant monitor*, which detects whether the environment is compliant with the physical assumptions of the plant model.
3. A *controller*, which actually picks a guaranteed-safe-and-live control decision to carry out
4. A *plant simulator*, which picks an adversarial behavior for the plant and evaluates the effect of the plant.

All of these are desirable under different circumstances: Synthesizing a controller is appealing because unlike an untrusted controller, it is guaranteed to make a safe decision so the feedback need not be invoked (i.e, it is proven to satisfy the controller monitor). Yet the downside of a synthesized controller is that it sacrifices flexibility: if our synthesized monitor guarantees correctness but was not optimized for operational efficiency, fuel efficiency etc., we might one day decide we prefer a controller which meets those criteria instead. In this case the flexibility given by the monitoring approach is preferred. On the plant side, plant monitors are perfect for deployment in production systems or during field tests: they allow us to assess whether assumptions on the environment are met in practice. But in early stages of development, especially if there is not a purpose-built simulator for our system already, we would much prefer to synthesize code which can actually (and faithfully) simulate the plant for us, which is helpful in designing the corresponding controller. The first two are monitors which merely detect noncompliance with a model, whereas the latter two actively control the system. Notably, the first two are supported by the existing ModelPlex (Mitsch & Platzer, 2016) tool while the latter two are not. Moreover, we obviously wish that we could synthesize these artifacts for arbitrarily complicated systems, whereas existing implementations of ModelPlex have limitations, for example that the controller must contain no differential equations and loops. Until recent advances concurrent with (Bohrer et al., 2018) ModelPlex could not exploit information from proofs, which prevented synthesizing

nontrivial plant monitors beyond nilpotent differential equations.

It is by looking at the advances that led to recent gains that we can then, however, discover the changes that achieve our lofty goal of synthesizing all four artifacts and doing so for arbitrary systems (and games). A key insight behind a recent advance in the ModelPlex tool (which generalized plants to polynomial ODEs so long as the plant is proven safe using only chains of inductive invariants) is that even in synthesizing a monitor, *the contents of a proof are essential to synthesis*. Specifically, that work extracted differential invariants from a safety proof in order to synthesize monitors In retrospect, this key insight is no longer surprising: synthesis should be at least as difficult as verification, and because verification of hybrid systems/games is undecidable, a general-case synthesis procedure should not exist. Yet even though verification is undecidable, proof *checking* is utterly decidable, as the proof contains all the difficult decisions made in determining the truth of a liveness statement. Therefore, the ultimate question behind this chapter is, "can the proof content be reused for synthesis"?, to which the thesis statement answers "yes". A proof in CdGL is nothing but a winning strategy for a hybrid game, which contains both all the control decisions made by the Angelic player and the assumptions monitored of the Demonic player. In short, the following could be called the punchline of the thesis:

> A box game proof witnesses monitorability and a diamond game proof witnesses controllability, thus a CdGL proof witnesses synthesizability

Whenever the players alternate turns in a game, the proof alternates modalities. Whenever we prove a $\langle\alpha\rangle P$ property, we do so by witnessing the decisions (computations) Angel performs to achieve P. In proving a $[\alpha]P$ property, we enumerate rules which Angel must uphold until control passes to Demon, and in doing so we enumerate the contents of a monitor.

This chapter aims to show that this content's computational interpretation can and should be used to automate the synthesis of all four artifacts, not just in special cases but in general cases, including:

- Model-predictive controllers using differential equations
- Differential equations with not only differential invariants but also differential ghosts, which together suffice (LICS, 2018) to prove any polynomial invariant of a polynomial ODE
- Controllers containing loops
- Sophisticated plants beyond the single ODE systems supported by prior work. For example, an *event-triggered model*, where the controller can respond immediately to some physical event, usually has a nondeterministic choice of ODEs as its plant. Sometimes systems which pass through multiple stages, such as in spaceflight, also find it useful to employ a separate ODE for each stage.

The implementation of ProofPlex contains an implementation of the relaxation theorems of the previous section, which first characterize which classical proofs also work as constructive proofs, then compute a hybrid system relaxation of the game. Once a relaxation for the controller model (and controller proof) in a hybrid game has been computed, verification can be made end-to-end by hooking it in to the existing VeriPhy pipeline. What remains is to supply

the untrusted controller. Because ProofPlex, unlike ModelPlex, also synthesizes controllers by exploiting Angelic proofs, it will produce an "untrusted" controller which in fact is trustworthy, because it is simply an implementation of the Angelic dynamics of the control proof in CdGL. By Chapter 6 the semantics of the synthesized controller refines the denotational semantics of both the hybrid game controller and of its systems relaxation, meaning it satisfies the controller monitor.

Chapter 9

Conclusion

The proposed thesis provides a comprehensive approach to end-to-end verification: i) Cyber-physical systems are modeled as hybrid games ii) The hybrid game is verified in a structured, high-level, constructive proof language iii) Correct controllers and monitors are synthesized from the proof and correctly compiled to machine code iv) The controller is linked with trusted hardware drivers for execution v) Formal foundations of CdGL are developed. In the special case of classical dL, the soundness proof has been formalized in a theorem-prover. The proposed thesis validates this approach by applying it to a classic ground robotics control problem.

The proposed thesis intentionally casts a wide net in defining end-to-end verification. Each researcher may prioritize different objectives within the broader mission of bringing CPS verification to practice. For this reason, we approach end-to-end verification dialectically by comparing the perspectives of a Logician, Logic-User, and Engineer. We show that the proposed approach exceeds the state-of-the-art in meeting the competing needs of our three characters. Our chain of formal proof artifacts provides a strong foundation for the Logician, our Kaisar proof language helps the Logic-User, and our synthesis tools help the Engineer. While end-to-end verification is the overarching theme which connects the proposed thesis chapters, many chapters are also of broader interest.

9.1 Timeline

The proposal will be October 25, 2019 at 9:00am Pittsburgh time. If all goes according to plan, I intend to defend in early Fall 2020. I intend to finish the CdGL theory first, since the theory will inform the implementation. The implementation work will be informed by VeriPhy and the Kaisar prototype, but I still expect this to take the most time.

Many of the most appropriate conferences have early 2020 deadlines. Since it is unlikely that all components will be finished by then, and because timelines are only estimates, I have listed several conference options for each work. I intend to defend approximately August 2020.

Total: 12 months

- 3 months (Sep-Nov) Extend CGL theory to CdGL theory (Submit to IJCAR around February 2020, or FOSSACS in Oct 2019 if done early)

- 3 months (Dec-Feb) Implement Kaisar with CdGL proof checking (Submit to ITP or FM late March 2020)
- 2 months: (Mar-Apr) Develop ProofPlex implementation (Submit to RV, May 2020, or if finished early, CAV in February 2020)
- 1 month: (May) Generalize ground robotics case study to CdGL
- 3 months: (June-August) Write thesis document, defend

References

Abramsky, S., Jagadeesan, R., & Malacaria, P. (2000). Full abstraction for PCF. *Inf. Comput.*, *163*(2), 409–470. Retrieved from https://doi.org/10.1006/inco.2000.2930 doi: 10.1006/inco.2000.2930

Aczel, P., & Gambino, N. (2006). The generalised type-theoretic interpretation of constructive set theory. *J. Symb. Log.*, *71*(1), 67–103. Retrieved from https://doi.org/10.2178/jsl/1140641163 doi: 10.2178/jsl/1140641163

Affeldt, R., & Cohen, C. (2017). Formal foundations of 3D geometry to model robot manipulators. In Y. Bertot & V. Vafeiadis (Eds.), *Proceedings of the 6th ACM SIGPLAN conference on certified programs and proofs, CPP 2017, Paris, France, January 16-17, 2017* (pp. 30–42). ACM. Retrieved from https://doi.org/10.1145/3018610.3018629 doi: 10.1145/3018610.3018629

Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., & Ulbrich, M. (Eds.). (2016). *Deductive software verification - the key Book - from theory to practice* (Vol. 10001). Springer. Retrieved from https://doi.org/10.1007/978-3-319-49812-6 doi: 10.1007/978-3-319-49812-6

Alechina, N., Mendler, M., de Paiva, V., & Ritter, E. (2001). Categorical and Kripke semantics for constructive S4 modal logic. In L. Fribourg (Ed.), *Computer science logic, 15th international workshop, CSL 2001* (Vol. 2142, pp. 292–307). Springer. Retrieved from https://doi.org/10.1007/3-540-44802-0_21 doi: 10.1007/3-540-44802-0_21

Allen, S. F., Bickford, M., Constable, R. L., Eaton, R., Kreitz, C., Lorigo, L., & Moran, E. (2006). Innovations in computational type theory using Nuprl. *J. Applied Logic*, *4*(4), 428-469. (http://www.nuprl.org/)

Althoff, M., & Dolan, J. M. (2014). Online verification of automated road vehicles using reachability analysis. *IEEE Trans. Robotics*, *30*(4), 903–918. Retrieved from https://doi.org/10.1109/TRO.2014.2312453

Alur, R., Henzinger, T. A., & Kupferman, O. (2002). Alternating-time temporal logic. *J. ACM*, *49*(5), 672–713. Retrieved from https://doi.org/10.1145/585265.585270 doi: 10.1145/585265.585270

Alur, R., Henzinger, T. A., Lafferriere, G., & Pappas, G. J. (2000, July). Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, *88*(7), 971-984. doi: 10.1109/5.871304

Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O. S., ... Weaver, M. (2017). CertiCoq: A verified compiler for Coq. In *The third international workshop on Coq for programming languages (CoqPL).*

Anand, A., & Knepper, R. A. (2015). ROSCoq: Robots powered by constructive reals. In C. Urban & X. Zhang (Eds.), *ITP* (Vol. 9236, pp. 34–50). Springer. Retrieved from https://doi.org/10.1007/978-3-319-22102-1_3 doi: 10.1007/978-3-319-22102-1_3

Anand, A., & Rahli, V. (2014). Towards a formally verified proof assistant. In G. Klein & R. Gamboa (Eds.), *ITP* (Vol. 8558, pp. 27–44). Springer. doi: 10.1007/978-3-319-08970-6_3

Andronick, J., & Felty, A. P. (Eds.). (2018). *Proceedings of the 7th ACM SIGPLAN international conference on certified programs and proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. ACM. Retrieved from http://dl.acm.org/citation.cfm?id=3176245

Apt, K., De Boer, F. S., & Olderog, E.-R. (2010). *Verification of sequential and concurrent programs*. Springer Science & Business Media.

Apt, K. R., Bergstra, J. A., & Meertens, L. G. L. T. (1979). Recursive assertions are not enough - or are they? *Theor. Comput. Sci.*, *8*, 73–87. Retrieved from https://doi.org/10.1016/0304-3975(79)90058-6 doi: 10.1016/0304-3975(79)90058-6

Artëmov, S. N., Davoren, J. M., & Nerode, A. (1997). Topological semantics for hybrid systems. In S. I. Adian & A. Nerode (Eds.), *Logical foundations of computer science, 4th international symposium, LFCS, Yaroslavl, Russia, July 6-12, 1997, proceedings* (Vol. 1234, pp. 1–8). Springer. Retrieved from https://doi.org/10.1007/3-540-63045-7_1 doi: 10.1007/3-540-63045-7_1

Avigad, J., & Chlipala, A. (Eds.). (2016). *CPP2016*. ACM. Retrieved from http://dl.acm.org/citation.cfm?id=2854065

Baier, C., & Tinelli, C. (Eds.). (2015). *Tools and algorithms for the construction and analysis of systems - 21st international conference, TACAS 2015, held as part of the European joint conferences on theory and practice of software, ETAPS 2015, London, UK, April 11-18, 2015. proceedings* (Vol. 9035). Springer. Retrieved from https://doi.org/10.1007/978-3-662-46681-0 doi: 10.1007/978-3-662-46681-0

Bancerek, G., Bylinski, C., Grabowski, A., Kornilowicz, A., Matuszewski, R., Naumowicz, A., ... Urban, J. (2015). Mizar: State-of-the-art and beyond. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, & V. Sorge (Eds.), *CICM* (Vol. 9150, p. 261-279). Springer. Retrieved from http://dblp.uni-trier.de/db/conf/mkm/cicm2015.html#BancerekBGKMNPU15 doi: 10.1007/978-3-319-20615-8

Barnett, M., Leino, K. R. M., & Schulte, W. (2005). The Spec# Programming System: An Overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, & T. Muntean (Eds.), *Construction and analysis of safe, secure, and interoperable smart devices: International workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, revised selected papers* (pp. 49–69). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-540-30569-9_3 doi: 10.1007/978-3-540-30569-9_3

Barras, B. (2010). Sets in Coq, Coq in sets. *J. Formalized Reasoning*, *3*(1), 29–48. doi: 10.6092/issn.1972-5787/1695

Barras, B., & Werner, B. (1997). *Coq in Coq* (Tech. Rep.). INRIA Rocquencourt.

Bauer, A. (2005). Realizability as connection between constructive and computable mathe-

matics. In T. Grubba, P. Hertling, H. Tsuiki, & K. Weihrauch (Eds.), *CCA 2005 - second international conference on computability and complexity in analysis, August 25-29, 2005, Kyoto, Japan* (Vol. 326-7/2005, pp. 378–379). FernUniversität Hagen, Germany.

Becker, H., Zyuzin, N., Monat, R., Darulova, E., Myreen, M. O., & Fox, A. C. J. (2018). A verified certificate checker for finite-precision error bounds in Coq and HOL4. In N. Bjørner & A. Gurfinkel (Eds.), *2018 formal methods in computer aided design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018* (pp. 1–10). IEEE. Retrieved from https://doi.org/10.23919/FMCAD.2018.8603019 doi: 10.23919/FMCAD.2018.8603019

Beyer, D., & Huisman, M. (Eds.). (2018). *Tools and algorithms for the construction and analysis of systems - 24th international conference, TACAS 2018, held as part of the european joint conferences on theory and practice of software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, proceedings, part I* (Vol. 10805). Springer. Retrieved from https://doi.org/10.1007/978-3-319-89960-2 doi: 10.1007/978-3-319-89960-2

Bhatia, A., Kavraki, L. E., & Vardi, M. Y. (2010). Motion planning with hybrid dynamics and temporal goals. In *CDC* (pp. 1108–1115). IEEE. Retrieved from https://doi.org/10.1109/CDC.2010.5717440

Bishop, E. (1967). Foundations of constructive analysis.

Bohrer, B., Fernández, M., & Platzer, A. (2019). dL$_\iota$: Definite descriptions in differential dynamic logic. In P. Fontaine (Ed.), *Automated deduction - CADE 27 - 27th international conference on automated deduction, Natal, Brazil, August 27-30, 2019, proceedings* (Vol. 11716, pp. 94–110). Springer. Retrieved from https://doi.org/10.1007/978-3-030-29436-6_6 doi: 10.1007/978-3-030-29436-6_6

Bohrer, B., & Platzer, A. (2018). A hybrid, dynamic logic for hybrid-dynamic information flow. In A. Dawar & E. Grädel (Eds.), *LICS* (p. 115-124). New York: ACM. doi: 10.1145/3209108.3209151

Bohrer, B., & Platzer, A. (2019). Toward structured proofs for dynamic logics. *CoRR*, *abs/1908.05535*. Retrieved from http://arxiv.org/abs/1908.05535

Bohrer, B., & Platzer, A. (2020). Constructive game logic. In *POPL*. ACM. (Under review)

Bohrer, B., Rahli, V., Vukotic, I., Völp, M., & Platzer, A. (2017). Formally verified differential dynamic logic. In Y. Bertot & V. Vafeiadis (Eds.), *CPP* (p. 208-221). ACM. doi: 10.1145/3018610.3018616

Bohrer, B., Tan, Y. K., Mitsch, S., Myreen, M. O., & Platzer, A. (2018). VeriPhy: Verified controller executables from verified cyber-physical system models. In D. Grossman (Ed.), *PLDI* (p. 617-630). ACM. doi: 10.1145/3192366.3192406

Bohrer, B., Tan, Y. K., Mitsch, S., Sogokon, A., & Platzer, A. (2019). A formal safety net for waypoint following in ground robots. *IEEE Robotics and Automation Letters*, *4*(3), 2910-2917.

Boldo, S., Filliâtre, J., & Melquiond, G. (2009). Combining Coq and Gappa for certifying floating-point programs. In J. Carette, L. Dixon, C. S. Coen, & S. M. Watt (Eds.), *MKM, held as part of CICM* (Vol. 5625, pp. 59–74). Springer. Retrieved from https://doi.org/10.1007/978-3-642-02614-0_10 doi: 10.1007/978-3-642-02614-0_10

Boldo, S., Jourdan, J., Leroy, X., & Melquiond, G. (2013). A formally-verified C compiler supporting floating-point arithmetic. In A. Nannarelli, P. Seidel, & P. T. P. Tang (Eds.), *Arith*

(pp. 107–115). IEEE Comp. Soc. Retrieved from https://doi.org/10.1109/ARITH.2013.30 doi: 10.1109/ARITH.2013.30

Boldo, S., & Melquiond, G. (2011). Flocq: A unified library for proving floating-point algorithms in Coq. In E. Antelo, D. Hough, & P. Ienne (Eds.), *ARITH* (pp. 243–252). IEEE Comp. Soc. Retrieved from https://doi.org/10.1109/ARITH.2011.40 doi: 10.1109/ARITH.2011.40

Bouissou, O., Goubault, E., Putot, S., Tekkal, K., & Védrine, F. (2009). HybridFluctuat: A static analyzer of numerical programs within a continuous environment. In A. Bouajjani & O. Maler (Eds.), (Vol. 5643, pp. 620–626). Springer. Retrieved from https://doi.org/10.1007/978-3-642-02658-4_46 doi: 10.1007/978-3-642-02658-4_46

Bourke, T., Brun, L., Dagand, P., Leroy, X., Pouzet, M., & Rieg, L. (2017). A formally verified compiler for Lustre. In A. Cohen & M. T. Vechev (Eds.), *PLDI* (pp. 586–601). ACM. Retrieved from http://doi.acm.org/10.1145/3062341.3062358 doi: 10.1145/3062341.3062358

Bridges, D. S., & Vita, L. S. (2007). *Techniques of constructive analysis*. Springer Science & Business Media.

Celani, S. A. (2001). A fragment of intuitionistic dynamic logic. *Fundam. Inform.*, *46*(3), 187–197. Retrieved from http://content.iospress.com/articles/fundamenta-informaticae/fi46-3-01

Chan, M., Ricketts, D., Lerner, S., & Malecha, G. (2016). Formal verification of stability properties of cyber-physical systems. In *Proc. CoqPL*.

Chatterjee, K., Henzinger, T. A., & Piterman, N. (2007). Strategy logic. In L. Caires & V. T. Vasconcelos (Eds.), *CONCUR 2007 - concurrency theory, 18th international conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, proceedings* (Vol. 4703, pp. 59–73). Springer. Retrieved from https://doi.org/10.1007/978-3-540-74407-8_5 doi: 10.1007/978-3-540-74407-8_5

Chen, F., & Rosu, G. (2007). MOP: an efficient and generic runtime verification framework. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, & G. L. S. Jr. (Eds.), *Proceedings of the 22nd annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada* (pp. 569–588). ACM. Retrieved from https://doi.org/10.1145/1297027.1297069 doi: 10.1145/1297027.1297069

Chen, X., Ábrahám, E., & Sankaranarayanan, S. (2013). Flow*: An analyzer for non-linear hybrid systems. In N. Sharygina & H. Veith (Eds.), *CAV* (Vol. 8044, pp. 258–263). Springer. Retrieved from https://doi.org/10.1007/978-3-642-39799-8_18 doi: 10.1007/978-3-642-39799-8_18

Chen, X., Schupp, S., Makhlouf, I. B., Ábrahám, E., Frehse, G., & Kowalewski, S. (2015). A benchmark suite for hybrid systems reachability analysis. In *NFM* (Vol. 9058).

Church, A. (1956). *Introduction to mathematical logic*. Princeton University Press.

Cimatti, A., Griggio, A., Mover, S., & Tonetta, S. (2014). Verifying LTL properties of hybrid systems with k-liveness. In A. Biere & R. Bloem (Eds.), *Computer aided verification - 26th international conference, CAV 2014, held as part of the Vienna summer of logic, VSL 2014, Vienna, Austria, July 18-22, 2014. proceedings* (Vol. 8559, pp. 424–440). Springer. Retrieved from https://doi.org/10.1007/978-3-319-08867-9_28 doi: 10

.1007/978-3-319-08867-9_28

Cimatti, A., Griggio, A., Mover, S., & Tonetta, S. (2015). HyComp: An SMT-based model checker for hybrid systems. In C. Baier & C. Tinelli (Eds.), *Tools and algorithms for the construction and analysis of systems - 21st international conference, TACAS 2015, held as part of the European joint conferences on theory and practice of software, ETAPS 2015, London, UK, April 11-18, 2015. proceedings* (Vol. 9035, pp. 52–67). Springer. Retrieved from https://doi.org/10.1007/978-3-662-46681-0_4 doi: 10.1007/978-3-662-46681-0_4

Clarke, E. M. (1980). Proving correctness of coroutines without history variables. *Acta Inf.*, *13*, 169–188. Retrieved from https://doi.org/10.1007/BF00263992 doi: 10.1007/BF00263992

Clint, M. (1973). Program proving: Coroutines. *Acta Inf.*, *2*, 50–63. Retrieved from https://doi.org/10.1007/BF00571463 doi: 10.1007/BF00571463

Cohen, C., & Rouhling, D. (2017). A formal proof in Coq of LaSalle's invariance principle. In M. Ayala-Rincón & C. A. Muñoz (Eds.), *Interactive theorem proving - 8th international conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings* (Vol. 10499, pp. 148–163). Springer. Retrieved from https://doi.org/10.1007/978-3-319-66107-0_10 doi: 10.1007/978-3-319-66107-0_10

Constable, R., Allen, S., Bromley, H., Cleaveland, W., Cremer, J., Harper, R., ... Smith, S. (1986). *Implementing mathematics with the Nuprl proof development system*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Coq Proof Assistant. (1989). Retrieved from http://coq.inria.fr/ (Accessed: 2016-11-28)

Corbineau, P. (2007). A declarative language for the Coq proof assistant. In M. Miculan, I. Scagnetto, & F. Honsell (Eds.), *Types for proofs and programs, international conference, TYPES 2007, Cividale del Friuli, Italy, May 2-5, 2007, revised selected papers* (Vol. 4941, pp. 69–84). Springer. Retrieved from https://doi.org/10.1007/978-3-540-68103-8_5 doi: 10.1007/978-3-540-68103-8_5

Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., & Vanzetto, H. (2012). TLA$^+$ proofs. In D. Giannakopoulou & D. Méry (Eds.), *FM 2012: Formal methods - 18th international symposium, Paris, France, August 27-31, 2012. proceedings* (Vol. 7436, pp. 147–154). Springer. Retrieved from https://doi.org/10.1007/978-3-642-32759-9_14 doi: 10.1007/978-3-642-32759-9_14

Cuijpers, P. J. L., & Reniers, M. A. (2005). Hybrid process algebra. *J. Log. Algebr. Program.*, *62*(2), 191–245. Retrieved from https://doi.org/10.1016/j.jlap.2004.02.001 doi: 10.1016/j.jlap.2004.02.001

Daumas, M., Rideau, L., & Théry, L. (2001). A generic library for floating-point numbers and its application to exact computing. In R. J. Boulton & P. B. Jackson (Eds.), *TPHOLs* (Vol. 2152, pp. 169–184). Springer. Retrieved from https://doi.org/10.1007/3-540-44755-5_13 doi: 10.1007/3-540-44755-5_13

Davenport, J. H., & Heintz, J. (1988). Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, *5*(1-2), 29–35. Retrieved from http://ac.els-cdn.com/S074771718880004X/1-s2.0-S074771718880004X-main.pdf?_tid=277cf174-6392-11e5-8c77-00000aacb35e{&}acdnat=1443191688

_bbe0878b71d55d3e156e2fdc77b58ce7 doi: 10.1016/S0747-7171(88)80004-X

Davis, M. (1981). Obvious logical inferences. In P. J. Hayes (Ed.), *Proceedings of the 7th international joint conference on artificial intelligence, IJCAI '81, Vancouver, BC, Canada, August 24-28, 1981* (pp. 530–531). William Kaufmann. Retrieved from http://ijcai.org/Proceedings/81-1/Papers/095.pdf

Degen, J., & Werner, J. (2006). Towards intuitionistic dynamic logic. *Logic and Logical Philosophy*, *15*(4), 305–324.

Delahaye, D. (2000). A tactic language for the system Coq. In *Proceedings of the 7th international conference on logic for programming and automated reasoning* (pp. 85–95). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=1765236.1765246

Doberkat, E. (2011). Towards a coalgebraic interpretation of propositional dynamic logic. *CoRR*, *abs/1109.3685*. Retrieved from http://arxiv.org/abs/1109.3685

Dubins, L. E. (1957, July). On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *Amer. J. Math.*, *79*(3), 497+. Retrieved from http://dx.doi.org/10.2307/2372560

Duggirala, P. S., Mitra, S., Viswanathan, M., & Potok, M. (2015). C2E2: A verification tool for stateflow models. In C. Baier & C. Tinelli (Eds.), *TACAS* (Vol. 9035, pp. 68–82). Springer. Retrieved from https://doi.org/10.1007/978-3-662-46681-0_5 doi: 10.1007/978-3-662-46681-0_5

Fainekos, G. E., Girard, A., Kress-Gazit, H., & Pappas, G. J. (2009). Temporal logic motion planning for dynamic robots. *Automatica*, *45*(2). Retrieved from https://doi.org/10.1016/j.automatica.2008.08.008

Fernández-Duque, D. (2018). The intuitionistic temporal logic of dynamical systems. *Logical Methods in Computer Science*, *14*(3). Retrieved from https://doi.org/10.23638/LMCS-14(3:3)2018 doi: 10.23638/LMCS-14(3:3)2018

Filippidis, I., Dathathri, S., Livingston, S. C., Ozay, N., & Murray, R. M. (2016). Control design for hybrid systems with TuLiP: The temporal logic planning toolbox. In *CCA*. IEEE. Retrieved from https://doi.org/10.1109/CCA.2016.7587949

Finucane, C., Jing, G., & Kress-Gazit, H. (2010). LTLMoP: Experimenting with language, temporal logic and robot control. In *IROS*. IEEE. Retrieved from https://doi.org/10.1109/IROS.2010.5650371

Fox, D., Burgard, W., & Thrun, S. (1997). The dynamic window approach to collision avoidance. *IEEE Robot. Automat. Mag.*, *4*(1). Retrieved from https://doi.org/10.1109/100.580977

Franchetti, F., Low, T. M., Mitsch, S., Mendoza, J. P., Gui, L., Phaosawasdi, A., … Veloso, M. (2017). High-assurance SPIRAL: End-to-end guarantees for robot and car control. *IEEE Control Systems*, *37*(2), 82-103. doi: 10.1109/MCS.2016.2643244

Frehse, G., Guernic, C. L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., … Maler, O. (2011). SpaceEx: Scalable verification of hybrid systems. In G. Gopalakrishnan & S. Qadeer (Eds.), *CAV* (Vol. 6806, pp. 379–395). doi: 10.1007/978-3-642-22110-1_30

Frittella, S., Greco, G., Kurz, A., Palmigiano, A., & Sikimic, V. (2016). A proof-theoretic semantic analysis of dynamic epistemic logic. *J. Log. Comput.*, *26*(6), 1961–2015. Retrieved from https://doi.org/10.1093/logcom/exu063 doi: 10.1093/

logcom/exu063

Fulton, N., Mitsch, S., Bohrer, B., & Platzer, A. (2017). Bellerophon: Tactical theorem proving for hybrid systems. In M. Ayala-Rincón & C. A. Muñoz (Eds.), *ITP* (Vol. 10499, p. 207-224). Springer. doi: 10.1007/978-3-319-66107-0_14

Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., & Platzer, A. (2015). KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In A. P. Felty & A. Middeldorp (Eds.), *CADE* (Vol. 9195, pp. 527–538). Springer. doi: 10.1007/978-3-319-21401-6_36

Fulton, N., & Platzer, A. (2016). A logic of proofs for differential dynamic logic: Toward independently checkable proof certificates for dynamic logics. In J. Avigad & A. Chlipala (Eds.), *Proceedings of the 2016 conference on certified programs and proofs, CPP 2016, St. Petersburg, FL, USA, January 18-19, 2016* (p. 110-121). ACM. doi: 10.1145/2854065.2854078

Ghilardi, S. (1989). Presheaf semantics and independence results for some non-classical first-order logics. *Arch. Math. Log.*, *29*(2), 125–136. Retrieved from https://doi.org/10.1007/BF01620621 doi: 10.1007/BF01620621

Ghosh, S. (2008). Strategies made explicit in dynamic game logic. *Logic and the foundations of game and decision theory*.

Gonthier, G., & Mahboubi, A. (2010). An introduction to small scale reflection in Coq. *J. Formalized Reasoning*, *3*(2), 95–152. Retrieved from https://doi.org/10.6092/issn.1972-5787/1979 doi: 10.6092/issn.1972-5787/1979

Gonthier, G., & Tassi, E. (2012). A language of patterns for subterm selection. In L. Beringer & A. P. Felty (Eds.), *Interactive theorem proving - third international conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. proceedings* (Vol. 7406, pp. 361–376). Springer. Retrieved from https://doi.org/10.1007/978-3-642-32347-8_25 doi: 10.1007/978-3-642-32347-8_25

Goranko, V. (2003). The basic algebra of game equivalences. *Studia Logica*, *75*(2), 221–238. Retrieved from https://doi.org/10.1023/A:1027311011342 doi: 10.1023/A:1027311011342

Griffin, T. (1990). A formulae-as-types notion of control. In F. E. Allen (Ed.), *Conference record of the seventeenth annual ACM symposium on principles of programming languages, San Francisco, California, USA, January 1990* (pp. 47–58). ACM Press. Retrieved from https://doi.org/10.1145/96709.96714 doi: 10.1145/96709.96714

Guéneau, A., Myreen, M. O., Kumar, R., & Norrish, M. (2017). Verified characteristic formulae for CakeML. In H. Yang (Ed.), *ESOP* (Vol. 10201, pp. 584–610). Springer. Retrieved from https://doi.org/10.1007/978-3-662-54434-1_22 doi: 10.1007/978-3-662-54434-1_22

Habermalz, E. (2000). *Interactive theorem proving with schematic theory specific rules*. Univ., Fak. für Informatik, Bibliothek.

Halbwachs, N., Lagnier, F., & Ratel, C. (1992). Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Software Eng.*, *18*(9), 785–793. Retrieved from https://doi.org/10.1109/32.159839 doi: 10.1109/32.159839

Harper, R., Honsell, F., & Plotkin, G. D. (1993). A framework for defining logics. *J. ACM*, *40*(1), 143–184. Retrieved from https://doi.org/10.1145/138027.138060

doi: 10.1145/138027.138060

Harrison, J. (1996). A Mizar mode for HOL. In J. von Wright, J. Grundy, & J. Harrison (Eds.), *Theorem proving in higher order logics, 9th international conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, proceedings* (Vol. 1125, pp. 203–220). Springer. Retrieved from https://doi.org/10.1007/BFb0105406 doi: 10.1007/BFb0105406

Harrison, J. (2006a). Floating-point verification using theorem proving. In M. Bernardo & A. Cimatti (Eds.), *Formal methods for hardware verification, SFM* (Vol. 3965, pp. 211–242). Springer. Retrieved from https://doi.org/10.1007/11757283_8 doi: 10.1007/11757283_8

Harrison, J. (2006b). Towards self-verification of HOL Light. In U. Furbach & N. Shankar (Eds.), *IJCAR2006* (Vol. 4130, p. 177-191). Springer.

Henzinger, T. A. (1996). The theory of hybrid automata. In *Proceedings, 11th annual IEEE symposium on logic in computer science, New Brunswick, New Jersey, USA, July 27-30, 1996* (pp. 278–292). IEEE Comp. Soc. Retrieved from https://doi.org/10.1109/LICS.1996.561342 doi: 10.1109/LICS.1996.561342

Hilken, B. P., & Rydeheard, D. E. (2001). *A first order modal logic and its sheaf models.*

Ho, S., Abrahamsson, O., Kumar, R., Myreen, M. O., Tan, Y. K., & Norrish, M. (2018). Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. In D. Galmiche, S. Schulz, & R. Sebastiani (Eds.), *Automated reasoning - 9th international joint conference, IJCAR 2018, held as part of the federated logic conference, FloC 2018, Oxford, UK, July 14-17, 2018, proceedings* (Vol. 10900, pp. 646–662). Springer. Retrieved from https://doi.org/10.1007/978-3-319-94205-6_42 doi: 10.1007/978-3-319-94205-6_42

Huang, J., Erdogan, C., Zhang, Y., Moore, B. M., Luo, Q., Sundaresan, A., & Rosu, G. (2014). ROSRV: runtime verification for robots. In B. Bonakdarpour & S. A. Smolka (Eds.), *Runtime verification - 5th international conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings* (Vol. 8734, pp. 247–254). Springer. Retrieved from https://doi.org/10.1007/978-3-319-11164-3_20 doi: 10.1007/978-3-319-11164-3_20

Hubbard, J. H., & West, B. H. (2012). *Differential equations: A dynamical systems approach* (Vol. 18). Springer. doi: 10.1007/978-1-4612-4192-8

Hupel, L., & Nipkow, T. (2018). A verified compiler from Isabelle/HOL to CakeML. In A. Ahmed (Ed.), *ESOP*. Springer.

Immler, F. (2015). Verified reachability analysis of continuous systems. In (pp. 37–51). Retrieved from http://dx.doi.org/10.1007/978-3-662-46681-0_3 doi: 10.1007/978-3-662-46681-0_3

Immler, F., & Traut, C. (2016). The flow of ODEs. In J. C. Blanchette & S. Merz (Eds.), *ITP* (Vol. 9807, pp. 184–199). Springer. Retrieved from https://doi.org/10.1007/978-3-319-43144-4_12 doi: 10.1007/978-3-319-43144-4_12

Ioannidis, E., Kaashoek, M. F., & Zeldovich, N. (2019). Extracting and optimizing formally verified code for systems programming. In J. M. Badger & K. Y. Rozier (Eds.), *NASA formal methods - 11th international symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, proceedings* (Vol. 11460, pp. 228–236). Springer. Retrieved from https://doi.org/10.1007/978-3-030-20652-9_15 doi: 10.1007/978-3-030-20652-9_15

Jeannet, B. (2003). Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, *23*(1), 5–37. Retrieved from https://doi.org/10.1023/A:1024480913162 doi: 10.1023/A:1024480913162

Jeannin, J., Ghorbal, K., Kouskoulas, Y., Gardner, R., Schmidt, A., Zawadzki, E., & Platzer, A. (2015). Formal verification of ACAS X, an industrial airborne collision avoidance system. In A. Girault & N. Guan (Eds.), *EMSOFT* (p. 127-136). IEEE Press. doi: 10.1109/EMSOFT.2015.7318268

Kaliszyk, C., Pak, K., & Urban, J. (2016). Towards a Mizar environment for Isabelle: foundations and language. In J. Avigad & A. Chlipala (Eds.), *Proceedings of the 5th ACM SIGPLAN conference on certified programs and proofs, Saint Petersburg, FL, USA, January 20-22, 2016* (pp. 58–65). ACM. Retrieved from http://doi.acm.org/10.1145/2854065.2854070 doi: 10.1145/2854065.2854070

Kamide, N. (2010). Strong normalization of program-indexed lambda calculus. *Bull. Sect. Logic Univ. Łódź*, *39*(1-2), 65–78.

Kamide, N., & Wansing, H. (2010). Combining linear-time temporal logic with constructiveness and paraconsistency. *J. Applied Logic*, *8*(1), 33–61. Retrieved from https://doi.org/10.1016/j.jal.2009.06.001 doi: 10.1016/j.jal.2009.06.001

King, J. C. (1971). A program verifier. In C. V. Freiman, J. E. Griffith, & J. L. Rosenfeld (Eds.), *Information processing, proceedings of IFIP congress 1971, volume 1 - foundations and systems, Ljubljana, Yugoslavia, August 23-28, 1971*. (pp. 234–249). North-Holland.

Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., ... Winwood, S. (2010). seL4: Formal verification of an operating-system kernel. *Commun. ACM*, *53*(6), 107–115. doi: 10.1145/1743546.1743574

Klein, G., & Nipkow, T. (2006). A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, *28*(4), 619–695. Retrieved from http://doi.acm.org/10.1145/1146811 doi: 10.1145/1146811

Kleymann, T. (1999). Hoare logic and auxiliary variables. *Formal Asp. Comput.*, *11*(5), 541–566. Retrieved from https://doi.org/10.1007/s001650050057 doi: 10.1007/s001650050057

Kloetzer, M., & Belta, C. (2008). A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Trans. Automat. Contr.*, *53*(1), 287–297. Retrieved from https://doi.org/10.1109/TAC.2007.914952

Krebbers, R., & Spitters, B. (2011). Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science*, *9*(1). Retrieved from http://dx.doi.org/10.2168/LMCS-9(1:01)2013 doi: 10.2168/LMCS-9(1:01)2013

Krebbers, R., Timany, A., & Birkedal, L. (2017). Interactive proofs in higher-order concurrent separation logic. In G. Castagna & A. D. Gordon (Eds.), *Proceedings of the 44th ACM SIGPLAN symposium on principles of programming languages, POPL 2017, Paris, France, January 18-20, 2017* (pp. 205–217). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=3009855 doi: 10.1145/3009837

Kumar, R., Arthan, R., Myreen, M. O., & Owens, S. (2016). Self-formalisation of higher-order logic: Semantics, soundness, and a verified implementation. *J. Autom. Reasoning*, *56*(3), 221–259. doi: 10.1007/s10817-015-9357-x

Kumar, R., Myreen, M. O., Norrish, M., & Owens, S. (2014). CakeML: A verified implementation of ML. In S. Jagannathan & P. Sewell (Eds.), *POPL* (pp. 179–192). ACM. Retrieved from http://doi.acm.org/10.1145/2535838.2535841 doi: 10.1145/2535838.2535841

Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, *3*(2), 125–143. Retrieved from https://doi.org/10.1109/TSE.1977.229904 doi: 10.1109/TSE.1977.229904

Lamport, L. (1992). Hybrid systems in TLA$^+$. In R. L. Grossman, A. Nerode, A. P. Ravn, & H. Rischel (Eds.), *Hybrid systems* (Vol. 736, pp. 77–102). Springer. Retrieved from http://dx.doi.org/10.1007/3-540-57318-6_25 doi: 10.1007/3-540-57318-6_25

Lamport, L. (1995). How to write a proof. *American Mathematical Monthly*, *102*(7), 600–608. Retrieved from http://lamport.azurewebsites.net/pubs/lamport-how-to-write.pdf

Lamport, L. (2012). How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*. doi: 10.1007/s11784-012-0071-6

Leavens, G. T., Baker, A. L., & Ruby, C. (1999). JML: A notation for detailed design. In H. Kilov, B. Rumpe, & I. Simmonds (Eds.), *Behavioral specifications of businesses and systems* (Vol. 523, pp. 175–188). Springer. Retrieved from https://doi.org/10.1007/978-1-4615-5229-1_12 doi: 10.1007/978-1-4615-5229-1_12

Leino, K. R. M. (2008, jun). This is Boogie 2. Microsoft Research. Retrieved from https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In E. M. Clarke & A. Voronkov (Eds.), *Logic for programming, artificial intelligence, and reasoning - 16th international conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, revised selected papers* (Vol. 6355, pp. 348–370). Springer. Retrieved from https://doi.org/10.1007/978-3-642-17511-4_20 doi: 10.1007/978-3-642-17511-4_20

Leino, K. R. M., Müller, P., & Smans, J. (2009). Verification of concurrent programs with chalice. In A. Aldini, G. Barthe, & R. Gorrieri (Eds.), *Foundations of security analysis and design v, FOSAD 2007/2008/2009 tutorial lectures* (Vol. 5705, pp. 195–222). Springer. Retrieved from https://doi.org/10.1007/978-3-642-03829-7_7 doi: 10.1007/978-3-642-03829-7_7

Leroy, X. (2006). Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett & S. L. P. Jones (Eds.), *POPL 2006* (pp. 42–54). ACM. Retrieved from http://doi.acm.org/10.1145/1111037.1111042 doi: 10.1145/1111037.1111042

Lipton, J. (1992). Constructive Kripke semantics and realizability. In *Logic from computer science* (pp. 319–357).

Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., & Zou, L. (2010). A calculus for hybrid CSP. In K. Ueda (Ed.), *Programming languages and systems - 8th Asian symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. proceedings* (Vol. 6461, pp. 1–15). Springer. Retrieved from https://doi.org/10.1007/978-3-642

-17164-2_1 doi: 10.1007/978-3-642-17164-2_1

Lochbihler, A. (2007). Jinja with threads. *Archive of Formal Proofs*, *2007*. Retrieved from https://www.isa-afp.org/entries/JinjaThreads.shtml

Lombardi, H. (2019). Théories géométriques pour lalgébre des nombres réels sans test de signe ni axiome de choix dépendant.
(Accessed: 2019-07-10. Unpublished draft (in French))

Loos, S. M., & Platzer, A. (2016). Differential refinement logic. In M. Grohe, E. Koskinen, & N. Shankar (Eds.), *LICS* (p. 505-514). ACM. doi: 10.1145/2933575.2934555

Loos, S. M., Platzer, A., & Nistor, L. (2011). Adaptive cruise control: Hybrid, distributed, and now formally verified. In M. Butler & W. Schulte (Eds.), *FM* (Vol. 6664, p. 42-56). Springer. doi: 10.1007/978-3-642-21437-0_6

Low, T. M., & Franchetti, F. (2017). High assurance code generation for cyber-physical systems. In *18th IEEE international symposium on high assurance systems engineering, HASE 2017, Singapore, January 12-14, 2017* (pp. 104–111). IEEE Comp. Soc. Retrieved from https://doi.org/10.1109/HASE.2017.28 doi: 10.1109/HASE.2017.28

Majumdar, R., Saha, I., & Zamani, M. (2012). Synthesis of minimal-error control software. In A. Jerraya, L. P. Carloni, F. Maraninchi, & J. Regehr (Eds.), *EMSOFT* (pp. 123–132). ACM. Retrieved from http://doi.acm.org/10.1145/2380356.2380380 doi: 10.1145/2380356.2380380

Malecha, G., & Bengtson, J. (2015, January). Rtac: A fully reflective tactic language. In *CoqPL'15.*

Mamouras, K. (2016). Synthesis of strategies using the Hoare logic of angelic and demonic nondeterminism. *Logical Methods in Computer Science*, *12*(3). Retrieved from https://doi.org/10.2168/LMCS-12(3:6)2016 doi: 10.2168/LMCS-12(3:6)2016

Martens, C., & Crary, K. (2012). LF in LF: Mechanizing the metatheories of LF in Twelf. In *Proceedings of the seventh international workshop on logical frameworks and meta-languages, theory and practice* (pp. 23–32).

Martin, B., Ghorbal, K., Goubault, E., & Putot, S. (2017). Formal verification of station keeping maneuvers for a planar autonomous hybrid system. In L. Bulwahn, M. Kamali, & S. Linker (Eds.), *FVAV@iFM* (Vol. 257, pp. 91–104). doi: 10.4204/EPTCS.257.9

Martinez, A. A., Majumdar, R., Saha, I., & Tabuada, P. (2010). Automatic verification of control system implementations. In L. P. Carloni & S. Tripakis (Eds.), *EMSOFT* (pp. 9–18). ACM. Retrieved from http://doi.acm.org/10.1145/1879021.1879024 doi: 10.1145/1879021.1879024

Matichuk, D., Murray, T., & Wenzel, M. (2016, March). Eisbach: A proof method language for Isabelle. *J. Autom. Reason.*, *56*(3), 261–282. Retrieved from http://dx.doi.org/10.1007/s10817-015-9360-2 doi: 10.1007/s10817-015-9360-2

Melquiond, G. (2012). Floating-point arithmetic in the Coq system. *Inf. Comput.*, *216*, 14–23. Retrieved from https://doi.org/10.1016/j.ic.2011.09.005 doi: 10.1016/j.ic.2011.09.005

Mitsch, S., Ghorbal, K., & Platzer, A. (2013). On provably safe obstacle avoidance for autonomous robotic ground vehicles. In P. Newman, D. Fox, & D. Hsu (Eds.), *Robotics: Science and systems.*

Mitsch, S., Ghorbal, K., Vogelbacher, D., & Platzer, A. (2017). Formal verification of obstacle

avoidance and navigation of ground robots. *I. J. Robotics Res.*, *36*(12), 1312-1340. doi: 10.1177/0278364917733549

Mitsch, S., & Platzer, A. (2016). ModelPlex: Verified runtime validation of verified cyber-physical system models. *Form. Methods Syst. Des.*, *49*(1), 33-74. (Special issue of selected papers from RV'14) doi: 10.1007/s10703-016-0241-z

Mullen, E., Pernsteiner, S., Wilcox, J. R., Tatlock, Z., & Grossman, D. (2018). Œuf: minimizing the Coq extraction TCB. In J. Andronick & A. P. Felty (Eds.), *Proceedings of the 7th ACM SIGPLAN international conference on certified programs and proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018* (pp. 172–185). ACM. Retrieved from https://doi.org/10.1145/3167089 doi: 10.1145/3167089

Myreen, M. O., & Davis, J. (2014). The reflective Milawa theorem prover is sound - (down to the machine code that runs it). In (pp. 421–436). Retrieved from http://dx.doi.org/10.1007/978-3-319-08970-6_27 doi: 10.1007/978-3-319-08970-6_27

Myreen, M. O., & Owens, S. (2012). Proof-producing synthesis of ML from higher-order logic. In P. Thiemann & R. B. Findler (Eds.), *ICFP* (pp. 115–126). ACM. Retrieved from http://doi.acm.org/10.1145/2364527.2364545 doi: 10.1145/2364527.2364545

Myreen, M. O., Owens, S., & Kumar, R. (2013). Steps towards verified implementations of HOL Light. In *ITP* (pp. 490–495).

Nilsson, P., Hussien, O., Balkan, A., Chen, Y., Ames, A. D., Grizzle, J. W., ... Tabuada, P. (2016). Correct-by-construction adaptive cruise control: Two approaches. *IEEE Trans. Contr. Sys. Techn.*, *24*(4). Retrieved from https://doi.org/10.1109/TCST.2015.2501351

Nipkow, T. (2002). Hoare Logics in Isabelle/HOL. In H. Schwichtenberg & R. Steinbrüggen (Eds.), *Proof and system-reliability* (pp. 341–367). Dordrecht: Springer Netherlands. Retrieved from http://dx.doi.org/10.1007/978-94-010-0413-8{_}11 doi: 10.1007/978-94-010-0413-8_11

Nipkow, T., Paulson, L. C., & Wenzel, M. (2002). *Isabelle/HOL: A proof assistant for higher-order logic* (Vol. 2283). Springer. Retrieved from https://doi.org/10.1007/3-540-45949-9 doi: 10.1007/3-540-45949-9

Noschinski, L. (2015). Generating cases from labeled subgoals. *Archive of Formal Proofs*, *2015*. Retrieved from https://www.isa-afp.org/entries/Case_Labeling.shtml

Owicki, S., & Gries, D. (1976, dec). An axiomatic proof technique for parallel programs I. *Acta Informatica*, *6*(4), 319–340. Retrieved from http://dx.doi.org/10.1007/BF00268134 doi: 10.1007/BF00268134

Owicki, S. S. (1975). *Axiomatic proof techniques for parallel programs*. Garland Publishing, New York.

Padon, O., McMillan, K. L., Panda, A., Sagiv, M., & Shoham, S. (2016). Ivy: safety verification by interactive generalization. In C. Krintz & E. Berger (Eds.), *Proceedings of the 37th ACM SIGPLAN conference on programming language design and implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016* (pp. 614–630). ACM. Retrieved from https://doi.org/10.1145/2908080.2908118 doi: 10.1145/2908080.2908118

Parikh, R. (1983). Propositional game logic. In *24th annual symposium on foundations of computer science, Tucson, Arizona, USA, 7-9 November 1983* (pp. 195–200). IEEE Comp. Soc. Retrieved from https://doi.org/10.1109/SFCS.1983.47 doi: 10.1109/SFCS.1983.47

Pauly, M. (2002). A modal logic for coalitional power in games. *J. Log. Comput.*, *12*(1), 149–166. Retrieved from https://doi.org/10.1093/logcom/12.1.149 doi: 10.1093/logcom/12.1.149

Pauly, M., & Parikh, R. (2003). Game logic - an overview. *Studia Logica*, *75*(2), 165–182. Retrieved from https://doi.org/10.1023/A:1027354826364 doi: 10.1023/A:1027354826364

Peleg, D. (1987). Concurrent dynamic logic. *J. ACM*, *34*(2), 450–479. Retrieved from https://doi.org/10.1145/23005.23008 doi: 10.1145/23005.23008

Pereira, A., & Althoff, M. (2015). Safety control of robots under computed torque control using reachable sets. In *IEEE international conference on robotics and automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015* (pp. 331–338). IEEE. Retrieved from https://doi.org/10.1109/ICRA.2015.7139020 doi: 10.1109/ICRA.2015.7139020

Pfenning, F., & Elliott, C. (1988). Higher-order abstract syntax. In R. L. Wexelblat (Ed.), *Proceedings of the ACM sigplan'88 conference on programming language design and implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988* (pp. 199–208). ACM. Retrieved from https://doi.org/10.1145/53990.54010 doi: 10.1145/53990.54010

Pfenning, F., & Schürmann, C. (1999). System description: Twelf - A meta-logical framework for deductive systems. In H. Ganzinger (Ed.), *Automated deduction - CADE-16, 16th international conference on automated deduction, Trento, Italy, July 7-10, 1999, proceedings* (Vol. 1632, pp. 202–206). Springer. Retrieved from https://doi.org/10.1007/3-540-48660-7_14 doi: 10.1007/3-540-48660-7_14

Platzer, A. (2007, Jun). Towards a hybrid dynamic logic for hybrid dynamic systems. In P. Blackburn, T. Bolander, T. Braüner, V. de Paiva, & J. Villadsen (Eds.), *International workshop on hybrid logic, HyLo'06, Seattle, USA, proceedings* (Vol. 174, p. 63-77). doi: 10.1016/j.entcs.2006.11.026

Platzer, A. (2008). Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, *41*(2), 143-189. doi: 10.1007/s10817-008-9103-8

Platzer, A. (2011, November). *The complete proof theory of hybrid systems* (Tech. Rep. No. CMU-CS-11-144). Pittsburgh, PA: School of Computer Science, Carnegie Mellon University.

Platzer, A. (2012a). A complete axiomatization of quantified differential dynamic logic for distributed hybrid systems. *Logical Methods in Computer Science*, *8*(4), 1-44. (Special issue for selected papers from CSL'10) doi: 10.2168/LMCS-8(4:17)2012

Platzer, A. (2012b). The complete proof theory of hybrid systems. In *LICS* (p. 541-550). IEEE. doi: 10.1109/LICS.2012.64

Platzer, A. (2012c). Logics of dynamical systems. In *LICS* (p. 13-24). IEEE. doi: 10.1109/LICS.2012.13

Platzer, A. (2015). Differential game logic. *ACM Trans. Comput. Log.*, *17*(1), 1:1–1:51. doi: 10.1145/2817824

Platzer, A. (2016). Logic & proofs for cyber-physical systems. In N. Olivetti & A. Tiwari (Eds.), *Ijcar* (Vol. 9706, p. 15-21). Springer. doi: 10.1007/978-3-319-40229-1_3

Platzer, A. (2017a). A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.*, *59*(2), 219-265. doi: 10.1007/s10817-016-9385-1

Platzer, A. (2017b). Differential hybrid games. *ACM Trans. Comput. Log.*, *18*(3), 19:1-19:44. doi: 10.1145/3091123

Platzer, A. (2018a). *Logical foundations of cyber-physical systems*. Springer. Retrieved from https://doi.org/10.1007/978-3-319-63588-0

Platzer, A. (2018b). Uniform substitution for differential game logic. In D. Galmiche, S. Schulz, & R. Sebastiani (Eds.), *IJCAR* (Vol. 10900, p. 211-227). Springer. doi: 10.1007/978-3-319-94205-6_15

Platzer, A. (2019). Differential game logic. *Archive of Formal Proofs*, *2019*. Retrieved from https://www.isa-afp.org/entries/Differential_Game_Logic.html

Platzer, A., & Quesel, J.-D. (2008a). KeYmaera: A hybrid theorem prover for hybrid systems. In A. Armando, P. Baumgartner, & G. Dowek (Eds.), *IJCAR* (Vol. 5195, p. 171-178). Springer. doi: 10.1007/978-3-540-71070-7_15

Platzer, A., & Quesel, J.-D. (2008b). Logical verification and systematic parametric analysis in train control. In M. Egerstedt & B. Mishra (Eds.), *HSCC* (Vol. 4981, p. 646-649). Springer. doi: 10.1007/978-3-540-78929-1_55

Platzer, A., & Tan, Y. K. (2018). Differential equation axiomatization: The impressive power of differential ghosts. In A. Dawar & E. Grädel (Eds.), *LICS* (p. 819-828). New York: ACM. doi: 10.1145/3209108.3209147

Polaris Industries, I. (2019). *MRZR D4*. Retrieved from https://military.polaris.com/en-us/mrzr-d4/ (Accessed: Sep 12, 2019)

Prabhakar, P., & Köpf, B. (2013). Verifying information flow properties of hybrid systems. In L. Bushnell, L. Rohrbough, S. Amin, & X. D. Koutsoukos (Eds.), *2nd ACM international conference on high confidence networked systems (part of CPS week), HiCoNS 2013, Philadelphia, PA, USA, April 9-11, 2013* (pp. 77–84). ACM. Retrieved from https://doi.org/10.1145/2461446.2461458 doi: 10.1145/2461446.2461458

Quesel, J.-D., & Platzer, A. (2012). Playing hybrid games with KeYmaera. In B. Gramlich, D. Miller, & U. Sattler (Eds.), *IJCAR* (Vol. 7364, p. 439-453). Springer. doi: 10.1007/978-3-642-31365-3_34

Rahli, V., & Bickford, M. (2016). A nominal exploration of intuitionism. In J. Avigad & A. Chlipala (Eds.), *CPP2016* (pp. 130–141). ACM. Retrieved from http://doi.acm.org/10.1145/2854065.2854077 doi: 10.1145/2854065.2854077

Ramanujam, R., & Simon, S. E. (2008). Dynamic logic on games with structured strategies. In G. Brewka & J. Lang (Eds.), *Principles of knowledge representation and reasoning: Proceedings of the eleventh international conference, KR 2008, Sydney, Australia, September 16-19, 2008* (pp. 49–58). AAAI Press. Retrieved from http://www.aaai.org/Library/KR/2008/kr08-006.php

Ricketts, D., Malecha, G., Alvarez, M. M., Gowda, V., & Lerner, S. (2015). Towards verification of hybrid systems in a foundational proof assistant. In *MEMOCODE 2015* (pp. 248–257). IEEE. Retrieved from http://dx.doi.org/10.1109/MEMCOD.2015.7340492 doi: 10.1109/MEMCOD.2015.7340492

Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., Tatlock, Z., et al. (2019). QED at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, *5*(2-3), 102–281.

Rizaldi, A., Immler, F., Schürmann, B., & Althoff, M. (2018). A formally verified motion planner for autonomous vehicles. In *ATVA* (Vol. 11138). Springer. Retrieved from https://doi.org/10.1007/978-3-030-01090-4_5

Rizaldi, A., Keinholz, J., Huber, M., Feldle, J., Immler, F., Althoff, M., ... Nipkow, T. (2017). Formalising and monitoring traffic rules for autonomous vehicles in Isabelle/HOL. In N. Polikarpova & S. Schneider (Eds.), *Integrated formal methods - 13th international conference, IFM 2017, Turin, Italy, September 20-22, 2017, proceedings* (Vol. 10510, pp. 50–66). Springer. Retrieved from https://doi.org/10.1007/978-3-319-66845-1_4 doi: 10.1007/978-3-319-66845-1_4

Rouhling, D. (2018). A formal proof in Coq of a control function for the inverted pendulum. In J. Andronick & A. P. Felty (Eds.), *Proceedings of the 7th ACM SIGPLAN international conference on certified programs and proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018* (pp. 28–41). ACM. Retrieved from https://doi.org/10.1145/3167101 doi: 10.1145/3167101

Ruchkin, I., Sunshine, J., Iraci, G., Schmerl, B. R., & Garlan, D. (2018). IPL: an integration property language for multi-model cyber-physical systems. In K. Havelund, J. Peleska, B. Roscoe, & E. P. de Vink (Eds.), *Formal methods - 22nd international symposium, FM 2018, held as part of the federated logic conference, FloC 2018, Oxford, UK, July 15-17, 2018, proceedings* (Vol. 10951, pp. 165–184). Springer. Retrieved from https://doi.org/10.1007/978-3-319-95582-7_10 doi: 10.1007/978-3-319-95582-7_10

Schiffelers, R. R. H., van Beek, D. A., Man, K. L., Reniers, M. A., & Rooda, J. E. (2003). Formal semantics of Hybrid Chi. In K. G. Larsen & P. Niebert (Eds.), *Formal modeling and analysis of timed systems: First international workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. revised papers* (Vol. 2791, pp. 151–165). Springer. Retrieved from https://doi.org/10.1007/978-3-540-40903-8_12 doi: 10.1007/978-3-540-40903-8_12

Schwichtenberg, H. (2008). Realizability interpretation of proofs in constructive analysis. *Theory Comput. Syst.*, *43*(3-4), 583–602. Retrieved from https://doi.org/10.1007/s00224-007-9027-4 doi: 10.1007/s00224-007-9027-4

Seto, D., Krogh, B., Sha, L., & Chutinan, A. (1998, June). The SIMPLEX architecture for safe on-line control system upgrades. In *ACC*. Retrieved from https://doi.org/10.1109/ACC.1998.703255

Sewell, T. A. L., Myreen, M. O., & Klein, G. (2013). Translation validation for a verified OS kernel. In H. Boehm & C. Flanagan (Eds.), *PLDI* (pp. 471–482). ACM. Retrieved from http://doi.acm.org/10.1145/2462156.2462183 doi: 10.1145/2462156.2462183

Shah, S., Dey, D., Lovett, C., & Kapoor, A. (2018). AirSim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field serv. robot.* (Vol. 5, pp. 621–635).

Stampoulis, A., & Shao, Z. (2010). VeriML: typed computation of logical terms inside a language with effects. In P. Hudak & S. Weirich (Eds.), *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore,*

Maryland, USA, September 27-29, 2010 (pp. 333–344). ACM. Retrieved from http://doi.acm.org/10.1145/1863543.1863591 doi: 10.1145/1863543.1863591

Stampoulis, A., & Shao, Z. (2012). Static and user-extensible proof checking. In J. Field & M. Hicks (Eds.), *Proceedings of the 39th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012* (pp. 273–284). ACM. Retrieved from http://doi.acm.org/10.1145/2103656.2103690 doi: 10.1145/2103656.2103690

Suenaga, K., & Hasuo, I. (2011). Programming with infinitesimals: A while-language for hybrid system modeling. In L. Aceto, M. Henzinger, & J. Sgall (Eds.), *Automata, languages and programming - 38th international colloquium, ICALP 2011, zurich, switzerland, july 4-8, 2011, proceedings, part II* (Vol. 6756, pp. 392–403). Springer. Retrieved from https://doi.org/10.1007/978-3-642-22012-8_31 doi: 10.1007/978-3-642-22012-8_31

Syme, D. (1997). DECLARE: A prototype declarative proof system for higher order logic..

Taly, A., & Tiwari, A. (2010). Switching logic synthesis for reachability. In L. P. Carloni & S. Tripakis (Eds.), *EMSOFT* (pp. 19–28). ACM. Retrieved from https://doi.org/10.1145/1879021.1879025

Tan, Y. K., Myreen, M. O., Kumar, R., Fox, A. C. J., Owens, S., & Norrish, M. (2016). A new verified compiler backend for CakeML. In J. Garrigue, G. Keller, & E. Sumii (Eds.), *ICFP* (pp. 60–73). ACM. Retrieved from http://doi.acm.org/10.1145/2951913.2951924 doi: 10.1145/2951913.2951924

Tarski, A. (1951, reprinting 1998). A decision method for elementary algebra and geometry. In B. F. Caviness & J. R. Johnson (Eds.), *Quantifier elimination and cylindrical algebraic decomposition* (pp. 24–84). Vienna: Springer.

Traut, C., & Noschinski, L. (2014). Pattern-based subterm selection in Isabelle. In *Proceedings of Isabelle workshop 2014*.

Van Benthem, J. (2001). Games in dynamic-epistemic logic. *Bulletin of Economic Research*, *53*(4), 219–248.

van Benthem, J. (2015). Logic of strategies: What and how? In J. van Benthem, S. Ghosh, & R. Verbrugge (Eds.), *Models of strategic reasoning - logics, games, and communities* (Vol. 8972, pp. 321–332). Springer. Retrieved from https://doi.org/10.1007/978-3-662-48540-8_10 doi: 10.1007/978-3-662-48540-8_10

van Benthem, J., & Bezhanishvili, G. (2007). Modal logics of space. In M. Aiello, I. Pratt-Hartmann, & J. van Benthem (Eds.), *Handbook of spatial logics* (pp. 217–298). Springer. Retrieved from https://doi.org/10.1007/978-1-4020-5587-4_5 doi: 10.1007/978-1-4020-5587-4_5

van Benthem, J., Bezhanishvili, N., & Enqvist, S. (2017). A propositional dynamic logic for instantial neighborhood models. In A. Baltag, J. Seligman, & T. Yamada (Eds.), *Logic, rationality, and interaction - 6th international workshop, LORI 2017, Sapporo, Japan, September 11-14, 2017, proceedings* (Vol. 10455, pp. 137–150). Springer. Retrieved from https://doi.org/10.1007/978-3-662-55665-8_10 doi: 10.1007/978-3-662-55665-8_10

van Benthem, J., & Pacuit, E. (2011). Dynamic logics of evidence-based beliefs. *Studia Logica*, *99*(1-3), 61–92. Retrieved from https://doi.org/10.1007/s11225-011

-9347-x doi: 10.1007/s11225-011-9347-x

van Benthem, J., Pacuit, E., & Roy, O. (2011). Toward a theory of play: A logical perspective on games and interaction. *Games*, *2*(1), 52–86. Retrieved from https://doi.org/10.3390/g2010052 doi: 10.3390/g2010052

van der Hoek, W., Jamroga, W., & Wooldridge, M. J. (2005). A logic for strategic reasoning. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, & M. J. Wooldridge (Eds.), *4th international joint conference on autonomous agents and multiagent systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands* (pp. 157–164). ACM. Retrieved from https://doi.org/10.1145/1082473.1082497 doi: 10.1145/1082473.1082497

van Oosten, J. (2002). Realizability: A historical essay. *Mathematical Structures in Computer Science*, *12*(3), 239–263. Retrieved from https://doi.org/10.1017/S0960129502003626 doi: 10.1017/S0960129502003626

Weihrauch, K. (2000). *Computable analysis - an introduction*. Springer. Retrieved from https://doi.org/10.1007/978-3-642-56999-9 doi: 10.1007/978-3-642-56999-9

Weispfenning, V. (1997). Quantifier elimination for real algebra - the quadratic case and beyond. *Appl. Algebra Eng. Commun. Comput.*, *8*(2), 85–101. Retrieved from https://doi.org/10.1007/s002000050055 doi: 10.1007/s002000050055

Wenzel, M. (1999). Isar - A generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin-Mohring, & L. Théry (Eds.), *Theorem proving in higher order logics, 12th international conference, TPHOLs'99, Nice, France, September, 1999, proceedings* (Vol. 1690, pp. 167–184). Springer. Retrieved from https://doi.org/10.1007/3-540-48256-3_12 doi: 10.1007/3-540-48256-3_12

Wenzel, M. (2006). Structured induction proofs in Isabelle/Isar. In J. M. Borwein & W. M. Farmer (Eds.), *Mathematical knowledge management, 5th international conference, MKM 2006, Wokingham, UK, August 11-12, 2006, proceedings* (Vol. 4108, pp. 17–30). Springer. Retrieved from https://doi.org/10.1007/11812289_3 doi: 10.1007/11812289_3

Wenzel, M. (2007). Isabelle/Isar – a generic framework for human-readable proof documents. In *University of Białystok.*

Wenzel, M., & Wiedijk, F. (2002). A comparison of Mizar and Isar. *J. Autom. Reasoning*, *29*(3-4), 389–411. Retrieved from https://doi.org/10.1023/A:1021935419355 doi: 10.1023/A:1021935419355

Wiedijk, F. (2001). Mizar light for HOL light. In R. J. Boulton & P. B. Jackson (Eds.), *Theorem proving in higher order logics, 14th international conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, proceedings* (Vol. 2152, pp. 378–394). Springer. Retrieved from https://doi.org/10.1007/3-540-44755-5_26 doi: 10.1007/3-540-44755-5_26

Wijesekera, D. (1990). Constructive modal logics I. *Ann. Pure Appl. Logic*, *50*(3), 271–301. Retrieved from https://doi.org/10.1016/0168-0072(90)90059-B doi: 10.1016/0168-0072(90)90059-B

Wijesekera, D., & Nerode, A. (2005). Tableaux for constructive concurrent dynamic logic. *Ann.*

Pure Appl. Logic, *135*(1-3), 1–72. Retrieved from https://doi.org/10.1016/j.apal.2004.12.001 doi: 10.1016/j.apal.2004.12.001

Yu, L. (2013). A formal model of IEEE floating point arithmetic. *Archive of Formal Proofs*. Retrieved from https://www.isa-afp.org/entries/IEEE_Floating_Point.shtml

Zaliva, V., & Franchetti, F. (2018). HELIX: a case study of a formal verification of high performance program generation. In K. Davis & M. Rainey (Eds.), *Proceedings of the 7th ACM SIGPLAN international workshop on functional high-performance computing, FHPC@ICFP 2018, St. Louis, MO, USA, September 29, 2018* (pp. 1–9). ACM. Retrieved from https://doi.org/10.1145/3264738.3264739 doi: 10.1145/3264738.3264739

Zhou, C., Wang, J., & Ravn, A. P. (1995). A formal description of hybrid systems. In R. Alur, T. A. Henzinger, & E. D. Sontag (Eds.), *Hybrid systems III: verification and control, proceedings of the DIMACS/SYCON workshop on verification and control of hybrid systems, October 22-25, 1995, Rutgers University, New Brunswick, NJ, USA* (Vol. 1066, pp. 511–530). Springer. Retrieved from https://doi.org/10.1007/BFb0020972 doi: 10.1007/BFb0020972

Ziliani, B., Dreyer, D., Krishnaswami, N. R., Nanevski, A., & Vafeiadis, V. (2013, September). Mtac: A monad for typed tactic programming in Coq. *SIGPLAN Not.*, *48*(9), 87–100. Retrieved from http://doi.acm.org/10.1145/2544174.2500579 doi: 10.1145/2544174.2500579