Verified Control Envelope Synthesis

Aditi Kabra

CMU-CS-26-118
May 2026

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
André Platzer, Co-chair
Stefan Mitsch, Co-chair
Eunsuk Kang
Armando Solar-Lezama (Massachusetts Institute of Technology)

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

Copyright © 2026 **Aditi Kabra**

This work is based on research that was sponsored by the Federal Railroad Administration Office of Research, Development and Technology under contract number 693JJ620C000025, a Swartz Center Innovation Commercialization Fellowship, the National Science Foundation (NSF) under award numbers 1739629, 2220311, and 2427581, by an Alexander von Humboldt Professorship, DARPA under award number FA8750-24-9-1000, and by the NVIDIA Academic Grant Program using NVIDIA DGX Spark.

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

Keywords: Formal Methods, Synthesis, Cyber-physical System, Hybrid Games, Control Envelopes

For Vandana Kabra,
who spent a large part of her life cultivating mine.

Abstract

Many cyber-physical systems, such as trains, planes, and self-driving cars, are safety-critical but difficult to reason about. The task of designing controllers for such systems is complex (the subject of an entire field, control theory), made even more challenging by the need to ensure correctness over all of the infinitely many possible scenarios that the system may face. This thesis develops techniques that let computers *automatically* synthesize the conditions that define correct control solutions, with *mathematical guarantees* that these conditions are correct.

Symbolic control envelopes are our representation of the control conditions that characterize *sets of safe control solutions*. They are represented parametrically in symbols that can be instantiated with any real-valued input (e.g., for a train control envelope, train weight w). Control envelopes provide a path to designing complex controllers that still have mathematical correctness guarantees by allowing separation of concerns during controller design. A verified (i.e., mathematically correct) safe control envelope can first identify the set of control solutions that ensure the safety-critical control objectives, and then non-formal techniques, e.g., machine learning, can optimize within that envelope for secondary objectives.

This thesis starts by designing, as a target for automation, the first verified train control envelope that accounts for all the forces in a realistic train kinematics model, demonstrating the feasibility and potential benefits of applying verified control envelopes to a realistic industrial-scale system. The thesis then automates the process of designing such a control envelope by creating the first framework for *symbolic control envelope synthesis*. The framework takes as input the *shape* of a control system, which indicates what control and environment behaviors are physically possible and what the desired control behavior is, making the synthesis question well-defined. It then automatically identifies the *symbolic control conditions* indicating when a given control action is correct, which is shown to correspond to the nondeterministic control policies of players in *hybrid games* (games with both continuous and discrete dynamics).

This thesis tackles the hybrid game control envelope synthesis problem in its full generality, developing the theory to solve for *all of differential game logic* (two-player, zero-sum games). It introduces a specialized algorithm, CESAR, to solve for an important class of control problems (time-triggered control, where the controller repeatedly polls to provide control decisions with some maximum time latency). By strategically using AI along with verification, it provides a general framework that leverages large language models for sound, scalable synthesis.

Acknowledgments

My advisors, André Platzer and Stefan Mitsch, provided invaluable wisdom and guidance. Their support and mentorship made this thesis possible. I thank my committee members, Eunsuk Kang and Armando Solar-Lezama, for their time and expertise. I am grateful to Jonathan Laurent, who I learned a lot from over the course of our long and fruitful collaboration, and Ruben Martins, whose support and mentorship I have greatly benefited from.

This thesis, and my Ph.D. more generally, has benefited from insightful conversations and feedback from members of the lab, past and present, including Rose Bohrer, Yong Kiam Tan, Katherine Kosaian, Noah Abou El Wafa, Long Qian, Marvin Brieger, Enguerrand Prebet, Jonathan Hellwig, Julia Butte, and Promit Panja. I am also grateful for collaborations with Christof Budnik, Georgi Markov, Parinitha Nagaraja, Luis Garcia, and Jainta Paul.

I am grateful for a friend group whose collective wisdom and support has had a strong positive influence on this Ph.D. and on my life. A non-exhaustive list of people who come to mind is Kush Jain, Jovina Vaswani, Cheta Sheth, Katy Voor, Matilda Wilson, Ritam Dutt, Sanjith Athlur, Margarida Ferreira, Carolina Carreira, Yuze He, Ningning Yang, Anup Agarwal, Nirav Atre, Adithya Philip, Hugo Sadok, Ian Dardik, Samvid Dharanikota, Nikitha Rao, Magdalen Dobson Manohar, Dorian Chen, Myra Dotzel, Wan Shen Lim, Ankush Jain, Tarana Laroia, and many more.

My family's love, support, and encouragement ensured that the Ph.D. journey never felt too lonely or insurmountable. My grandparents, Badrinarayan and Nirmala Chhaparwal, and Sarjubai Kabra provided their affection, *aashirvad*, and a constant interest in the progress of my work. My aunts, uncles, cousins, in-laws and extended family have been a great source of joy and encouragement. I especially thank my parents, Vandana and Nandkishor Kabra, my brother, Ameya Kabra, and my husband, Sagar Bharadwaj, for their unconditional support and care.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cyber-physical systems (CPS) are systems like trains, planes, and self-driving cars, where discrete software interacts with continuous physics. These systems are becoming an increasingly prominent part of our lives, and are often safety-critical, since software errors can have tragic or expensive consequences. For such systems, designing correct controllers is difficult, and requires careful reasoning about what the system must do *now* to stay safe *forever*, accounting precisely for environmental nondeterminism, interacting forces, and strategic control over an uncountably infinite state space. This is the subject of an entire field, control theory. Can we *automatically* identify the principles that will yield correct control for a given control system, with *mathematical guarantees* that our identification is correct? We seek not a single numeric control solution but symbolic expressions characterizing entire classes of solutions. This thesis provides a positive answer, guided by the following thesis statement.

> *Symbolic reasoning can automatically derive correct control constraints.*

We encode the ability to reason *symbolically* about control systems such that computers become able to compute the expressions that characterize control solutions that are mathematically provably correct.

1.1 Introduction

Our representation of the constraints defining a *set of correct control solutions* is a *control envelope*. This thesis synthesizes control envelopes given the *shape* of the control system, i.e., what control decisions and environment dynamics are physically possible, and the desired control contract. Such information is necessary to make the synthesis question meaningful, and is generally already available to engineers in the form of client specifications. The specification of the control system may contain symbolic parameters representing any real number (e.g., train weight w while designing a train controller). The synthesized control envelope will be parametric in those symbols, making it adaptable. The generated control envelope is *verified*, that is, mathematically guaranteed to be correct for all of the (usually uncountably infinitely many) possible behaviors of the system.

Control envelopes are useful to synthesize controllers downstream. Suppose we start with a verified control envelope, representing a set of as many control solutions as possible that are all guaranteed to satisfy the mission-critical or safety-critical control objectives. Then non-formal methods, including machine learning, can be used to optimize within the control envelope for secondary or evolving objectives[1]. In this way control envelopes provide a way to disentangle design of the safety-critical aspects of controllers, which deserve expensive verification, from other concerns like performance and robustness which benefit from scalable non-formal optimization techniques.

We address the control envelope synthesis problem for the very general class of all problems where the shape of control can be expressed as a hybrid game (in differential game logic [121]). A hybrid game is a game with both discrete and continuous dynamics. It is typically used to model cyber-physical systems as a game played between a controller and an environment player. The controller player chooses control actions (for example, for a train, whether to accelerate), while the environment player resolves environmental non-determinism (such as how the curvature of the track evolves, which affects train motion dynamics). A control envelope corresponds to the nondeterministic policy of a controller player in a hybrid game against an adversarial environment player. The controller player wins if it maintains the desired control contract. Its policy indicates which control actions it can take in a given state to continue to have a way to win, thereby defining a space of safe control. Given a hybrid game specification defining what is physically possible, we therefore synthesize the controller player's control envelope. A special property that our work considers (compared to controller synthesis) is that synthesized control conditions should be as permissive as possible, allowing control actions unless they are genuinely unsafe. This way, there is as much room as possible for non-formal optimization within the envelope, allowing for maximal performance on secondary objectives.

To solve the problem in its full generality, we develop key technical innovations including (1) a new abstraction to represent control envelopes in a form that composes at the level of imperative program constructs (a *subvalue map*, or a symbolic analog to value tables from game theory [10]), (2) a hybrid games analog of Brzozowski derivatives [25] to step through games syntactically, and (3) a precondition calculus that defines correct control envelopes that never let the controller get stuck or lose the game. With these fundamental tools, it becomes possible to synthesize control envelopes through symbolic execution, except for subgames entailing unbounded sequences of gameplay (loops). For these, our precondition calculus, like most [40], *checks* the defining control conditions (loop invariants), which must be generated externally.

To *generate* such conditions about unbounded repetition games, we develop refinement and rewrite procedures. *Refinements* rewrite games in a way that makes them easier to symbolically execute, but harder for the controller player to play, which maintains soundness when the refined game is used in synthesis. *Rewrites* also make games easier to symbolically execute, but do not necessarily maintain soundness. Instead, any conditions they generate are checked retrospectively to recover soundness. To expand to new classes of problems, the user can simply add new refinement and rewrite rules to a library of rewrite heuristics that the synthesis framework uses. For a practically important class of problems (with *time-triggered control*, the common controller

---

[1] One way to use the control envelopes in combination with a machine learning (ML) based optimization step is to *shield* an ML controller agent's decisions. At training time, the agent is rewarded for staying within the envelope [131], and at runtime, fallbacks are enforced when it tries to deviate from the verified set of solutions [46, 50, 98]. This allows behavior to be optimized by gradient descent within the safe solutions. Control envelopes can also statically verify a model [152].

design pattern where the controller repeatedly polls, chooses an action, and then goes to sleep for some maximum time latency, and *action permanence*, a hybrid analog to idempotence), we identify a systematic set of refinements that yield a procedure to solve control envelope synthesis. The procedure is total computable (guaranteed to terminate) for an interesting subset of problems (where dynamics have solutions within the decidable fragment of real arithmetic).

We use large language models (LLMs) combined strategically with verification to soundly scale synthesis without having to manually identify new custom rewrite strategies for new problem classes. For cases when the engineer has a high-level idea for the control policy they want to include in the control envelope, our synthesis tool accepts informal guidelines as input for which it finds the detailed mathematical control conditions. Synthesis uses *automated verification* as a subroutine and *guarantees* that the generated control envelope is correct. To make automated verification scale, we build a pipeline to automate theorem proving by having LLMs interact with a theorem prover. This allows scaling of the overall synthesis pipeline.

Finally, having addressed the challenge of synthesizing control conditions given the shape of the control system, we explore the problem of automatically discovering the shape of the control system itself. In practice, engineers often have informal specifications of the control system as natural language design documents. The task of transforming these into the formal input to control envelope synthesis is the problem of *autoformalization*. An experimental exploration of applying LLMs to autoformalization for hybrid games shows both challenges and promise.

Overall, this thesis studies the synthesis of control envelopes for cyber-physical systems. It enables computers to automatically design control envelopes with formal guarantees by recasting control-theoretic reasoning into *syntactic* formulations. This involves introducing a compositional representation and calculus for control envelopes, identifying systematic refinement and rewrite-based approaches to make synthesis effective across different problem classes, and developing frameworks that combine LLMs strategically with formal reasoning to push the boundaries of automatic verification and synthesis. Together, the techniques of this thesis make it possible to automatically derive symbolic constraints defining correct control. This lays the foundation for controllers that are both formally guaranteed to be correct and support high-performance, learning-based optimization.

1.2 Structure

The thesis is structured as follows.

Chapter 2 provides background on the logic that the thesis uses. We use hybrid *games* written in *differential game logic* (dGL) [121], and hybrid *systems* represented in a subset of dGL called *differential dynamic logic* (dL) [116]. dGL uses an imperative programming language-like syntax. It has a deductive proof system, meaning that we can prove statements about the games by applying syntactic, compositional proof rules. The choice of using a symbolic, deductive logic allows us to produce control envelopes with nice properties: they are parametric in input symbols, making them adaptable, they cleanly handle infinite state spaces, and reason precisely about continuous dynamics without discretization. This choice is crucial to our technical approach later in the thesis: it allows the use of classic programming language techniques like symbolic execution and is amenable to the use of LLMs because of proximity to code.

Chapter 3 establishes a challenging case study that will serve as a target for automation. An objective of our work is to be able to synthesize control envelopes complex enough to apply to real-world systems. Designing symbolic control envelopes with high fidelity to real-world dynamics even manually is so challenging that case studies from the literature still usually make simplifications while modeling the environment and controller logically [92, 100, 101, 106, 126]. This leaves them a few steps away from being immediately applicable to real systems. So we first create, as a target for the synthesis tool, a verified control envelope that solves the practically important problem of train control [69] accounting for all the forces in a realistic train kinematics model [24]. The train control envelope defines what acceleration and braking actions are safe while ensuring that the train never goes past its end of motion authority, past which there is a risk of collision. This case study poses challenges common in other safety-critical embedded systems – complex dynamics with transcendental arithmetic, competing forces with subtle interaction, and effects whose exact magnitude is unknown at proof time.

Chapter 4 uses large language models to automate the process of proving theorems in dGL. This reduces the effort required to verify control envelopes, such as those from the train case study. Moreover, scalable theorem proving becomes a key component of scalable synthesis later in the thesis, because in order to scale, the synthesis procedure heuristically guesses control conditions and relies on verification as an oracle to check correctness. Next, the thesis shifts to the problem of synthesizing control conditions.

Chapter 5 solves control envelope synthesis for an interesting class of problems (with *time-triggered control*, the common controller design pattern where the controller repeatedly polls, chooses an action, and then goes to sleep for some maximum time latency, and *action permanence*, a hybrid analog to idempotence). To do so, it introduces the CESAR (Control Envelope Synthesis via Angelic Refinements) algorithm [70]. We implicitly characterize an optimal solution to control envelope synthesis using hybrid games, and then solve these games to obtain an explicit solution via symbolic execution aided by *game refinement* (transforming the game to be easier to execute symbolically while still preserving some properties).

Chapter 6 generalizes synthesis to all the problems where the shape of control can be characterized by a hybrid game. Given a hybrid game specification defining what is physically possible and what control properties are desired, we synthesize the controller player's control envelope (nondeterministic policy). To solve the problem in its full generality, this chapter introduces the concept of a *subvalue map*, a symbolic analog to value tables [10], to represent control envelopes such that their verification and synthesis composes at the level of imperative program constructs. The chapter provides a precondition calculus to verify/synthesize such control envelopes. This precondition calculus again makes it possible to use symbolic execution for synthesis, but still runs into subgames that are too complex to symbolically execute. Because of the extreme generality of the problem we tackle in this chapter, unlike in CESAR, no single set of systematic refinements is sufficient to simplify all possible subgames that we can encounter. So instead we use template-based *rewrite heuristics* that help hypothesize solutions when symbolic execution alone fails. The hypothesized solutions are checked to recover correctness.

Chapter 7 increases scalability by leveraging large language models. Instead of heuristics that need to be encoded case-by-case, we use large language models to guess control conditions, with retrospective verification per our developed theory to maintain soundness. The resulting scalability lets us synthesize a control envelope for the train case study. Finally, *Chapter 8* per-

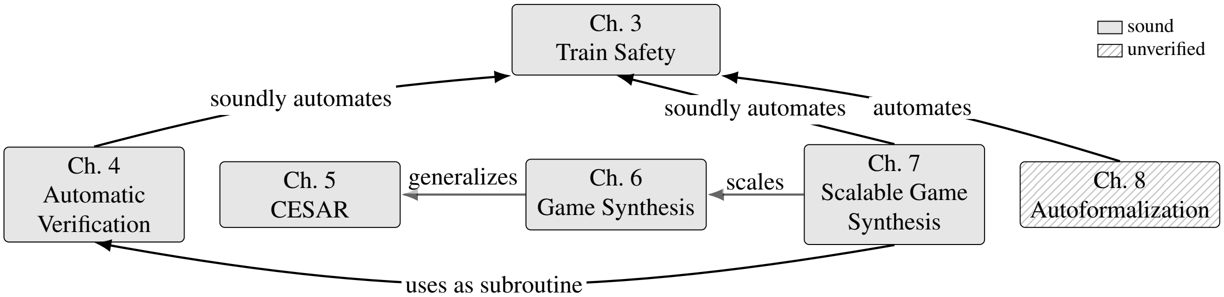


Figure 1.1: Chapter relationships

forms an initial exploration of using LLMs to obtain the game specification itself from informal descriptions. While the use of this step *can lose correctness because of the lack of formal guarantees on the correctness of the specification*, it can still be useful to streamline the control envelope design process if used in combination with manual review and checking the generated specification against real-world data when available.

The chapters are organized to be read in order. Fig. 1.1 shows a visualization of the relationship between them.

1.3 Related Work

We situate symbolic control envelopes within the larger scientific context. Individual chapters will have more specific and detailed related work sections pertinent to the specific contributions of each chapter. This thesis is based on some recently published works [68, 70, 72, 76]. Briefly, our work shows the following distinguishing features: 1. it is about permissive control *envelopes*, as opposed to deterministic controllers, 2. it uses *symbolic* reasoning (in contrast with numerical computation) to reason about infinitely many solutions at once, and 3. it reasons compositionally for rich *imperative program-like* control specifications.

Hybrid controllers. Instead of *controllers*, control *envelopes* strive to represent not one but as many safe control solutions of a system as possible, typically infinitely many. From a synthesis perspective, maximally permissive control *envelope* synthesis involves an extra universal quantifier compared to controller synthesis. Thus, controller synthesis techniques [12, 91, 148], for which popular approaches use temporal logic [5, 12, 164], games [108, 153], and CEGIS-like guidance by counterexamples [1, 36, 133, 143], do not immediately apply. Control *envelope* synthesis does not, for example, fit the CEGIS quantifier alternation pattern $\exists\forall$. Controller synthesis approaches with a *compositional logic approach* include refinement [16, 105], Hoare logic [95], *Differential Floyd-Hoare logic* [61], and constructive proof [20]. They soundly construct *deterministic* policies, and do not provide a direct representation of nondeterministic policies (control envelopes) from which to derive *sets* of permissible control actions.

Numeric safety shields computed by numerical methods [2, 48, 82] share the objective of characterizing the space of safe control. Our approach is *symbolic*, with the usual (dis)advantages compared to numerical methods. Numerical safety shields can handle dynamical systems that are hard to analyze symbolically, but typically do not support unbounded state spaces, and can lose rigorous formal safety guarantees when they discretize continuous systems. They do not produce shields that are *parametric* in the model's parameters, except by increasing dimensionality which does not scale well. Our choice of a symbolic logic approach, that traditionally requires reasoning through a few manual steps of applying deductive proof rules, also provides a simple path to automation by large language models, which can choose which proof rules to apply and benefit from soundness checking by existing interactive theorem provers.

Runtime monitoring can also be performed using a *symbolic* approach [46, 50, 98, 112, 138]. These runtime monitoring systems *use* control envelopes to monitor the safety of the system at runtime, resulting in *symbolic* shields. They raise an alarm upon deviation from the envelope, and can enforce safe fallback control. They bridge the gap between having a verified control envelope and enforcing it during actual system runs. We *synthesize* the control envelopes in the first place, providing the input to such monitoring systems. For example, in Chapter 3, our verified control envelope is applied by runtime monitoring system ModelPlex [98] to maintain train safety in our experiments.

Game theory and reinforcement learning [139, 145, 156] have the concept of nondeterministic player/agent policies which are similar in spirit to control envelopes. But these works operate at a different level of abstraction than our imperative program-like hybrid games. Also, our work is symbolic rather than numeric, and creates envelopes parametric in the symbolic parameters of the input game. Finally, our control envelope reasoning is syntactically formalized, suitable for computerized verification and formally justified synthesis.

Barrier functions and safe set approaches [4, 6, 33, 35, 97, 130, 144] also identify regions of safe and unsafe control. However, they typically model control by simple mathematical transitions in state space. We again solve for the higher level of abstraction of dL/dGL, with program structures such as loops, as well as the adversarial dynamics of games. Our envelopes indicate which actions, within the complex program structure, are safe, as opposed to just identifying safe regions in state space.

Hybrid automata provide an alternative formalism for modeling hybrid systems and games, in contrast with the logics used in this thesis which define game dynamics by imperative program-like constructs. Compared to synthesis approaches based on automata formalisms, we target specifications at a different level of abstraction where the game has imperative programming constructs like loops and branches, and is closer to code. Our different target formalism, with different composition patterns and proof techniques, requires a different technical approach to verification and synthesis. Beyond the difference in the level of abstraction of supported input specifications, automata based formalisms typically use model checking to establish correctness, instead of symbolic proofs, leading to different strengths and weaknesses. When problems related to control envelope synthesis are treated in those frameworks [13, 41, 63, 154, 166], different tradeoffs become necessary, such as remaining within linear dynamics and disallowing model specifications in terms of symbolic parameters to make efficient model checking possible. Our approach lets us use program analysis techniques like loop invariants and their continuous analog, differential invariants, which have traditionally been harder to automate but in return make it possible to solve classes of problems that are difficult to treat in model checking frameworks. For example, our synthesis approach gracefully handles symbolic parameters, unbounded state space and time horizons, and is capable of performing exact dynamical reasoning without the need for discretization. This is because our formalism gives us access to a proof calculus that is complete for reasoning symbolically about the very general class of dynamics whose solutions are Noetherian functions [128]. It allows our reasoning to be symbolic and parametric, even for highly complex dynamics. In a future where AI becomes increasingly proficient at symbolic reasoning, this would permit powerful generality without sacrificing automation, and therefore scalability.

Chapter 2

Background

This chapter recalls the logics used throughout this thesis: differential dynamic logic (dL) [117] for reasoning about hybrid systems, and its extension, differential game logic (dGL) [121], for reasoning about hybrid games. The *compositional*, *symbolic* representation of hybrid systems that these logics provide will be crucial to our approach in the rest of the thesis. The chapter starts with high-level, intuitive explanations of the logics, followed by formal details for the interested reader.

2.1 Differential Dynamic Logic

Differential dynamic logic (dL) is a logic with a deductive proof system for hybrid systems [116, 122]. We provide an introduction to the logic in this chapter, but a more comprehensive treatment can be found in the literature [123].

Differential dynamic logic extends first-order logic with the notion of *hybrid programs*. A hybrid program runs according to a binary relation between states, mapping start states to end states that a program could reach. The program constructs include assignment, for example, $x := e_1$, which instantaneously assigns expression e_1 to variable x. In the special case of nondeterministic assignment, $x := *$, the transition relation accounts for any possible real value being assigned to x.

The test operator, as in $?Q$, aborts the current run if formula Q is false. The continuous evolution operator, $\{x' = f(x) \,\&\, Q\}$ follows the ordinary differential equation (ODE) $x' = f(x)$ for some nondeterministic amount of time, with evolution domain constraint Q being true throughout the evolution. Sequential composition, $\alpha;\beta$, runs program α followed by program β, for example, a discrete train controller α followed by the train's ODE β. The nondeterministic choice operator $\alpha \cup \beta$ runs either program α or β, for example, either accelerate the train with α or brake with β. The loop operator α^* runs hybrid program α any nondeterministically chosen $n \geq 0$ times. It is useful, e.g., for running a train control loop indefinitely. Eq. (2.1) summarizes the syntax of hybrid programs.

$$\alpha, \beta \equiv \; x := e \mid \alpha;\beta \mid ?Q \mid \alpha^* \; \mid \alpha \cup \beta \mid x := * \mid \{x' = f(x) \& Q\} \tag{2.1}$$

To express safety properties about hybrid programs, we use the box modality $[\alpha]Q$, which is true in any state from which all runs of hybrid program α end in states in which the formula Q is true. For liveness properties, we use the diamond modality $\langle\alpha\rangle Q$, which is true in any state from which there is at least one run of hybrid program α that ends in a state in which Q is true. Besides these modalities, dL formulas are generated by the usual constructs of first-order logic, as summarized by Eq. (2.2), where $\sim \in \{<, \leq, =, \geq, >\}$ and θ_1, θ_2 are arithmetic expressions in $+, -, \cdot, /$ over the reals, x is a variable, α is a hybrid program.

$$\phi ::= \theta_1 \sim \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \forall x\, \phi \mid \exists x\, \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi \tag{2.2}$$

For proofs of dL formulas, we use dL inference rules [118, 123]. To provide a flavor of how these rules work, we discuss a few here: *loop* (loop), *differential invariant* (dI), and *differential cut* (dC). A *sequent* $\Gamma \vdash \Delta$ with a finite list of antecedent formulas Γ and succedent formulas Δ is short for $\bigwedge_{\phi\in\Gamma} \phi \rightarrow \bigvee_{\psi\in\Delta} \psi$. An *inference rule* has the form

$$\frac{\text{premise 1} \quad \cdots \quad \text{premise n}}{\text{conclusion}},$$

where the premises and conclusion are sequents. The rule can be read as saying that if all premises are valid, then the conclusion is valid. We next show examples of dL inference rules.

$$\text{(loop)} \quad \frac{\Gamma \vdash J \qquad J \vdash [\alpha]J \qquad J \vdash P}{\Gamma \vdash [\alpha^*]P}$$

$$\text{(dI)} \quad \frac{\Gamma \vdash e_1 \leq e_2 \quad Q \vdash [x' := f(x)]e_1' \leq e_2'}{\Gamma \vdash [x' = f(x) \,\&\, Q]e_1 \leq e_2}$$

$$\text{(dC)} \quad \frac{\Gamma \vdash [x' = f(x) \,\&\, Q \wedge C]P \quad \Gamma \vdash [x' = f(x) \,\&\, Q]C}{\Gamma \vdash [x' = f(x) \,\&\, Q]P}$$

As usual, the loop rule uses an invariant J that holds initially, inductively after each step, and implies the post condition that we seek to prove. A differential invariant preserves properties along the flow of a differential equation: if $e_1 \leq e_2$ initially and e_1 grows slower than e_2, so $e_1' \leq e_2'$ (where e_1' and e_2' are evaluated after substituting in the assignment $x' := f(x)$ from the differential equation), then it remains true that $e_1 \leq e_2$. A more general rule form and its explanation can be found in the literature [122, 123]. The idea behind a differential cut is that if formula C holds true at the end of every possible run of differential equation $x' = f(x)$, then C must hold true throughout its evolution. Differential cuts can be used to accumulate knowledge about a differential equation.

Fig. 2.1 shows an example of a dL proof. It proves that if initially position x is nonnegative and velocity v is nonnegative, then x remains nonnegative after following the ODE $x' = v$ for any duration of time. The theorem proved is at the bottom, $x \geq 0 \wedge v \geq 0 \vdash [x' = v]x \geq 0$. Applying proof rules creates a proof tree upwards. The proof uses the differential invariant rule (dI) to show that $x \geq 0$ throughout the evolution without actually having to solve the ODE by arguing instead that initially, $x \geq 0$, and based on its derivative x', it never decreases. This ability to establish properties of ODEs without solving them is a key advantage when dealing with more complex systems where closed-form solutions may not be available or lie outside the decidable fragment of arithmetic. The proof also uses the differential cut rule (dC) to transform the conclusion into the right form to apply dI. Upon reaching a sequent such as $x \geq 0 \vdash x \geq 0$

that is free of modalities, it is possible to use a solver such as Z3 [39] to discharge the proof obligation (denoted by rule $\mathbb{R}$).

$$\mathrm{dC}\,\dfrac{\mathrm{dI}\,\dfrac{\mathbb{R}\,\dfrac{*}{x \ge 0 \vdash x \ge 0} \quad [:=]\,\dfrac{\mathbb{R}\,\dfrac{*}{v \ge 0 \vdash v \ge 0}}{v \ge 0 \vdash [x' := v]x' \ge 0}}{x \ge 0 \land v \ge 0 \vdash [x' = v \,\&\, v \ge 0]x \ge 0} \qquad \dfrac{\text{truncated for brevity}}{x \ge 0 \land v \ge 0 \vdash [x' = v]v \ge 0}}{x \ge 0 \land v \ge 0 \vdash [x' = v]x \ge 0}$$

Figure 2.1: Example of a dL proof tree.

2.2 Differential Game Logic

Differential game logic (dGL) is a logic for two-player hybrid games that has a relatively complete axiomatization. A detailed explanation can be found elsewhere [121, 123], but we provide an overview. A dGL game is played by two adversarial players, canonically called Angel and Demon. It extends dL with two versions of each program construct where there is nondeterminism: one where Angel gets to choose how the nondeterminism is resolved, and one where Demon gets to choose how it is resolved.

An example of a game is $(v := 1 \cap v := -1)\,;\,\{x' = v\}$. In this game, first Demon chooses between setting velocity v to 1, or to -1. Then, Angel evolves position x as $x' = v$ for a duration of her choice. Differential game logic uses modalities to set win conditions for the players. For example, in the formula $[(v := 1 \cap v := -1)\,;\,\{x' = v\}]x \neq 0$, Demon wins the game when $x \neq 0$ at the end of the game and Angel wins otherwise. The overall formula represents the set of states from which Demon can win the game, which is $x \neq 0$ because when $x < 0$, Demon has the *winning strategy* to pick $v := -1$, so no matter how long Angel evolves $x' = v$, x remains negative. Likewise, when $x > 0$, Demon can pick $v := 1$. However, when $x = 0$, Angel has a winning strategy: to evolve $x' = v$ for zero time, so that x remains zero regardless of Demon's choice.

Table 2.1 summarizes dGL's program notation. As before, assignment $x := \theta$ instantly changes the value of variable x to the value of θ. Challenge $?\psi$ continues the game if ψ is satisfied in the current state, otherwise Angel loses immediately. In continuous evolution $x' = \theta \,\&\, \psi$ Angel follows the differential equation $x' = \theta$ for some duration of her choice, but loses immediately on violating ψ at any time. Sequential game $\alpha; \beta$ first plays α and when it terminates without a player having lost, continues with β. Choice $\alpha \cup \beta$ lets Angel choose whether to play α or β. For repetition α^*, Angel repeats α some number of times, choosing to continue or terminate after each round. The dual game α^d switches the roles of players. For example, in the game $?\psi^d$, Demon passes the challenge if the current state satisfies ψ, and otherwise loses immediately. Notice that in games restricted to the structures listed above but without α^d, all choices are resolved by Angel alone with no adversary, and hybrid games coincide with hybrid systems in differential dynamic logic (dL) [123][1].

---

[1] The dL fragment still does have special properties like completeness results and additional proof rules than dGL [123, 128].

In order to express properties about hybrid games, differential game logic formulas refer to the existence of winning strategies for objectives of the games (e.g., a controller has a winning strategy to achieve collision avoidance despite an adversarial environment). The set of dGL formulas is generated by the same grammar as dL, shown in Eq. (2.2), except that α is a hybrid game instead of a hybrid program.

Comparisons of arithmetic expressions, Boolean connectives, and quantifiers over the reals are as usual. The modal formula $\langle\alpha\rangle\phi$ expresses that player Angel has a winning strategy to reach a state satisfying ϕ in hybrid game α. Modal formula $[\alpha]\phi$ expresses the same for Demon. The fragment without modalities is first-order real arithmetic. Its fragment without quantifiers is called *propositional arithmetic* $\mathcal{P}_{\mathbb{R}}$. $[\alpha \cup \beta]\phi$ expresses that Demon has a winning strategy when Angel chooses between α and β to achieve ϕ, while $[\alpha \cap \beta]\phi$ expresses that Demon has a winning strategy to achieve ϕ when Demon chooses whether to play game α or β. Correspondingly, $\langle\alpha \cap \beta\rangle\phi$ expresses that Angel has a winning strategy to achieve ϕ when Demon has a choice between α and β. A formula ϕ is *valid*, written $\vDash \phi$, iff it is true in every state ω. States are functions assigning a real number to each variable. For instance, $\phi \rightarrow [\alpha]\psi$ is valid iff, from all initial states satisfying ϕ, Demon has a winning strategy in game α to achieve ψ.

2.3 Semantics

This section recalls the semantics of dL [117] and dGL [121]. It is not essential to understanding the thesis at a high level, but is a quick reference for readers interested in the formal details. We list the semantics after introducing the necessary notation.

ω denotes a state, which is a mapping from variables to real numbers. $\mathcal{S}$ denotes the set of all states. A state maps each variable to a real number. $\omega[\![x]\!]$ denotes the value of variable x in state ω. $X^{\complement}$ denotes the complement of set X. denotes the state that is the same as ω except that variable x maps to value r.

Table 2.1: Hybrid game operators for two-player hybrid systems

Game	**Effect**
$x := \theta$	assign value of term θ to variable x
$?\psi$	Angel passes challenge if formula ψ holds in current state, else loses immediately
$\big(x'_1 = \theta_1, \ldots, x'_n = \theta_n \,\&\, \psi\big)$	Angel evolves x_i along differential equation system $x'_i = \theta_i$ for choice of duration ≥ 0, loses immediately when violating ψ
$\alpha; \beta$	sequential game, first play hybrid game α, then hybrid game β
$\alpha \cup \beta$	Angel chooses to follow either hybrid game α or β
α^*	Angel repeats hybrid game α, choosing to stop or go after each α
α^d	dual game switches player roles between Angel and Demon
$\alpha \cap \beta$	demonic choice $(\alpha^d \cup \beta^d)^d$ gives choice between α and β to Demon
$\alpha^\times$	demonic repetition $((\alpha^d)^*)^d$ gives control of repetition to Demon

2.3.1 Differential Dynamic Logic

Definition 2.1 (dL formula semantics)**.** The *semantics of a* dL *formula* ϕ is the subset $\llbracket\phi\rrbracket \subseteq \mathcal{S}$ of states in which ϕ is true. It is defined inductively as follows

1. $\llbracket\theta_1 \sim \theta_2\rrbracket = \{\omega \in \mathcal{S} \,:\, \omega\llbracket\theta_1\rrbracket \sim \omega\llbracket\theta_2\rrbracket\}$ where $\sim \in \{<, \leq, =, \geq, >\}$
2. $\llbracket\neg\phi\rrbracket = (\llbracket\phi\rrbracket)^\complement$
3. $\llbracket\phi \wedge \psi\rrbracket = \llbracket\phi\rrbracket \cap \llbracket\psi\rrbracket$
4. $\llbracket\phi \vee \psi\rrbracket = \llbracket\phi\rrbracket \cup \llbracket\psi\rrbracket$
5. $\llbracket\phi \rightarrow \psi\rrbracket = (\llbracket\phi\rrbracket)^\complement \cup \llbracket\psi\rrbracket$
6. $\llbracket\exists x\, \phi\rrbracket = \{\omega \in \mathcal{S} \,:\, \omega_x^r \in \llbracket\phi\rrbracket$ for some $r \in \mathbb{R}\}$
7. $\llbracket\forall x\, \phi\rrbracket = \{\omega \in \mathcal{S} \,:\, \omega_x^r \in \llbracket\phi\rrbracket$ for all $r \in \mathbb{R}\}$
8. $\llbracket p(\theta_1, \ldots, \theta_k)\rrbracket = \{\omega \in \mathcal{S} \,:\, (\omega\llbracket\theta_1\rrbracket, \ldots, \omega\llbracket\theta_k\rrbracket) \in (p)\}$
9. $\llbracket\langle\alpha\rangle\phi\rrbracket = \{\omega \,:\, \nu \in \llbracket\phi\rrbracket$ for some state ν with $(\omega, \nu) \in \llbracket\alpha\rrbracket\}$
   where $\llbracket\alpha\rrbracket$ is the semantics of hybrid program α in Def. 2.2.
10. $\llbracket[\alpha]\phi\rrbracket = \{\omega \,:\, \nu \in \llbracket\phi\rrbracket$ for all states ν such that $(\omega, \nu) \in \llbracket\alpha\rrbracket\}$
    where $\llbracket\alpha\rrbracket$ is the semantics of hybrid program α in Def. 2.2.

Definition 2.2 (Semantics of hybrid programs)**.** A hybrid program α is interpreted as a binary relation on states $\llbracket\alpha\rrbracket \subseteq \mathcal{S} \times \mathcal{S}$, mapping start states to end states that a program could reach, defined inductively as follows [117]:

1. $\llbracket x := \theta\rrbracket = \{(\omega, \nu) \,:\, \nu = \omega_x^{\omega\llbracket\theta\rrbracket}\}$
2. $\llbracket x := *\rrbracket = \{(\omega, \nu) \,:\, \nu = \omega_x^r$ for some $r \in \mathbb{R}\}$
3. $?Q = \{(\omega, \omega) \,:\, \omega \in \llbracket Q\rrbracket\}$
4. $\llbracket\{x' = f(x) \,\&\, Q\}\rrbracket = \{(\omega, \nu) \,:\, \varphi(0) = \omega$ except at x' and $\varphi(r) = \nu$ for a solution $\varphi : [0, r] \rightarrow \mathcal{S}$ of any duration r satisfying $\varphi \models x' = f(x) \wedge Q\}$
5. $\llbracket\alpha \cup \beta\rrbracket = \llbracket\alpha\rrbracket \cup \llbracket\beta\rrbracket$
6. $\llbracket\alpha; \beta\rrbracket = \llbracket\alpha\rrbracket \circ \llbracket\beta\rrbracket = \{(\omega, \nu) \,:\, (\omega, \mu) \in \llbracket\alpha\rrbracket$ and $(\mu, \nu) \in \llbracket\beta\rrbracket\}$
7. $\llbracket\alpha^*\rrbracket = \llbracket\alpha\rrbracket^* = \bigcup_{n\in\mathbb{N}}\llbracket\alpha\rrbracket^n$, where $\alpha^{n+1} = \alpha^n \,;\, \alpha$ and $\alpha^0 \equiv ?\mathrm{true}$.

2.3.2 Differential Game Logic

Def. 2.3 defines the semantics of dGL formulas. It uses $\varsigma_\alpha(\llbracket\phi\rrbracket)$ to denote *Angel's winning region function* (applied for game α and Angel's winning condition $\llbracket\phi\rrbracket$) which will be defined later in Def. 2.4. Similarly, $\delta_\alpha(\llbracket\phi\rrbracket)$ shows *Demon's winning region function* which will be defined later in Def. 2.4.

Definition 2.3 (dGL semantics)**.** The *semantics of a* dGL *formula* ϕ is the subset $\llbracket\phi\rrbracket \subseteq \mathcal{S}$ of states in which ϕ is true. It is defined inductively as follows

1. $\llbracket\theta_1 \sim \theta_2\rrbracket = \{\omega \in \mathcal{S} \,:\, \omega\llbracket\theta_1\rrbracket \sim \omega\llbracket\theta_2\rrbracket\}$ where $\sim \in \{<, \leq, =, \geq, >\}$
2. $\llbracket\neg\phi\rrbracket = (\llbracket\phi\rrbracket)^\complement$
3. $\llbracket\phi \wedge \psi\rrbracket = \llbracket\phi\rrbracket \cap \llbracket\psi\rrbracket$
4. $\llbracket\phi \vee \psi\rrbracket = \llbracket\phi\rrbracket \cup \llbracket\psi\rrbracket$
5. $\llbracket\phi \rightarrow \psi\rrbracket = (\llbracket\phi\rrbracket)^\complement \cup \llbracket\psi\rrbracket$
6. $\llbracket\exists x\, \phi\rrbracket = \{\omega \in \mathcal{S} \,:\, \omega_x^r \in \llbracket\phi\rrbracket$ for some $r \in \mathbb{R}\}$

7. $[\![\forall x\, \phi]\!] = \{\omega \in \mathcal{S} \,:\, \omega_x^r \in [\![\phi]\!]$ for all $r \in \mathbb{R}\}$
8. $[\![p(\theta_1, \ldots, \theta_k)]\!] = \{\omega \in \mathcal{S} \,:\, (\omega[\![\theta_1]\!], \ldots, \omega[\![\theta_k]\!]) \in (p)\}$
9. $[\![\langle\alpha\rangle\phi]\!] = \varsigma_\alpha([\![\phi]\!])$
10. $[\![[\alpha]\phi]\!] = \delta_\alpha([\![\phi]\!])$

A dGL formula ϕ is *valid*, written $\vDash \phi$, iff it is true in all states, i.e. $[\![\phi]\!] = \mathcal{S}$.

Definition 2.4 (Semantics of hybrid games)**.** The *semantics of a hybrid game* α is a function $\varsigma_\alpha(\cdot)$ that, for each set of Angel's winning states $X \subseteq \mathcal{S}$, gives the *winning region*, i.e. the set of states $\varsigma_\alpha(X)$ from which Angel has a winning strategy to achieve X in α (whatever strategy Demon chooses). It is defined inductively as follows

1. $\varsigma_{x:=\theta}(X) = \{\omega \in \mathcal{S} \,:\, \omega_x^{\omega[\![\theta]\!]} \in X\}$
2. $\varsigma_{x'=f(x)\,\&\,Q}(X) = \{\varphi(0) \in \mathcal{S} \,:\, \varphi(r) \in X$ for some $r \in \mathbb{R}_{\geq 0}$ and (differentiable) $\varphi : [0, r] \to \mathcal{S}$ such that $\varphi(\zeta) \in [\![Q]\!]$ and $\frac{d\,\varphi(t)(x)}{dt}(\zeta) = \varphi(\zeta)[\![f(x)]\!]$ for all $0 \leq \zeta \leq r\}$
3. $\varsigma_{?Q}(X) = [\![Q]\!] \cap X$
4. $\varsigma_{\alpha\cup\beta}(X) = \varsigma_\alpha(X) \cup \varsigma_\beta(X)$
5. $\varsigma_{\alpha;\beta}(X) = \varsigma_\alpha(\varsigma_\beta(X))$
6. $\varsigma_{\alpha^*}(X) = \bigcap\{Z \subseteq \mathcal{S} \,:\, X \cup \varsigma_\alpha(Z) \subseteq Z\}$
7. $\varsigma_{\alpha^d}(X) = (\varsigma_\alpha(X^\complement))^\complement$

The *winning region* of Demon, i.e. the set of states $\delta_\alpha(X)$ from which Demon has a winning strategy to achieve X in α (whatever strategy Angel chooses) is defined inductively as follows

1. $\delta_{x:=\theta}(X) = \{\omega \in \mathcal{S} \,:\, \omega_x^{\omega[\![\theta]\!]} \in X\}$
2. $\delta_{x'=f(x)\,\&\,Q}(X) = \{\varphi(0) \in \mathcal{S} \,:\, \varphi(r) \in X$ for all $r \in \mathbb{R}_{\geq 0}$ and (differentiable) $\varphi : [0, r] \to \mathcal{S}$ such that $\varphi(\zeta) \in [\![Q]\!]$ and $\frac{d\,\varphi(t)(x)}{dt}(\zeta) = \varphi(\zeta)[\![\theta]\!]$ for all $0 \leq \zeta \leq r\}$
3. $\delta_{?Q}(X) = ([\![Q]\!])^\complement \cup X$
4. $\delta_{\alpha\cup\beta}(X) = \delta_\alpha(X) \cap \delta_\beta(X)$
5. $\delta_{\alpha;\beta}(X) = \delta_\alpha(\delta_\beta(X))$
6. $\delta_{\alpha^*}(X) = \bigcup\{Z \subseteq \mathcal{S} \,:\, Z \subseteq X \cap \delta_\alpha(Z)\}$
7. $\delta_{\alpha^d}(X) = (\delta_\alpha(X^\complement))^\complement$

The dGL axioms and proof rules [121] used in this thesis are summarized in Fig. 2.2.

$$
\begin{array}{rl}
([\cdot]) & [\alpha]P \leftrightarrow \neg\langle\alpha\rangle\neg P \\
(\langle{:=}\rangle) & \langle x := e\rangle p(x) \leftrightarrow p(e) \\
(\langle{}'\rangle) & \langle x' = f(x)\rangle p(x) \leftrightarrow \exists t{\geq}0\, \langle x := y(t)\rangle p(x) \qquad (y'(t) = f(y)) \\
(\langle ? \rangle) & \langle ?Q\rangle P \leftrightarrow Q \wedge P \\
(\langle\cup\rangle) & \langle \alpha \cup \beta\rangle P \leftrightarrow \langle\alpha\rangle P \vee \langle\beta\rangle P \\
(\langle ; \rangle) & \langle \alpha ; \beta\rangle P \leftrightarrow \langle\alpha\rangle\langle\beta\rangle P \\
(\langle{}^{*}\rangle) & \langle \alpha^{*}\rangle P \leftrightarrow P \vee \langle\alpha\rangle\langle\alpha^{*}\rangle P \\
(\langle{}^{d}\rangle) & \langle \alpha^{d}\rangle P \leftrightarrow \neg\langle\alpha\rangle\neg P \\
(\langle\cap\rangle) & \langle \alpha \cap \beta\rangle P \leftrightarrow \langle\alpha\rangle P \wedge \langle\beta\rangle P \\
(\langle{}^{\times}\rangle) & \langle \alpha^{\times}\rangle P \leftrightarrow P \wedge \langle\alpha\rangle\langle\alpha^{\times}\rangle P \\
([{:=}]) & [x := e]p(x) \leftrightarrow p(e) \\
([{}']) & [x' = f(x)]p(x) \leftrightarrow \forall t{\geq}0\, [x := y(t)]p(x) \qquad (y'(t) = f(y)) \\
([?]) & [?Q]P \leftrightarrow (Q \rightarrow P) \\
([\cup]) & [\alpha \cup \beta]P \leftrightarrow [\alpha]P \wedge [\beta]P \\
([;]) & [\alpha ; \beta]P \leftrightarrow [\alpha][\beta]P \\
([{}^{*}]) & [\alpha^{*}]P \leftrightarrow P \wedge [\alpha][\alpha^{*}]P \\
([{}^{d}]) & [\alpha^{d}]P \leftrightarrow \neg[\alpha]\neg P \\
([\cap]) & [\alpha \cap \beta]P \leftrightarrow [\alpha]P \vee [\beta]P \\
([{}^{\times}]) & [\alpha^{\times}]P \leftrightarrow P \vee [\alpha][\alpha^{\times}]P
\end{array}
$$

$$
\begin{array}{rcrc}
\textbf{(loop)} & \dfrac{\Gamma \vdash J, \Delta \quad J \vdash [\alpha]J \quad J \vdash P}{\Gamma \vdash [\alpha^{*}]P, \Delta} & \textbf{(ind)} & \dfrac{P \rightarrow [\alpha]P}{P \rightarrow [\alpha^{*}]P} \\
\textbf{(M)} & \dfrac{P \rightarrow Q}{\langle\alpha\rangle P \rightarrow \langle\alpha\rangle Q} & \textbf{(FP)} & \dfrac{P \vee \langle\alpha\rangle Q \rightarrow Q}{\langle\alpha^{*}\rangle P \rightarrow Q} \\
\textbf{(M}[\cdot]\textbf{)} & \dfrac{P \rightarrow Q}{[\alpha]P \rightarrow [\alpha]Q} & \textbf{(FP}^{\times}\textbf{)} & \dfrac{P \vee [\alpha]Q \rightarrow Q}{[\alpha^{\times}]P \rightarrow Q}
\end{array}
$$

Figure 2.2: dGL axiomatization and derived axioms and rules

Chapter 3

Verified Train Controllers

An objective of this thesis is to be able to synthesize control envelopes complex enough to provide controllers for real-world systems. However, designing and verifying symbolic control envelopes with high fidelity to real-world dynamics is so challenging that most existing work makes simplifications in their physical *model* of the world (e.g., ignoring resistance in train motion), or handles only a subset of situations (e.g. only handling single-lane highway driving for autonomous car control). To demonstrate a symbolic control envelope that is complex enough to apply to the real world, we create one for *train control* that uses a detailed physical model of train motion[1]. Simulated experiments demonstrate the utility of this train control envelope: in induced controllers, train safety is maintained, and by careful formal reasoning, inefficient, forced early braking is limited. This chapter is based on published work [69]. Subsequently, the goal of this thesis becomes making synthesis possible for control envelopes such as this one.

3.1 Introduction

This chapter uses control envelopes as a tool to design verified train controllers. This is a practically important, representative problem with challenges common in other safety-critical embedded systems: complex dynamics with transcendental arithmetic, competing forces with subtle interaction, and effects whose exact magnitude is unknown at proof time. Train controllers decide when to enforce braking to prevent movement authority[2]violation and collisions. They must account for all the competing influences that govern train motion. Uphill slopes decrease velocity, for example, which decreases resistance, which permits a more rapid increase in velocity, slope and curve effect, all while the train's brake force builds gradually until saturation as air pressure propagates along brake pipes. These complex interactions make it hard to design a safe and efficient train controller, and even harder to ensure it is always safe. This chapter designs and verifies train control envelopes (sets of controllers) for a realistic freight train kinematics model [23, 24], henceforth called the Federal Railroad Administration model (FRA model) after its appearance in influential technical reports published by the Federal Railroad Administration.

---

[1] Train control envelopes are practically important and already widely deployed. They are also called *train protection systems*, or in the context of the specific control objective of safe braking, *positive train control* systems.

[2] The movement authority of a train is the region of the track where it can safely be per the scheduling system.

In handling these dynamics, this chapter also identifies generalizable verified control envelope design techniques.

Existing studies of formally verified train motion [101, 126, 167] do not account for at least two effects amongst track grade, track curvature, resistance, and air brake propagation time, rendering their results inapplicable to most real-world scenarios. We surmount the challenges of verification against the full dynamics of the FRA model, in which these effects interact subtly with each other. Our verification results are significant, because these parameters influence the motion of the train in safety-critical and/or performance-critical ways. Neglecting track slope profile and the gradual propagation of air pressure braking, in particular, can render otherwise verifiably safe train controllers unsafe, since their influence may diminish the train's ability to decelerate, causing collisions. Our verification is valid for the realistic FRA model [23, 24].

Before verifying control envelope safety, we first design the control envelopes, balancing efficiency with provable safety. Conservative control envelopes are mathematically more simplistic, and easier to design and verify, but make railway operations inefficient, violating performance objectives. We start by presenting a conservative safe control envelope and then demonstrate how to iteratively make it more efficient by exploiting characteristics of the physical train dynamics for better but safe control.

Train control envelopes are assessed relative to a (changing) destination stopping point—called *end of movement authority*: overshoot of the end of movement authority is a safety violation, because that risks collision with other trains; efficiency is measured in terms of end of movement authority undershoot. We prove absence of end of movement authority overshoot when using our control envelopes in the FRA model by verification and demonstrate efficiency by simulation.

To keep our proofs as general and widely applicable as possible, we leverage nondeterministic controllers (control *envelopes*) and a paradigm of mathematical abstraction. Each control envelope is set-valued such that *all* of its control choices are simultaneously proved safe under all circumstances in the FRA model. The safety of these control envelopes implies the safety of *all* of their specializations [93], giving railroads significant freedom in how to adapt the verified control envelopes for their purposes. Control envelope verification follows a two-stage process: we first prove mathematical models of abstract train control motion, and then obtain proofs of the actual physical models of train control by *uniform substitution* [122] to replace the abstract function symbols of the mathematical models with physical terms specific to the FRA model or even specific railroads.

Crucially, our approach uses three different types of models: (i) the high-fidelity physics model describing the kinematic motion of trains along the track, (ii) our generalized mathematical abstractions of the physics model, and (iii) the simplified but *computable* approximations of motion models used by the respective train control envelopes. Our verification results prove that the safety of (i) derives from the safety of (ii) and that all control decisions following (iii) are safe in (ii).

Our proof is written in differential dynamic logic [116, 122], and performed using the hybrid systems theorem prover KeYmaera X [52]. We compare the efficiency with concrete control algorithms [23] for a number of train consists (arrangement of locomotives and cars) and scenarios [23]. Our verified control envelopes are imposed on unverified controllers via runtime enforcement checks using ModelPlex [99]. We illustrate their behavior in simulation. The proved models are permissive and only interfere in train control operation when acting otherwise risks movement authority violation.

Contributions: The primary contribution of this chapter is the formal verification results justifying the safety of train control in the FRA model. The secondary contribution is the design of new, safe train control envelopes. Technical contributions are the ideas of the formal proofs, and of substituting a specialized ground physics model into control envelopes designed with an abstract mathematical model. Finally, our simulations demonstrate the impact of verified train control envelopes in different terrain. Proofs are available at `https://doi.org/10.1184/R1/19542610` with ideas sketched inline.

3.2 Related Work

Due to their significance as safety-critical transportation systems, there have been many efforts to verify the safety of train control systems [47]. One approach is extensive simulation of train braking models [43, 151]. However, simulation can only show safety in a limited number of cases and is less appropriate when free acceleration is interspersed with braking. Similar limitations apply to test-based safety assurance of train control [27]. Our work uses differential dynamic logic (dL), a logic with a deductive proof system for hybrid systems [116, 122]. We write mathematical proofs that guarantee safety over the infinite state space in a model of physical motion.

In the realm of formal verification, there have been many studies of railway systems [14, 32, 45]. Discrete aspects of train control have been verified at industrial scale [44, 56]. Many studies [21, 31, 78, 79, 157] focus on scheduling trains to avoid route intersections. Train communication systems have been formally verified [45, 158]. Such studies are complementary to our work, which focuses on the motion of the train as it interacts with the environment. Results on the correctness of the motion of trains permit correct interaction with scheduling.

Studies of train motion have verified the European train control system with moving blocks [126], and the Chinese train control system [167], while ignoring the effect of resistance, air brakes, track grade and curvature. The FRA model has been verified while ignoring grade and curve [101]. Our work differs by accounting for *all* forces in the FRA model, creating controllers designed and verified against realistic physics.

3.3 FRA Model of Train Kinematics

This section introduces the FRA model, which provides the forces acting on a train [24, Eq. (1)]. After the net force on the train has been identified, Newton's second law, using train mass, determines the acceleration that the train experiences, which in turn determines change in velocity and change in position for train control design. The forces are

$$\sum F = -F_G - F_C - F_B + F_L - F_R = m_T a \tag{3.1}$$

where F_G denotes force due to track grade, F_C resistance due to track curvature, F_B force from the brakes, F_L force from the locomotive engine (tractive effort), F_R resistive forces, and m_T, train mass. Newton's second law, $\sum F = m_T a$, determines train acceleration a. Resistive force F_R follows the modified Davis equation [24, Eq. (18)]:

$$F_R = A_R w + B_R n + C_R w v + D_R v^2 \ \ ,$$

where A_R, B_R, C_R, D_R are experimentally determined positive constants whose numerical values, given a choice of units, can be found in other sources [23, 37, 62]. Further, n is the number of axles, and w is the weight of the train.

Grade and curve forces depend on the train position p on the track. Grade force [24, Eq. (19)] is proportional to train weight w and average track grade *grade*(p) underneath the train:

$$F_G = A_G w \cdot \textit{grade}(p) \ \ .$$

Similarly, curve force [24, Eq. (21)] is a function of average track curvature *curve*(p) along the train and train weight w:

$$F_C = A_C w \cdot \textit{curve}(p) \ \ .$$

where A_C and A_G are positive multiplicative constants. Braking force can be modeled as the minimum of two linear functions to capture the effect of air pressure brake force buildup [24, Fig. 17] and stabilization. Let F_{B0} be the force acting immediately on brake application, f_p, the slope with which brake force increases, t, the elapsed time, and $F_{B\text{max}}$, the force after air pressure in the air pressure brakes saturates.

$$F_B = min(F_{B0} + f_p t, F_{B\text{max}}) \ \ .$$

Brake enforcement and train protection algorithms [23, 24] approximately solve a differential equation derived from Eq. (3.1) to estimate the velocity and position of the train at future times.

3.4 Mathematical Model Abstraction

In order to maximize generality of the embedded software design, minimize verification effort, and simplify future proof maintenance, we present a mathematical abstraction of the FRA model [23, 24]. Concrete verified control envelopes and their safety proofs for the fully expanded model can be obtained automatically from the verified abstract model using uniform substitution [122].

Our abstract train kinematic model in (3.2) is an ODE in time. The rate of change of position is velocity, and the rate of change of velocity is acceleration. The variables and constants involved, along with their signs, when relevant, are (i) Train position p, (ii) velocity v, (iii) velocity and position-independent component of acceleration a_l ranging from immediate braking ability $-b_{\text{max}} < 0$ to maximum train engine acceleration $a_{\text{max}} > 0$, (iv) acceleration due to air brakes a_a in range $a_{b\text{max}} < 0$ to 0, (v) rate of change m_b of air brake acceleration, which is $m_p < 0$ when brakes are ramping up and 0 otherwise, (vi) map a_s from position to acceleration due to grade, (vii) map a_c from position to acceleration due to curvature, and (viii) velocity-dependent resistance a_r. In the chosen sign convention, resistive acceleration is negative.

$$p' = v, v' = a_l + a_a + a_s(p) + a_r(v) + a_c(p), a_b' = m_b \tag{3.2}$$

$$\text{with } a_l \in [-b_{\max}, a_{\max}], a_a = \max(a_b, a_{b\max}), m_b \in \{0, m_p\}$$

The Davis equation resistance $a_r(v) = -\frac{C_R w v + D_R v^2}{m_T}$ has the shape $a_r = a_1 v + a_2 v^2$ when $a_1 = -C_r g$ with gravity g summarizes the linear coefficient of velocity, and $a_2 = -\frac{D_R}{m_T}$ summarizes the quadratic coefficient. Grade and curvature are represented by unspecified but bounded functions a_s and a_c that map train positions to a numeric value for acceleration due to slope and average curvature, respectively. The quantity a_l summarizes locomotive tractive effort ($a_l \geq 0$) and train deceleration ($a_l < 0$) as commanded by the train controller, with adjustment for the velocity-independent resistance.

We later instantiate the proved abstract kinematic train model by dL's uniform substitution [122] to easily get proofs for specific physical train models such as the FRA model. Similarly, proofs for specific train configurations result from also substituting values for coefficients, or even for a specific train state when additionally substituting speed and position.

3.5 Model Structure

We develop a conservative train control envelope in dL based on the abstract train kinematic model Eq. (3.2) of Section 3.3, presenting it modularly and introducing conceptually important model components and functions along the way. We address the challenge of representing track grade and curve, which are unknown at proof time, using unspecified maps. In order to reason about them, we bound the maps with assumptions quantifying over all arguments. Our solution permits us to capture the full Eq. (3.2) without conservatively neglecting $a_s(p)$ and $a_c(p)$ when reasoning about it during verification. It generalizes to other embedded software that must reason about unspecified, bounded functions, such as noise or potential fields (e.g. electro-magnetic or gravitational effect).

We prove the control envelope safe: relative to Eq. (3.2), the control envelope will provably never permit the train's position to exceed the end of movement authority e, though it might be inefficient, braking unnecessarily early. Later, by revising modular components and functions to be more arithmetically sophisticated, Section 3.6 will retain provable safety but make the control envelope more efficient. Fig. 3.1 shows the relationship between the resulting control envelope models.

3.5.1 Model Description

Train control operates in a time control loop. The control has a latency of time $T > 0$—a controller has to wait at most this long before being able to change the throttle position. In practice, reaction time T is typically on the order of 1 second, but train controllers often keep decisions in effect for a 10 second period without revising them [37]. Every control cycle, the control envelope computes an overapproximation $\texttt{stopDist}(p, v, a_b)$ of the *stopping distance*, the distance that the train will travel before stopping if it were to accelerate during the current control cycle, but then brake continuously starting at the next control cycle, until it comes to a halt. If the distance left to the end of movement authority exceeds $\texttt{stopDist}(p, v, a_b)$, the control envelope continues free driving (with any acceleration or deceleration choice within the

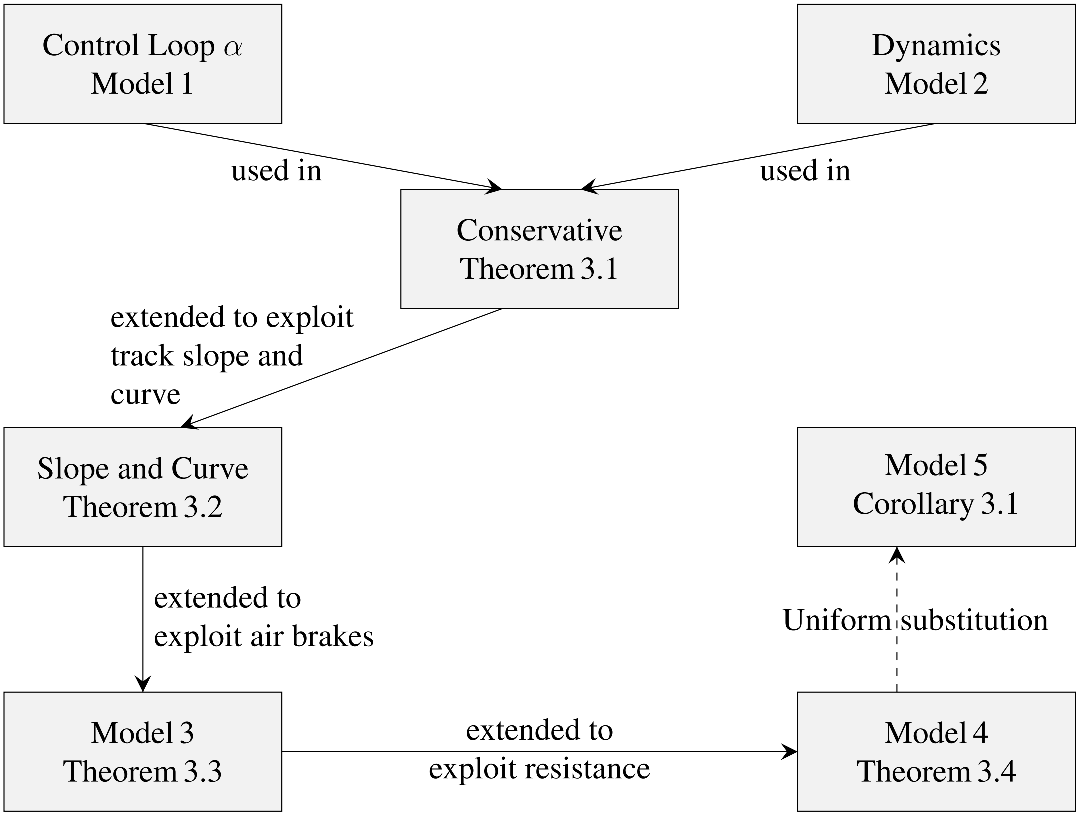


Figure 3.1: Relationship between train models in this chapter.

physical limits of the train), otherwise it brakes. This control cycle, $\alpha(\texttt{stopDist})$, is parametric in stopping distance `stopDist`, and expressed as a hybrid program in Model 1.

Model 1 Train control $\alpha(\texttt{stopDist})$.

```
Reset Timer  | $t := 0;$
Free Driving | $\big(\quad (?(e - p > \texttt{stopDist}(p, v, a_b));$
             |     $a_l := *;\ ? - b_{\max} \leq a_l < a_{\max}; a_b := 0;\ m_b := 0)$
Braking      |   $\cup\ a_l := -b_{\max};\ m_b := m_p\big)$
```

Train Dynamics Model 2 describes the physical motion of the train according to the abstract mathematical model of Eq. (3.2) augmented with a clock $t' = 1$ to switch back to control after at most time T. *We leave all constants symbolic and prove safety for all values*, so that individual railroads have the flexibility to instantiate them with the values that apply to their system and inherit our safety results. This generality in control envelope design comes at the cost of higher proof complexity, but compared to the alternative where we would initialize these constants with a conservative value, this allows for more efficient, tailored controllers and control envelopes adapted to the specific rail operation.

Model 2 Train dynamics.

$$\text{Dyn.}\left|\begin{array}{l} p' = v, a_b' = m_b, t' = 1, \\ v' = a_l + \max(a_b, a_{b\mathrm{max}}) + a_s(p) + a_r(v) + a_c(p) \;\&\; t \le T \land v \ge 0 \end{array}\right.$$

Stopping Distance In order to decide between free driving and braking, the control envelope computes the upper bound $\texttt{stopDist}(p, v, a_b)$ on the distance covered over one time period of acceleration and subsequently braking to a stop.

Thus, our models provide two distinct specifications of the distance that the train will take to stop. The first, indirect specification is through the differential equation in Model 2 that implicitly describes the physical motion of the train. The second, approximate specification is `stopDist` in Line 2 of Model 1, an explicit arithmetic expression that the control envelope can evaluate efficiently to make decisions at runtime.

Efficiency concerns demand that $\texttt{stopDist}(p, v, a_b)$ be as tight as possible; if the bound is too large, the control envelope would enforce braking unnecessarily. But verifiable safety requires $\texttt{stopDist}(p, v, a_b)$ to provably be an upper bound on the distance that the train covers (as determined by the dynamics). The tightest possible bound is the exact solution of the differential equation. However, even ignoring the effect of air brakes, the differential equation requires trigonometric solutions. Transcendental function arithmetic is undecidable [136]. To ensure mathematical provability[3], we develop polynomial approximations, which is a delicate design task because automated decision procedures for polynomial real arithmetic validity are computationally expensive [38, 160]. This constrains the complexity of the polynomial approximations that can be used as upper bounds. We therefore strike a balance between conflicting concerns: striving for efficiency while satisfying mathematical provability.

To illustrate this approach, we start with a simple conservative expression for `stopDist`. This expression is similar in complexity to previous work [101, 126], but is now proved safe for the full model including slope, curve friction, air brake propagation, and aerodynamic drag. We later improve on this approximation, focusing on one contributing factor at a time.

3.5.2 Stopping Distance: Conservative

This section constructs a first, conservative control envelope by instantiating control loop α with an expression for `stopDist`, and proving it safe. Referring back to the train dynamics in Eq. (3.2), we first need an upper bound for v. Integrating this bound via $p' = v$ computes a stopping distance upper bound.

[3] In KeYmaera X, even when manually simplifying differential and modal expressions, arithmetic subgoals are outsourced to arithmetic decision procedures, which are subject to limitations in proving real arithmetic.

The first impediment to obtaining a provable upper bound for v is that grade and curvature maps a_s and a_c are arbitrary functions, constrained only by upper and lower bounds, and bounded gradients. At runtime, the train knows their exact values as the control envelope is instantiated with maps for the railroad it runs on. However, these maps are unknown at proof time. And yet, the proof has to show safety of the train control ahead of time for *all* possible track maps in order to justify safety of the train control envelope. In order to obtain a provable upper bound on stopping distance, the proof is therefore based on the limited information that we do have about the maps: upper bounds on the potential values of a_s and a_c.

A naïve upper bound on a_s is the value of acceleration that the train experiences when it is on the steepest permissible downward slope, m_s. Our proof will show that the distance required to stop for any permissible grade map cannot exceed the distance computed with the steepest downward slope. It first shows that the true acceleration is bounded above by an acceleration that uses the highest permissible value of grade acceleration, then that actual velocity cannot exceed the velocity computed using the worst-case acceleration, and consequently, that traveled distance cannot exceed the stopping distance computed using the worst-case estimate of velocity.

Accounting for grade force is important. On a downward hill, for example, a train with a control envelope that ignores grade would roll forward even at the end of its movement authority which may cause accidents. In contrast, we can safely ignore curve resistance when approximating `stopDist`, since resistances shorten stopping distance (upper bound 0). These simplifications result in differential equation $v_2' = a_l + m_s + a_r(v)$, since $max(a_b, a_{b\text{max}}) = 0$ while the train is accelerating, where v_2 is the upper bound on v that we integrate to compute `stopDist`. However, the solution $v_2(t)$ is still transcendental:

$$\left(\sqrt{4(a_l + m_s)a_2 - a_1^2}\right) \cdot \frac{\tan\left(t\frac{\sqrt{4(a_l+m_s)a_2-a_1^2}}{2} + \tan^{-1}\left(\frac{a_1+2a_2v_0}{\sqrt{4(a_l+m_s)a_2-a_1^2}}\right)\right) - a_1}{2a_2}$$

The culprits are the linear and quadratic terms in velocity from the Davis equation. With another simplification of 0 as an upper bound for a_r (resistance always works against the train's motion), we derive a polynomial expression for `stopDist`:

$$\begin{aligned} v_3' = a_l + m_s \quad &\Rightarrow \quad v_3(t) = v_0 + (a_l + m_s) \cdot t \\ p_3' = v \quad &\Rightarrow \quad p_3(t) = p_0 + (v_0 + \frac{a_l + m_s}{2} \cdot t) \cdot t \end{aligned} \tag{3.3}$$

where v_0 and p_0 are the initial values of speed and position. The solutions provide a conservative stopping distance bound.

$$\mathtt{stopDist}_b(p, v, a_b) = \underbrace{vT + \frac{a_{\max} + m_s}{2} \cdot T^2}_{\text{distance while accelerating}} + \underbrace{\frac{(v + (a_{\max} + m_s)T)^2}{2(b_{\max} - m_s)}}_{\text{stopping from increased speed}} \tag{3.4}$$

The conservative stopping distance $\mathtt{stopDist}_b$ ignores its arguments p and a_b, but later refinements of `stopDist` functions also depend on p and a_b, which is why they are passed in.

The first two terms are the distance covered by the train in one control cycle of acceleration, while the third term is the distance that the train needs to stop should it start braking right after, assuming the worst-case value of 0 for a_a. This conservative distance is similar to what has been used in the literature [126], but has been adjusted to account for grade force with its worst-case accelerating or decelerating effects. Substituting Eq. (3.4) into control cycle α results in the dL hybrid program of the conservative control envelope: $\alpha(\texttt{stopDist}_b)$.

A physically important quantity is *braking distance*, the distance that the train will travel before coming to a stop if it were to start braking right now. We derive an upper bound, $\texttt{brakeDist}_b(v, a_b)$, which will be crucial to our proofs and the initial assumptions, and is improved upon later.

$$\texttt{brakeDist}_b(v, a_b) = \frac{v^2}{2(b_{\text{max}} - m_s)} \tag{3.5}$$

The last term of Eq. (3.4) is $\texttt{brakeDist}_b(v_3(T), 0)$ with v_3 according to Eq. (3.3) when $a_l = a_{\text{max}}$.

Initial assumptions $\texttt{init(brakeDist)}$ parametrized by $\texttt{brakeDist}$, and assumptions on air brakes $\texttt{initAirbrake}$ are required to prove the conservative control envelope safe. Assumptions about unspecified functions are represented by universal quantification over their input. This representation permits derivation of a formula about the unspecified function at any point of the train's evolution by substituting the quantified input with current values.

$$\begin{aligned}
\texttt{init(brakeDist)} &= a_{\text{max}} > 0 \land b_{\text{max}} > 0 \land a_1 < 0 \land a_2 < 0 \land e - p > \texttt{brakeDist}(v, 0) \\
&\quad \land b_{\text{max}} - m_s > 0 \land v \geq 0 \land m_s \geq 0 \land T > 0 \\
&\quad \land \forall x\,(|a_s(x)| \leq m_s) \land \forall x\,(a_c(x) \leq 0) \\
\texttt{initAirbrake} &= m_p < 0 \land a_{b\text{max}} < 0 \land m_b = 0 \land a_b = 0
\end{aligned} \tag{3.6}$$

Theorem 3.1 presents the dL formula representing the safety of the conservative control envelope.

Theorem 3.1. *The conservative braking control envelope guarantees that the train always remains within the end of the movement authority. The* dL *formula below is provable, where α is the control loop from Model 1 parameterized with Eq.* (3.4) *for* $\texttt{stopDist}$.

$$\textit{init}(\textit{brakeDist}_b) \land \textit{initAirbrake} \rightarrow [(\alpha(\textit{stopDist}_b); \textbf{\textit{Model 2}})^*]\, e - p > 0$$

Proof. The proof has been done in the theorem prover KeYmaera X, but we present its central ideas here. We use loop invariant $e - p \geq \texttt{brakeDist}_b(v, 0) \land a_b \leq 0 \land v \geq 0$ and split into cases for free driving and braking corresponding to the nondeterministic choice in Lines 2–4 of Model 1. On braking, the invariant is maintained because the derivative of the distance that the train will take to come to a stop does not exceed the derivative of the distance to the end of movement authority, i.e., $(\texttt{stopDist}_b)' \leq (e - p)'$, by dI. On free driving for a control period T, we first restate that the train maintains a distance to the end of movement authority of at least $\texttt{stopDist}$ adjusted for time t since the last control decision, i.e., $v(T - t) + \frac{a_{\text{max}} + m_s}{2}(T - t)^2 + \frac{(v + (a_{\text{max}} + m_s)(T - t))^2}{2(b_{\text{max}} - m_s)}$ by dC and dI. The required inequality relation between the derivatives, $-v \geq -v + v'(T - t) - (a_{\text{max}} + m_s)(T - t) + \frac{(v + (a_{\text{max}} + m_s)(T - t))(v' - (a_{\text{max}} + m_s))}{(b_{\text{max}} - m_s)}$, holds because $v' - (a_{\text{max}} + m_s) \leq 0$. $\square$

3.6 Safe Efficiency Improvements

We improve on the overapproximation for stopping distance in order to make our control envelope more efficient. The FRA model presents two challenges common in embedded controllers: it uses functions whose exact values are unknown at proof time (slope and curve maps), and has many interacting forces. Our techniques address these problems with two general principles: using quantified worst case bounds on unknown functions, and separation of dependencies. The first technique relies on the observation that the track changes are gradual and predictable (the rate of change of unknown functions is bounded). It drastically improves bounds on the effect of grade and curve over one time period of acceleration, after resolving *circular dependencies* between the variables of motion. The second technique improves the estimate `brakeDist` by accounting for air brake dynamics. It demonstrates a handling of triple integration, using *mode-splitting* to deal with the non-analytical change of behavior when brakes saturate. The third technique uses *Taylor polynomials* to capture the effect of resistance, which would otherwise lead to transcendental arithmetic. The track environment discussion in Section 3.6.1 and Section 3.6.2 will become assumptions of our models, while the calculations in Section 3.6.3 and Section 3.6.4 are machine-checked to be correct approximations as part of the KeYmaera X proofs.

3.6.1 Bound on Gradient

The train control envelope knows the current slope $a_s(p)$ and vertical curves of the track, which determine transitions from one track grade to another. This knowledge results in a bound h_{max} on the difference in grade per unit length [62, p.616–619]:

$$\left|\frac{\partial a_s(x)}{\partial x}\right| \leq h_{\text{max}} \quad \Rightarrow |a_s'(p)| \leq v h_{\text{max}}$$

The second inequality follows from the first using the chain rule and $p' = v$. After time T, a_s could have increased by no more than $u h_{\text{max}} T$, where u is some upper bound on v over the course of T time, which we derive later in this section.

3.6.2 Bound on Curve Resistance

Similar to bounding the gradient change, we compute an upper bound on the rate of change of curve resistance as a function of velocity using track geometry. Curve resistance depends on average curve $\mathit{curve}(p)$.[4] Assume that the tightest permissible curvature for the railroad corresponds to radius r. The greatest change in average curvature occurs when a train goes from a track with the greatest permissible curvature to a straight track (or vice versa). Over a small period of time dt, the portion of the train transitioning from greatest curvature to 0 curvature is

---

[4] In practice, degree of curvature is the angle subtended by a 100ft arc of the track, 100ft chord of the track, or 100.7ft arc in the US. In Europe, radius is generally used [62]. In our rate of change of curvature derivation, we will use the mathematical idealization for average degree of curvature: $\mathit{curve}(p) = \int_{p-l}^{p} \frac{1}{l r(x)} dx$ where l is the length of the train and $r(x)$ is the radius of curvature at track point x. This approximately relates to the US definition by a multiplicative factor, with a small error introduced by the granularity of the 100ft measuring arc. This multiplicative factor can be rolled into the multiplicative coefficient c_{cft}.

dv, where v is velocity. So the rate of change of *curve*(p) (taken in radians) with respect to time is $-\frac{v}{lr}$ where l is the length of the train. For a given train, a_c relates to *curve* with some constant multiplicative factor q. We use $a_c' = q\frac{v}{lr} = c_{cft}v$ with the constant factor $c_{cft} = \frac{q}{lr}$.

With this bound on the maximum rate of change of a_c, we now estimate the upper bound on curve resistance over time T, where p_0 is initial train position, to be

$$a_c(p_0) + uc_{cft}T$$

As before, u is an upper bound on velocity for duration T.

3.6.3 Tight Stopping Distance Approximation

The upper bounds $\overline{a}_s$ on gradient from Section 3.6.1 and $\overline{a}_c$ on curve resistance from Section 3.6.2 are summarized as:

$$\begin{aligned} a_s(p) &\leq \overline{a}_s(p_0) = \min(m_s, a_s(p_0) + uh_{\max}T) \\ a_c(p) &\leq \overline{a}_c(p_0) = \min(0, a_c(p_0) + uc_{cft}T) \end{aligned}$$

This enables us to improve our estimation of stopping distance:

$$v_4' = a_l + \overline{a}_s(p_0) + \overline{a}_c(p_0) \Rightarrow v_4(t) = v_0 + \big(a_l + \overline{a}_s(p_0) + \overline{a}_c(p_0)\big)T \tag{3.7}$$

Upper bound v_4 is tighter than v_3 of Eq. (3.3) and thus integrates to an improved stopping distance estimate. It depends (transitively through $\overline{a}_s$ and $\overline{a}_c$) on the unknown upper bound u on velocity, which we still need to estimate provably correctly.

Circular Dependencies: The upper bound on velocity, u, is undefined in expression (3.7) above. We cannot use the bound v_4 for u, since v_4 itself is phrased in terms of u. The problem is a circular dependency between a_s and v: the bound on slope acceleration a_s depends on speed v, while the upper bound on speed v, in turn, depends on slope acceleration a_s; likewise for a_c. Physically, this is because if the train is moving faster, we know less about the nature of the track—its curve and slope—after the passage of some time, as the train is farther from its previous position on the track. However, we need information about the grade curve in order to better estimate the velocity that the train is traveling at. In order to cut through these circular dependencies, we use the conservative estimations of these quantities from (3.3) as a base case to bootstrap incrementally finer computations, as presented below.

We first use the initial upper bounds m_s for a_s and 0 for a_c to get a conservative bound $v(t) \geq v_0 + (a_{\max} + m_s)t$, so that we can set $u = v_0 + (a_{\max} + m_s)T$. Since $(a_{\max} + m_s)$ is a positive upper bound on the train's acceleration, velocity could have increased no more than $(a_{\max} + m_s)T$. Hence u is indeed an upper bound on v through the T time interval. Substituting this u refines the gradient and curve resistance bounds.

$$\begin{aligned} \overline{a}_s(p_0) &= \min(m_s, a_s(p_0) + \overbrace{(v_0 + (a_{\max} + m_s)T)}^{u}\, h_{\max}T) \\ \overline{a}_c(p_0) &= \min(0, a_c(p_0) + \underbrace{(v_0 + (a_{\max} + m_s)T)}_{u}\, c_{cft}T) \end{aligned}$$

These expressions give the chosen definitions of $\overline{a}_s$ and $\overline{a}_c$ by replacing placeholder velocity bound u.

We could in principle further improve this upper bound on speed by using v_4 to obtain an even better bound on a_s and a_c, which could in turn yield an improved bound on v. However, extra levels of extrapolation increase proof cost and computation time when the control envelope is run. Each extra intermediate bound requires a constant number of extra proof steps, but provides diminishing efficiency gains in return. Intuitively, proof length is asymptotically linear in number of iterations because under optimal proof rule application ordering, each iteration induces one extra application of rule dC to introduce the intermediate bound into the proof tree, and rule dI to justify this intermediate bound.

The `stopDist` expression below uses v_4 with $u = v_0 + (a_{\max} + m_s)T$ to estimate stopping distance, which is sufficiently tight to make useful control decisions (see Section 3.8).

$$\mathtt{stopDist}_s(p, v, a_b) = vT + \left(\frac{a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)}{2}\right)T^2 - \frac{\Big(v + \big(a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)\big)T\Big)^2}{2(b_{\max} - m_s)} \tag{3.8}$$

We need further initial assumptions to prove the improved *slope-exploiting control envelope* $\alpha(\mathtt{stopDist}_s)$ safe. These assumptions represent the result of the track environment discussion of Section 3.6.1 and 3.6.2 used for the computer-checked proof.

$$\begin{aligned}\mathtt{init}_s(\mathtt{brakeDist}) \equiv \mathtt{init}(\mathtt{brakeDist}) \land a_{\max} - m_s + m_c > 0 \land h_{\max} \geq 0 \\ \land\, m_c \leq 0 \land c_{cft} \geq 0 \land \forall x\, (a_c(x) \geq m_c) \land \forall x' \forall x\, (|a_c'| \leq x' c_{cft}) \land \forall x' \forall x\, (|a_s'| \leq x' h_{\max})\end{aligned} \tag{3.9}$$

The technique of this section applies to *time-triggered controllers* (where a control loop runs with some known maximum latency and sensors measure current state every cycle) for physical systems with functions affecting the environment that are unknown except for bounds on their rate of change. The future value of the functions can be bounded in terms of their worst-case rate of change. Furthermore, these bounds can be used to compute bounds on other variables in the system, just as here a bound on velocity was used to bound slope and curve effect, which was again used to obtain a better bound on velocity. The situation arises frequently in practice: examples of unknown functions are a potential field, or a noise or error effect, which may have circular dependence with position.

Theorem 3.2. *The slope-estimating control envelope guarantees that the train stays within its movement authority. The* dL *formula below is provable, where α is the control loop from Model 1 parameterized with Eq.* (3.8) *for* `stopDist`.

$$\mathtt{init}_s(\mathtt{brakeDist}_b) \land \mathtt{initAirbrake} \rightarrow [(\alpha(\mathtt{stopDist}_s); \textit{Model 2})^*]\, e - p > 0 \tag{3.10}$$

Proof. By proof in KeYmaera X. The proof builds on the ideas from Theorem 3.1. We apply the loop rule with the same loop invariant as Theorem 3.1. If the train brakes, differential invariant rule dI again shows that the loop invariant holds throughout differential equation evolution.

If the train chooses to accelerate, then as before, we restate that the train maintains at least a distance of $\mathtt{stopDist}_s$ adjusted for time t since the last control decision. Unlike before, $\overline{a}_s$ instead of m_s accounts for worst-case a_s, and $\overline{a}_c$ instead of 0 accounts for worst-case a_c. Again, dI proves that this adjusted inequality remains true. In order to prove the required inequality on the derivatives, we first use differential cut rule dC to show $\overline{a}_s(p_0) \geq a_s(p)$ throughout the control cycle, and $\overline{a}_c(p_0) \geq a_c(p)$. There are two branches for each cut corresponding to how the $\min$ in $\overline{a}_s$ and $\overline{a}_c$ resolve. For example, using initial position and velocity p_0 and v_0, for $\overline{a}_s$, we need to show that $a_s(p) \leq m_s$, and $a_s(p) \leq a_s(p_0) + uh_{\text{max}}T$. While the former follows from the quantified assumption on $a_s(p)$, to prove the latter, we again adjust it for elapsed time t, to argue that $a_s(p) \leq a_s(p_0) + (v_0 + uh_{\text{max}}(T - t))$, proved using dI. The required derivative inequality follows from instantiating the quantified assumption bounding the rate of change of $(a_s(p))' \leq uh_{\text{max}}$ with the current position, and showing that u is an upper bound on v in the control loop. The argument for $\overline{a}_c$ is analogous. □

3.6.4 Effect of Air Pressure Brakes

The term $\texttt{brakeDist}_b$ conservatively neglects the *significant* effects of air brakes to avoid reasoning about their time dependence. This section derives a tighter $\texttt{brakeDist}_a$ that accounts completely for air brakes. It specifies a control envelope that simultaneously benefits from the slope and curve estimation of the previous section, and from air brake dynamics. The central insight required to prove the improved control envelope safe is how to compose reasoning about time-dependent air brake propagation and velocity-dependent slope and curve estimations from the previous section. We first show that $\texttt{brakeDist}_a$, the component of `stopDist` affected by air brakes, is the desired upper bound on distance to brake throughout the control loop. Then, holding $\texttt{brakeDist}_a$ constant, we perform the differential reasoning on slope and curve estimation described in the previous section. The two results together permit an overall proof of safety of the air brake-exploiting control envelope.

To derive the improved $\texttt{brakeDist}_a$, we first compute some intermediate functions from air brake dynamics. In Eq. (3.2) during brake rampup, with slope relaxed pessimistically to m_s, and curve and resistance to 0, $\max(a_b, a_{b\text{max}})$ evaluates to a_b, and m_b to m_p. The solution for v in the resulting differential equation $v' = b_{\text{max}} - m_s + a_b, a_b' = m_p$ is quadratic in t:

$$v = v_0 - (b_{\text{max}} - m_s + a_b)t + \frac{1}{2}m_p t^2 \tag{3.11}$$

Function t_b below computes the time the train takes to achieve full braking by subtracting current brake buildup a_b from maximal air braking $a_{b\text{max}}$, and dividing by the rate of increase in air brake force m_p. If the train comes to a stop before air brake saturation, it instead evaluates to the time until the train stops, as computed by solving Eq. (3.11) for $v = 0$.

$$t_b(v, a_b) = \min\Big((a_{b\text{max}} - a_b)/m_p, \frac{(b_{\text{max}} - m_s + a_b) - |(b_{\text{max}} - m_s + a_b)^2 - 2m_p v|}{m_p}\Big) \tag{3.12}$$

The distance that the train travels before either stopping or reaching maximum air brake effect is $\int_0^{t_b(v,a_b)} p\,dt = vt_b(v, a_b) + \frac{1}{2}(b_{\text{max}} - m_s + a_b)t_b(v, a_b)^2 + \frac{1}{6}(m_p)t_b(v, a_b)^3$. The velocity of the train after this period of buildup, by Eq. (3.11), is $v_f = v - (b_{\text{max}} - m_s + a_b)t_b(v, a_b) + \frac{1}{2}m_p t_b(v, a_b)^2$. So after the brakes finished ramping up, the distance traveled until the train comes to a halt is $\frac{v_f^2}{2(b_{\text{max}} - m_s - a_{b\text{max}})}$, using Newton's third equation of motion. If the train stops before finishing brake rampup, v_f evaluates to zero, as required. Adding the upper bounds on distance traveled before and after brake rampup results in $\texttt{brakeDist}_a(v, a_b)$ in Eq. (3.13), an upper bound on braking distance that accounts for the effect of air brakes. While this derivation for control envelope design is manual, its result will be verified by computer-checked proof in Theorem 3.3.

$$\begin{aligned}\texttt{brakeDist}_a(v, a_b) &= vt_b(v, a_b) - \frac{1}{2}(b_{\max} - m_s + a_b)t_b(v, a_b)^2 \\ &\quad + \frac{1}{6}(m_p)t_b(v, a_b)^3 + \frac{\left(v - (b_{\max} - m_s + a_b)t_b(v, a_b) + \frac{1}{2}m_p t_b(v, a_b)^2\right)^2}{2(b_{\max} - m_s - a_{b\max})} \\ \texttt{stopDist}_a(p, v, a_b) &= vT + \left(\frac{a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)}{2}\right)T^2 \\ &\quad + \texttt{brakeDist}_a\Big(v + \big(a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)\big)T, 0\Big)\end{aligned} \tag{3.13}$$

In order to prove the invariant that after a control cycle of braking, $e - p > \texttt{brakeDist}_a(v, a_a)$, we must reason about the three dynamically distinct cases: (i) when $\max(a_b, a_{b\max})$ is $a_{b\max}$; (ii) when it is a_b, and $t_b(v, a_b)$ evaluates to $(a_{b\max} - a_b)/m_p$; and (iii) when it is a_b but $t_b(v, a_b)$ evaluates to $\frac{(b_{\max} - m_s + a_b) - |(b_{\max} - m_s + a_b)^2 - 2m_p v|}{m_p}$. We split the model dynamics into these three evolution domains, branching between the three possibilities in a loop to transition between modes freely (Model 3). This does not affect the semantics of evolution: per train dynamics, mode transition happens at most once (from (ii) to (i) or (ii) to (iii)). Unrolling the loop to two iterations, one per mode, suffices to model train behavior. This split formulation simplifies syntactic proofs.

Model 3 Mode-split train dynamics.

$$\text{Dyn.}\;\left|\;\begin{array}{l}(\quad \{\text{Model 2} \mathrel{\&} a_{b\max} \geq a_b\} \\ \quad \cup \{\text{Model 2} \mathrel{\&} a_{b\max} \leq a_b \wedge (a_{b\max} - a_b)/m_p \geq t_p\} \\ \quad \cup \{\text{Model 2} \mathrel{\&} a_{b\max} \leq a_b \wedge (a_{b\max} - a_b)/m_p \leq t_p\})^*\end{array}\right.$$

$$\text{where } t_p = \frac{(b_{\max} - m_s + a_b) - |(b_{\max} - m_s + a_b)^2 - 2m_p v|}{m_p}$$

Theorem 3.3. *The air-brake-exploiting control envelope guarantees that the train stays within its movement authority. The* dL *formula below is provable, where α is the control loop from Model 1 parameterized with Eq.* (3.13) *for* `stopDist`.

$$\mathit{init}_s(\mathit{brakeDist}_a) \wedge \mathit{initAirbrake} \rightarrow [(\alpha(\mathit{stopDist}_a); \textit{Model 3})^*]\, e - p > 0$$

Proof. By proof in KeYmaera X. The high level idea is to use an outer loop invariant $e - p \geq \texttt{brakeDist}(v, a_a) \wedge a_b \leq 0 \wedge v \geq 0$ and again split into free driving and braking cases. On braking, we show that the outer loop invariant is maintained in each of the three dynamics modes using an inner loop invariant consisting of 4 formulas, the most important of which is $e - p > \texttt{brakeDist}(v, \max(a_b, a_{b\max}))$.

On free driving, we first show that $\texttt{brakeDist}_{a0} = \texttt{brakeDist}_a\Big(\big(v + \big(a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)\big)T\big), 0\Big)$ is truly an upper bound on all reachable $\texttt{brakeDist}(v, \max(a_b, a_{b\max}))$ values in the control cycle. Then, holding $\texttt{brakeDist}_{a0}$ constant, we follow a proof similar to Theorem 3.2 to show that, in every mode, the increase in p does not exceed the decrease in distance buffer $vT + \left(\frac{a_{\max}+\overline{a}_s(p)+\overline{a}_c(p)}{2}\right)T^2$. The free driving inner loop invariant consists of 14 formulas, most of which state that various upper bound expressions (such as on velocity, grade and curve) remain upper bounds over the course of the loop. By monotonicity, then, $e - p >$ distance buffer $+ \texttt{brakeDist}_{a0} \geq \texttt{brakeDist}(v, \max(a_b, a_{b\max}))$. $\square$

3.6.5 Exploiting Resistance

Exactly accounting for the quadratic dependence of resistance on velocity, as discussed in Section 3.5.2, leads to an undecidable, transcendental exact solution for stopping distance. The control envelope must instead use an approximation. Since polynomial arithmetic is decidable, Taylor polynomials are a natural way to obtain decidable approximations. This section applies Taylor approximation to the FRA model, identifying techniques generalizable to verified control for other embedded systems with transcendental dynamics.

The Davis equation implies $v' \geq (a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)) + a_1 v + a_2 v^2$, where slope and curve bounds $\overline{a}_s$ and $\overline{a}_c$ are from Section 3.6.3. This is a lemma for the dL proof of Theorem 3.4 justified as follows: consider two identical trains on tracks t_1 and t_2, starting with the same velocity. We want to bound the velocity v_1 of the train on t_1. Suppose t_2 is the track with worst case track and grade, and that a train on t_2 (the "ghost train", that we have constructed for the sake of our argument) always accelerates so that $v_2' = (a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)) + a_1 v + a_2 v^2$. On the other hand, on track t_1, the real train that we require a proof about only obeys the restriction $|a_s| < m_s$. If $v_2 - v_1$ is to become negative, it must cross the boundary where its value is 0. However, whenever $v_1 = v_2$, necessarily, $v_2' > v_1'$. This ghost train argument serves a purpose similar to the "circular dependencies" argument of Section 3.6.3: reasoning about mutually influencing factors one at a time. The ghost train permits us to represent and reason about a transcendental bound on velocity, v_2, derived using slope and curve estimates $\overline{a}_s$ and $\overline{a}_c$.

Given that $v' \geq (a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)) + a_1 v + a_2 v^2$, the first-order Taylor polynomial of this expression for velocity is $v_0 + \Big((a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)) + a_1 v_0 + a_2 v_0^2\Big)t$. Using this approximation at time T, with $a_l = a_{\max}$, as an upper bound for velocity after a time period of acceleration, we compute Eq. (3.14) for stopping distance that leverages resistance. While this derivation is manual, its result will be verified by computer-checked proof in Theorem 3.4.

$$\begin{aligned} \overline{v}' &= (a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)) + a_1 v + a_2 v^2 \\ \texttt{stopDist}_t(v) &= vT + \left(\frac{a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)}{2}\right)T^2 + \texttt{brakeDist}_a\Big(v + \big(\overline{v}'\big)T, 0\Big) \end{aligned} \tag{3.14}$$

Unlike previous stopping distance estimates, this expression is *not* always an upper bound. It uses resistance for the original velocity v_0, which is only a conservative bound when resistance is low enough to permit acceleration. This condition is captured by predicate `vbound` in Eq. (3.15).

$$\texttt{vbound}(v) \equiv (a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)) + a_1 v + a_2 v^2 \geq 0 \tag{3.15}$$

For the Taylor approximation control envelope (Model 4), we define the $\mathtt{stopDist}_{tp}$ predicate (3.16), that unlike previous expressions for stopping distance, returns a truth value. It uses previous definitions $\mathtt{stopDist}_t$ from Eq. (3.14), $\mathtt{vbound}$ (3.15) to determine when $\mathtt{stopDist}_t$ is applicable, and $\mathtt{stopDist}_s$ from Eq. (3.8) is used as a fallback.

$$\begin{aligned}\mathtt{stopDist}_{tp}(p, v, a_b) \equiv e - p > \mathtt{stopDist}_s(p, v, a_b) \lor \\ \Big(\mathtt{vbound}(v) \land e - p > \mathtt{stopDist}_t(v)\Big)\end{aligned} \tag{3.16}$$

Higher order Taylor polynomials permit analogous reasoning. Theorem 3.4 expresses that the *Taylor polynomial control envelope* in Model 4 is safe.

Model 4 Taylor polynomial control envelope and dynamics.

```
Reset Timer  | $t := 0;$
Free driving | $\big((?\mathtt{stopDist}_{tp}(p, v, 0); a_l := *;\ ? - b_{\max} \le a_l < a_{\max}; a_b := 0;\ m_b := 0)$
Braking      |   $\cup\ a_l := -b_{\max}; m_b := m_p\big);$
Dynamics     | Model 3
```

Theorem 3.4. *The Taylor polynomial control envelope guarantees that the train stays within its movement authority. The* dL *formula below is provable.*

$$\mathtt{init}_s(\mathtt{brakeDist}_a) \land \mathtt{initAirbrake} \to [(\textit{Model 4})^*]\, e - p > 0$$

Proof. By proof in KeYmaera X. We start by showing that the loop invariant from Theorem 3.3 is maintained (using dL rule loop). The case where the train is braking proceeds similar to Theorem 3.3. When the train is accelerating, we need to show that the control envelope has insisted on a sufficient distance margin (stopping distance), so that even after a time period, the train has enough space to stop. As the Taylor polynomial computes the stopping distance, we must prove that it actually is an upper bound. We use a monotonicity argument by introducing an auxiliary variable that represents a "ghost" train, perpetually traveling down worst possible slope $\overline{a}_s$ and curve $\overline{a}_c$. This isolates slope and curve from the effect of resistance, breaking interdependence. We derive the Taylor polynomial result on the ghost train, show that it goes no slower than the real train, and that consequently the Taylor polynomial result must hold for the real train. Other elements of the proof remain similar to Theorem 3.1. □

3.7 Kinematic Train Model Proofs

A proof for the FRA model [24] derives from the proof for our abstract mathematical models, e.g. Theorem 3.4, by uniform substitution [122], which replaces abstract function symbols with specific terms using the correspondence in Section 3.4. Model 5 lists the model resulting from substitution.

Model 5 Train control envelope in kinematic motion model.

```
Reset Timer   1  $t := 0;$
Free Driving  2  $\big(\ \ ?\mathtt{stopDist}_{tp}(p, v, 0); a_l := *;\ ?{-b_{\max}} \le a_l < a_{\max}; a_b := 0;\ m_b := 0$
Braking       3  $\ \ \cup\ a_l := -b_{\max};\ m_b := m_p\big);$
              4  $(\ \ \ \{\text{dyn} \ \&\ a_{b\max} \ge a_b\}$
Mode Split    5  $\ \ \cup \{\text{dyn} \ \&\ a_{b\max} \le a_b \land (a_{b\max} - a_b)/m_p \ge t_p\}$
              6  $\ \ \cup \{\text{dyn} \ \&\ a_{b\max} \le a_b \land (a_{b\max} - a_b)/m_p \le t_p\})^*$
```

$$\text{with dyn} \equiv \{p' = v, a_b' = m_b, t' = 1, v' = a_l + \max(a_b, a_{b\max}) - \frac{A_G w \cdot \textit{grade}(p)}{m_T} - \frac{A_C w \cdot \textit{curve}(p)}{m_T} - \frac{C_R w v + D_R v^2}{m_T} \ \&\ t \le T \land v \ge 0\}$$

Corollary 3.1 (Kinematic train model is safe)**.** *The train control envelope for the FRA model never overshoots the end of movement authority, i.e., the following formula is provable with* $b_{max} = \frac{F_B}{m_T} + \frac{A_R w + B_R n}{m_T}$ *and* $a_{max} = \frac{F_L}{m_T} - \frac{A_R w + B_R n}{m_T}$*:*

$$\mathtt{init}_s(\mathtt{brakeDist}_a) \land \mathtt{initAirbrake} \rightarrow [(\textit{Model 5})^*]\, e - p > 0 \ .$$

Proof. By uniform substitution from Theorem 3.4, using the substitutions σ below:

$$\sigma = \begin{cases} a_{\max} \mapsto \frac{F_L}{m_T} - \frac{A_R w + B_R n}{m_T} & b_{\max} \mapsto \frac{F_B}{m_T} + \frac{A_R w + B_R n}{m_T} \\ a_s(p) \mapsto -\frac{A_G w \cdot \textit{grade}(p)}{m_T} & a_c(p) \mapsto -\frac{A_C w \cdot \textit{curve}(p)}{m_T} \\ a_r(v) \mapsto -\frac{C_R w v + D_R v^2}{m_T} & \end{cases}$$

□

3.8 Evaluation

For validation, we use ModelPlex [99] to derive a controller monitor from Model 5 that measures the safety margin in decisions of previous brake enforcement envelopes [23, 24] and our verified control envelopes. That way, we measure if, and how well, our verified train control envelopes and existing control envelopes agree in order to assess the safety of those existing systems and the efficiency of our model. Existing brake enforcement envelopes brake to a full stop once engaged.

The ModelPlex monitor computes a robustness measure indicating how close a decision is to losing the safety proof. When the robustness measure is positive, the decision is guaranteed to remain provably safe so that the system enjoys the safety proof of the verified model. When it is negative, emergency brakes should be applied for safety reasons. The ModelPlex controller monitor follows the structure of the verified model when it computes robustness. For example, a monitor for Line 2 of Model 5 describes its effect with the formula $\mathtt{stopDist}_{tp}(p, v, 0) \land$

Table 3.1: FRA train parameter instantiation example.

Param.	Value	Unit	Description
A_R	0.6	$^{\text{lbf}}/_{\text{ton}}$	(constant) weight coefficient
B_R	20	$^{\text{lbf}}/_{\text{axle}}$	(constant) train-size coefficient
C_R	0.01	$^{\text{lbf}}/_{\text{ton}\cdot\text{mph}}$	(linear) speed coefficient
D_R	0.07	$^{\text{lbf}}/_{\text{mph}^2}$	(quadratic) aerodynamic coefficient
	0.294		(for loaded cars)
A_G	20	$^{\text{lbf}}/_{\text{ton}}$	(constant) weight coefficient

$-b_{\max} \leq a_l < a_{\max}$ which translates to the robustness measure

$$\min\Big(\max\big(\min(e - p - \texttt{stopDist}_t(v), (a_{\max} + \overline{a}_s(p) + \overline{a}_c(p)) + a_1 v + a_2 v^2),$$
$$e - p - \texttt{stopDist}_a(p, v, 0)\big), a_l + b_{\max},\ a_{\max} - a_l\Big)\ .$$

The most important elements of the full Model 5 monitor are:

- in free driving (when $\texttt{stopDist}_{tp}$ is satisfied) it combines remaining position margin (the larger of Taylor margin $e - p - \texttt{stopDist}_t$ when $\texttt{vbound}(v)$, or fallback margin $e - p - \texttt{stopDist}_a$) with acceleration choice robustness ($\min(a_l + b_{\max}, a_{\max} - a_l)$ from control decision $a := *; ?\ -b_{\max} \leq a_l < a_{\max}$) and speed robustness ($v$ from evolution domain $\ldots \,\&\, v \geq 0$);
- during braking, which is always allowed, it measures speed robustness (v per evolution domain constraint).

Because ModelPlex's robustness measure combines multiple quantities of incompatible units, there is no direct interpretation of its magnitude, but only of its sign.

For validation, we implement the train model of Eq. (3.2) in Python by numerical integration, instantiating the model parameters per FRA model [23, 24]. These parameter values are estimated from train test runs and standards [23] and require careful consideration of their units (Table 3.1).

Our evaluation compares start braking and stopping points of trains, highlighting braking performance in terms of overshoot (safety risk) and undershoot (performance objective of a maximum undershoot of 1000ft [24]) of the end of movement authority. We follow [23, p. 47] and implement the baseline control envelopes using numeric forward Euler integration to simulate the model in order to determine the stopping distance. Our verified control envelopes neither use numeric integration nor include the dynamic model, but instead decide based on the stopping distance overapproximation $\texttt{stopDist}_{tp}$ of Eq. (3.16).

The most interesting train behavior arises from the subtle interplay between air pressure propagation, aerodynamic/rolling resistance, and acceleration or deceleration due to slope. It peaks on crests that change gradient from uphill to downhill and in troughs that change gradient from downhill to uphill. When calculating stopping distance, numerical integration in the baseline enforcement algorithms discretizes train speed and position to calculate forces, which overestimates resistance while simultaneously underestimating available brake force. Acceleration/deceleration due to slope is even more subtle as it depends on the position of the train along the slope (e.g., on a crest changing from uphill to downhill, deceleration on the uphill segment is overestimated until the train passes the top, afterwards acceleration is underestimated). These effects do *not* balance out and thus make numerical integration errors unreliable and hard to predict. Moreover, changing the integration step size shifts how distance estimates are biased towards undershoot or overshoot (e.g., in typical configurations, brake rampup is the dominating influence on stopping distance, and so larger integration step sizes bias towards undershoot). As a result, for any given configuration of numerical integration in enforcement algorithms, we can construct scenarios where the numerical integration underestimates stopping distance and train enforcement exhibits unsafe behavior. Our formal models and proofs design *provably correct* stopping distance overapproximations instead of using numerical integration and are, therefore, not subject to these intricate safety tradeoffs.

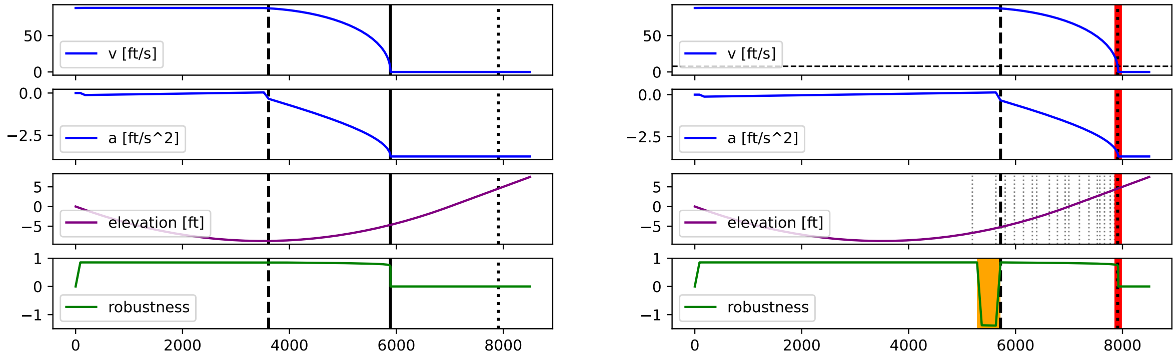
3.8.1 Stopping Behavior in Troughs

Our first validation in Fig. 3.2 follows [24, Fig. 10] with a train configuration of a medium (75 cars), loaded (car weight 286klbf), mixed freight train traveling at initial speed 60mph in a trough, with train starting position at 0ft. The trough is configured with uniform rate of change from 0.5% downhill to -0.5% uphill between positions 0ft and 7000ft. Locomotive tractive effort and locomotive braking compensate for speed loss and gain due to gradient. Air brakes engage only for the full braking maneuver when attempting not to exceed the movement authority. We configure our monitors and controllers to match the trough maximum uphill/downhill gradient of 0.5% for slope estimation. A sweep of movement authority endpoints in the range 4000ft–8000ft with 10ft steps identifies challenging configurations. The base enforcement algorithm [24] underestimates stopping distance for 19 (of the 400) endpoints. Fig. 3.2 compares the base enforcement algorithm to our verified control envelope on one of these challenging points at 7910ft, past the maximum uphill slope of the trough. The base enforcement algorithm in Fig. 3.2a uses numeric integration to determine the stopping distance, and adds a generous constant fudge factor plus speed-dependent safety offset that results in the train initiating braking at 3609ft. The maneuver finishes at 5889ft, stopping 2021ft short of the end of movement authority. In Fig. 3.2c, our verified control envelope does *not* use any such offsets. It initiates braking much later at position 5368ft and stops at 7578ft (significantly closer to the movement authority endpoint, limiting undershoot to 332ft).

Fig. 3.2b, removes the safety offset from the base enforcement algorithm in an experiment illustrating the subtle interplay between forces and their safety consequences. Even though the uphill segment of the trough helps reduce stopping distance, numeric integration overestimates this effect and initiates braking too late, which our monitor detects at 5368ft. Should the train ignore the warning, it overshoots by 9ft (highlighted in red[5]) with a remaining speed of $7.8\frac{\text{ft}}{\text{s}}$ when passing the desired stopping point. Initiating fallback control $a_l = -b_{\text{max}}$ at the monitor violation would have kept the train from overshooting the movement authority.

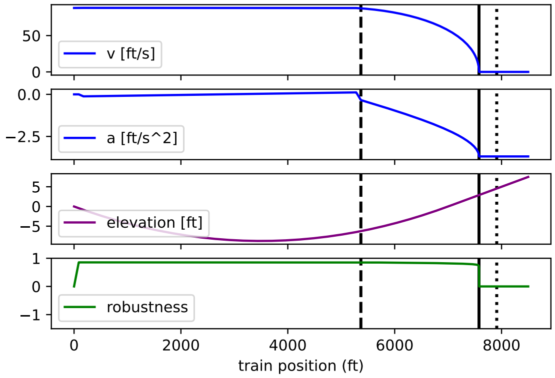
---

[5] The controller monitor subsequently no longer flags a violation, because the base enforcement algorithm then applies the correct decision of maximum braking (agreeing with the model). But it does so too late, as the earlier monitor warning was ignored.



(a) Base enforcement algorithm [24] initiates braking at 3609ft and stops at position 5889ft with an undershoot of 2021ft, which exceeds the 1000ft maximum undershoot performance objective [24].

(b) Base enforcement algorithm [24] without offset overshoots by 9ft ↯. Light-gray dotted lines mark other challenging movement authority endpoints in the range 4000ft–8000ft that would also be violated.



(c) Verified control envelope (Model 5) limits undershoot to 332ft.

Figure 3.2: Comparison of start braking (dashed vertical line) and stopping points (solid vertical lines) in a trough from 0.5% downhill at start to -0.5% uphill at 7000ft, with end of movement authority (dotted vertical line) at 7910ft.

3.8.2 Stopping Behavior on Crests

An assumption in the brake enforcement algorithms and thus an initial condition in our proof is that train locomotives are not underpowered: their tractive effort is enough to overcome maximum uphill slope and stay stopped on the maximum downhill slope. Fig. 3.3 illustrates how an underpowered, initially stopped train starts rolling downhill until the air brakes build enough deceleration to stop the train. This configuration, violating initial conditions, is unsafe to start in the first place.

Underpowered locomotives are especially challenging on a crest where (full) tractive effort is needed to limit the speed loss on the uphill slope and regain desired speed on the downhill segment, but air brakes are needed to stay stopped. Fig. 3.4 compares the behavior of the base enforcement algorithms and our verified algorithm with underpowered locomotives on a crest with movement authority endpoint at 7920ft. In this experiment, we configure our monitors and

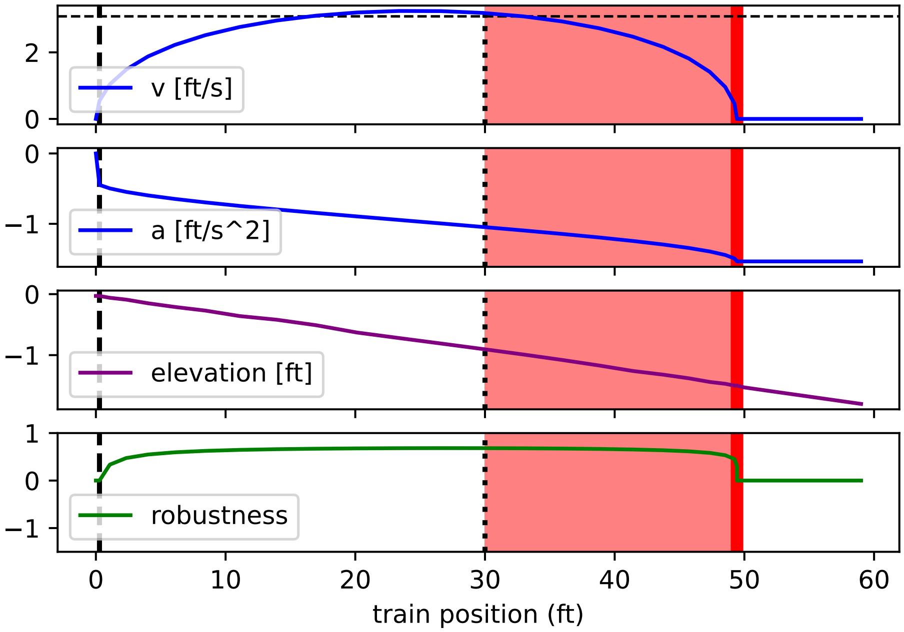


Figure 3.3: Initial conditions of proof (Corollary 3.1) not satisfied ↯: an underpowered train on a 3% downhill slope needs air brakes to stay stopped; the stopping point is closer than the rolling distance during brake rampup.

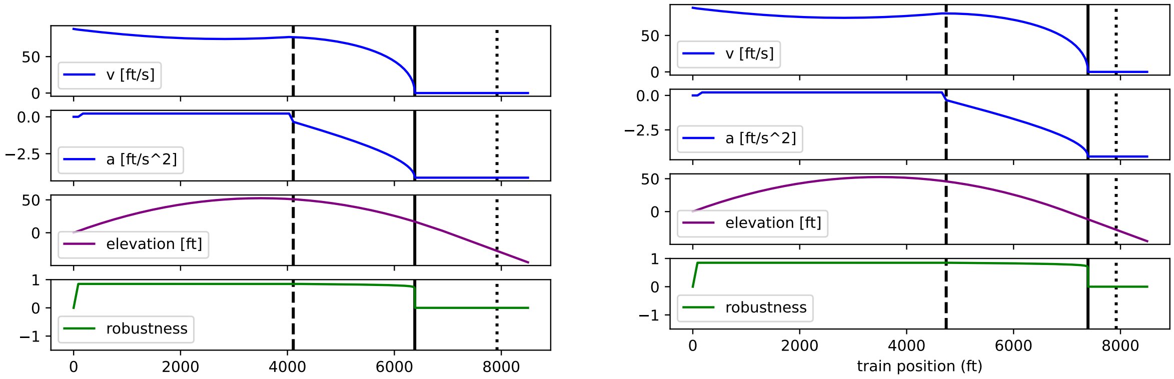
controllers for a maximum uphill/downhill gradient of 3% for slope estimation.

Numeric integration underestimates stopping distance in several configurations, whereas our verified control envelope correctly identifies the need to engage air brakes in time while simultaneously avoiding the inefficiencies of fudge factors.

3.9 Conclusion

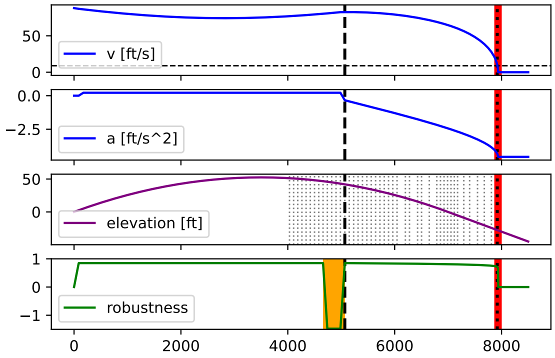
We created formally verified train control envelopes that account for the FRA model with all its competing influences of track grade, curve resistance, air brakes and Davis resistance. Control envelope efficiency is improved using techniques that generalize to resolve challenges in safety-critical embedded software design. Validation in simulation shows significant improvement in undershoot over conservative control envelopes that use safety offsets, and improved safety compared to control envelopes without safety offsets. VeriPhy[18], a verified pipeline that automatically converts verified dL models to verified executables can bridge the gap between the real arithmetic verification of the chapter and floating points used by software.

This chapter demonstrates that control envelopes can be designed to account for the complexity of the real world. It shows both the benefits and the challenges of doing so. The thesis next develops automation for the design process of control envelopes such as those discussed in this chapter.



(a) Base enforcement algorithm [24] undershoots by 1539ft.

(b) Verified control envelope (Model 5) limits undershoot to 528ft.



(c) Base enforcement algorithm [24] without offset overshoots by 14ft ↯. Light-gray dotted lines mark other movement authority endpoints in the range 4000ft–8000ft that the algorithm would also overshoot.

Figure 3.4: Initial conditions of proof (Corollary 3.1) not satisfied ↯: an underpowered train on a crest from 3% uphill to −3% downhill slows down uphill despite full tractive effort and regains speed on the downhill segment, but needs air brakes to stay stopped; in this scenario, the movement authority ends past the rolling distance during brake rampup.

Chapter 4

Automatic Verification

In the previous chapter, we developed train control envelopes and then verified them. The verification process was time-consuming and complex, and a significant contributor to the difficulty of creating the verified control envelope. In this chapter, we explore the use of large language models (LLMs) in combination with the theorem prover KeYmaera X to *automate* the verification of control envelopes. This will be a crucial building block for scalable control envelope synthesis later in this thesis.

4.1 Introduction

Extensive research has gone into developing formal methods for hybrid systems and games, with different approaches making different tradeoffs. Some tools gain impressive scalability by limiting the systems that they handle to mathematically simpler dynamics such as piecewise constant [26] or affine [84, 134]. Other tools very effectively handle nonlinear dynamics over short time horizons [3, 17] using numerical techniques, but lose precision of prediction over longer time horizons, struggling when time is unbounded. The general trend is that scalability is gained at the cost of expressiveness and precision. On the most expressive end are deductive, symbolic techniques in high-level logics [52, 120, 140, 159, 163, 165]. They accept nonlinear dynamics interacting with discrete control in rich, program-like structures, permit reasoning in terms of *symbolic parameters* that make theorems adaptable and reusable, and cleanly handle infinite state space as well as unbounded domains including unbounded time horizons, all with precise reasoning free of discretization or approximation. However, they have one big disadvantage: despite efforts towards automation [59, 142], they generally require effort-intensive interactive theorem proving by an expert, which becomes a bottleneck for the complexity of systems handled and broader adoption[1]. In particular, this thesis uses high level symbolic logics (dL, dGL) to express and reason about control envelopes, and therefore must contend with the same bottleneck.

LLMs have recently shown promise in automating (and therefore scaling) interactive theorem

---

[1] Hybrid systems are fundamentally undecidable, and reasoning about real arithmetic is prohibitively expensive: for example polynomial real arithmetic is doubly exponential [38]. Still, using the deductive proving system of differential dynamic logic (dL), for example, it is possible to prove most theorems since dL is complete for algebraic invariants [127].

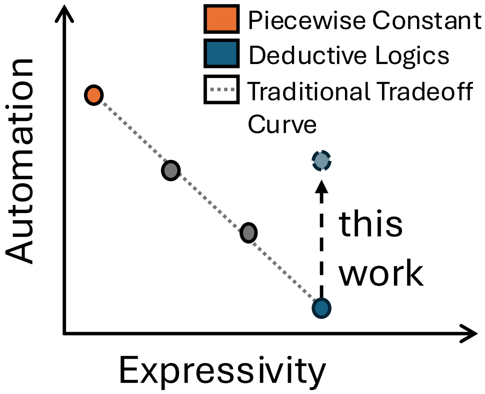


Figure 4.1: Schematic representation of tradeoffs in formal methods applied to Hybrid Systems.

proving for mathematics and (purely discrete) software [129, 132, 155]. This chapter explores the same premise for the data-scarce, mathematically complex domain of hybrid systems and games. We identify five challenging case studies in verification and create LLM-assisted pipelines to prove them. The implication is a change in the tradeoff landscape of formal methods for hybrid systems, as the application of LLMs brings unprecedented automation to symbolic, deductive verification, illustrated schematically in Fig. 4.1.

We create a pipeline that lets the LLM interact with the interactive theorem prover KeYmaera X over multiple rounds, analyzing the problem, and summarizing and fixing its own mistakes. This verification engine is already useful for manual control envelope design, since it automates the checking step after the human hypothesizes a control envelope. But additionally, in later chapters, it will be used as a subroutine while automatically *synthesizing* control envelopes. The key contribution of this chapter is an *automatic theorem proving system* that scales symbolic logic based verification for hybrid systems and games, with the pipeline design justified by ablation experiments on challenging case studies. The implementation and the experiments reported (LLM responses cached and re-playable) are available online at `https://doi.org/10.1184/R1/32248389` [74].

4.2 Related Work

Hybrid games, and in particular the tactic language of KeYmaera X, are significantly more data-scarce than *LLM-based verification* in mathematics [129, 155] and software [86, 88, 132], with few proof and synthesis examples available online. To our knowledge, the work that this chapter is based on [75] is the first exploration of LLM-based verification and synthesis for hybrid systems and games using a deductive, high-level symbolic logic like differential game logic [121].

Theorem provers for hybrid systems traditionally focus on a mix of interactive proof techniques and proof heuristics for automation [103]. HHL Prover [159] and HHLPy [140] use proof scripts with annotated invariants for loops and differential equations [103]. IsaVODEs [163] uses mostly scripted proofs with automation provided by Isabelle [113]. Plaidypvs [141] embeds differential dynamic logic in PVS. KeYmaera X [52] selects proof tactics based on the syntactic shape of proof obligations; proof heuristics provide a manually selected priority between tactic alternatives, attempted in order with timeouts. These proof heuristics achieve automation in hybrid systems, nonlinear continuous systems, and hybrid games, outperforming even proof scripts of other hybrid systems theorem provers [102, 103]. This chapter introduces a reasoning pipeline to push automation beyond the reach of KeYmaera X heuristics.

4.3 Case Studies

Five case studies are carefully developed to pose *challenging verification and synthesis problems*. None of them are solved automatically by existing tools alone. The synthesis variants of the problems consist of *hybrid games* defining the control options and environment behaviors that are physically possible, the formal desired control objectives, and *informal descriptions* of the desired control behavior within the many possibilities. The synthesis question consists of identifying the exact symbolic constraints that characterize which control action can be taken when. Multiple control decisions may be permitted at any given time, and following different decisions leads to different controllers, each of which is correct. Thus a synthesized solution identifies a *set* of control solutions, each of which is guaranteed to be correct. In the *verification* variants of the problems, the input is the same hybrid game, restricted to a set of (uncountably infinitely many) initial states for which *we think* there exists a correct solution. The task is to *prove* that there exists a control solution that works for all of these initial states using the theorem prover KeYmaera X.

Lotka-Volterra

The first problem (Model 6), which serves as a running example, is about maintaining safe populations under the *nonlinear* predator-prey *Lotka-Volterra* model given the ability to introduce additional predators and prey at unpredictable intervals of *unbounded duration*. Section 4.5.1 presents this example in further detail, demonstrating how a combination of high-level understanding of the dynamical behavior of the problem as well as low-level knowledge of the dGL calculus is required to solve it.

Train

The second case study (Model 23) is about verifying the ability of a *train* control system to maintain safety over train dynamics with *Davis resistance (nonlinear) and time-dependent airbrakes*. This case study poses structural complexity with a *nested inner loop* over *different modes* of air brake operation. The structural complexity demands a long proof tactic during verification, and the generation of multiple interdependent control constraints/invariants during synthesis.

Chemical Reaction

The third problem (Model 24) is about maintaining safe temperatures during a *chemical reaction*. This system again requires reasoning about complex dynamics and different, arbitrarily alternating control modes over unbounded iterations. The solution to the dynamics lies *outside*

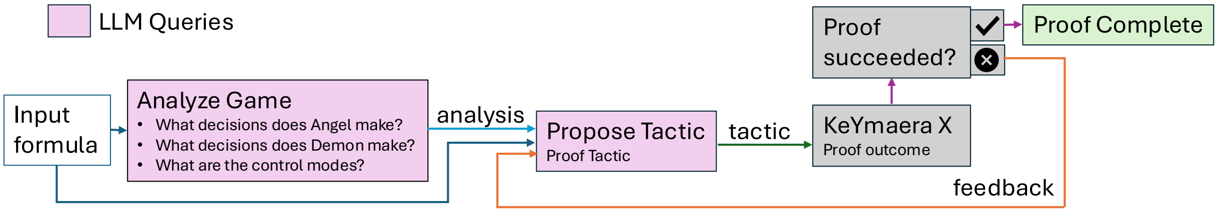


Figure 4.2: Automated Theorem Proving Pipeline. Appendix C.3 shows prompts.

the decidable (polynomial) fragment of real arithmetic, requiring non-trivial *deductive reasoning about differential equations via invariants*.

Coolant

The fourth problem (Model 25) is about the ability of a *coolant* system to meet heat absorption quotas in a nuclear power plant while limiting coolant discharge, with a *complex, timed, reach-avoid objective*, different discrete modes, and dynamics that behave very differently depending on initial conditions of variables. Anecdotally, it took a human expert three days to manually prove this case study.

Van der Pol

The fifth problem (Model 26) is about choosing the right initial conditions for a *Van der Pol* oscillator to maintain a safety property, which again requires reasoning about *nonlinear dynamics* over unbounded iterations, and therefore, effectively, *unbounded time*. The *chaotic* nature of the Van der Pol oscillator dynamics makes the *choice of initial conditions* delicate. *Appendix C.2 presents these case studies in detail.*

4.4 Automated Theorem Proving

We design an automated theorem proving (ATP) framework that uses LLMs to prove hybrid game/systems theorems (including those representing control envelopes) in KeYmaera X. A strength of our framework is its generality: we design the pipeline and prompts to focus on generic information about all parts of dGL, and the pipeline works across all five case studies without any case-study-specific engineering.

4.4.1 Pipeline Description

Our automated theorem proving system follows the pipeline in Fig. 4.2. It uses the natural design of a conversational agent that iteratively proposes tactics that KeYmaera X checks. With access to the full conversation history, it corrects mistakes based on KeYmaera X output. To make the LLM aware of the high-level meaning of the theorem, we add an initial problem analysis step.

In the *Analyze Game* step, given information about the syntax of dGL, the LLM produces an analysis of the input formula that contains the following parts: what actions are controlled by which player, what modes (regions of different qualitative behavior) the game has, and comments about the overall game play pattern. Then the proving loop begins. This consists of two steps repeated until the proof is complete. First, the LLM is prompted to propose a tactic. The prompt includes a *detailed guide on writing tactics in Bellerophon*, the tactic language of KeYmaera X [53]. The LLM is given the formula to prove, and a record of past proof attempts. The second step is to run the proposed tactic in KeYmaera X. If the tactic completes the proof, the pipeline ends successfully. Otherwise, KeYmaera X produces some output indicating where the proof failed. This output feeds back into the next tactic proposal step, continuing the loop. Appendix C.3 lists the detailed prompts for each query. The pipeline is implemented using the Delphyne framework [87] for oracular programming with LLMs.

A useful variant of this pipeline is one where after each iteration of the proving loop, the LLM is given a chance to reflect and decide if in fact the original theorem it is trying to prove is not valid. This variant is a useful component in our synthesis system later in the pipeline, where it helps identify and repair incorrect control conditions. Additionally, when it is important to limit input context length, making long histories of proof attempts impractical, a variant of the pipeline summarizes the proof state, mistakes so far, and global strategy after each iteration of the proving loop.

4.4.2 Outcomes

Table 4.1 summarizes the outcomes of running the tool. We try multiple models, running three times on each case study (except for GPT-5.5 with no reasoning, where we run only once per benchmark because of high dollar costs for failure runs[2]). We place an output limit of three hundred thousand tokens; successful runs are generally much shorter than this, and a failed run occurs when the tool fails to find a proof within this token limit. A case study is *solved* when at least one run finds a proof. *Pass@1* measures the average, over case studies, of the probability that a single run succeeds. None of the case studies can be solved by KeYmaera X's existing automation alone [142].

GPT-5.5 with high reasoning effort achieves the best pass@1 over all tested configurations, succeeding in all runs. GPT-5.4 with high reasoning effort also solves all problems, albeit with some failed runs. GPT-5.4-mini, a much smaller model, with high reasoning effort solves only 2/5 case studies. GPT-5.4-mini without reasoning, as well as Qwen3.5 32B A3B with thinking enabled, run locally, solve 1/5 case studies. The pipeline for the Qwen experiment is modified to include a summarization step after each iteration of the proving loop instead of retaining the entire conversation history in order to avoid exceeding the input context window. Without reasoning, models perform significantly worse. Table 4.2 shows a detailed per-benchmark breakdown of the results.

The results suggest that our tool is general relative to the LLM used. As smaller and open-source models catch up in capability with large frontier models, the tool will become easier and cheaper to use. Section 4.5 discusses the successes and failures in-depth.

---

[2] An observation that may not be immediately apparent from Table 4.1 is that failed runs that exhaust the token budget can be very costly. For example, although the successful runs of GPT-5.5 without reasoning cost only $17.85, the total cost of all five runs was $127.03 because the failed runs cost $63.43 and $45.75 (with the difference between these two caused by different input token counts).

Model	**Reasoning**	**Solved**	**Pass@1**	**Avg Cost ($)**	**Avg Calls**	**Avg Output (k Tokens)**
Qwen3.5	✓	1/5	0.13 ± 0.13	—	88.0 ± 18.0	254 ± 17
GPT-5.4-mini		1/5	0.13 ± 0.13	1.07 ± 1.05	138.5 ± 130.5	41 ± 39
GPT-5.4-mini	✓	2/5	0.33 ± 0.21	0.70 ± 0.17	11.0 ± 2.7	148 ± 36
GPT-5.4	✓	5/5	0.93 ± 0.07	2.21 ± 0.44	9.4 ± 1.4	92 ± 18
GPT-5.5		3/5	0.60 ± 0.24	5.95 ± 5.25	47.3 ± 33.7	35 ± 30
GPT-5.5	✓	**5/5**	$\mathbf{1.00 \pm 0.00}$	$\mathbf{2.05 \pm 0.56}$	$\mathbf{6.1 \pm 1.3}$	$\mathbf{63 \pm 17}$

Table 4.1: Verification results across models. *Reasoning* indicates whether `reasoning_effort` is set to `high` or `none` (OpenAI), or enabled (Qwen). *Solved* indicates the number of case studies where at least one run succeeded. *Pass@1* is the mean, over case studies, of the empirical success rate of a single run ± standard error across case studies. Cost, calls, and output tokens are averaged *only over successful runs* ± standard error. Output token counts are reported in thousands. As Qwen3.5 runs are local, they do not have costs.

Configuration	**Chem. Reaction**		**Coolant**		**Train**		**Lotka-Volterra**		**Van der Pol**	
	Succ.	Cost	Succ.	Cost	Succ.	Cost	Succ.	Cost	Succ.	Cost
Qwen3.5, reas.	2/3	—	0/3	—	0/3	—	0/3	—	0/3	—
GPT-5.4-mini, no reas.	2/3	1.07	0/3	—	0/3	—	0/3	—	0/3	—
GPT-5.4-mini, reas.	2/3	0.33	0/3	—	0/3	—	0/3	—	**3/3**	**0.95**
GPT-5.4, reas.	3/3	0.81	3/3	2.09	**3/3**	**1.22**	**3/3**	**4.38**	2/3	2.74
GPT-5.5, no reas.	**1/1**	**0.17**	0/1	—	1/1	16.44	0/1	—	1/1	1.24
GPT-5.5, reas.	3/3	0.88	**3/3**	**1.80**	3/3	1.89	3/3	4.68	3/3	1.01

Table 4.2: Per-benchmark verification results. *Succ.*: successful runs out of total runs. *Cost*: average dollar cost of *successful runs*.

4.5 Verification Outcomes Discussion

We next discuss the results in depth. Section 4.5.1 first sets up the background knowledge and context for this discussion by walking through an example.

4.5.1 Verification Impact by Example

This section illustrates how the combination of a symbolic logic with the right level of expressivity and LLMs to automate reasoning resolves challenges beyond the reach of existing automatic techniques. For simplicity of explanation, we focus on one case study in particular, but the style of challenges and their resolutions generalize to all case studies and most dGL theorems including those about control envelopes.

Case Study: Lotka-Volterra

We introduce the first case study as a running example. A forest department must maintain predator population `y` and prey population `x` above minimum thresholds y_{mn} and x_{mn} respectively, for arbitrarily long. Because of unpredictable funding cycles, they can introduce additional prey and

Model 6 Lotka-Volterra Population Control.

$$\begin{array}{r|ll}
\textsf{assume} & 1 & \mathtt{x} \le \frac{g}{d} \land \mathtt{y} \le \frac{a}{b} \land \frac{g}{d} \ge x_{\text{mn}} \land \frac{a}{b} \ge y_{\text{mn}} \land \mathtt{x} > 0 \land \mathtt{y} > 0 \\
 & 2 & \quad \land\, a > 0 \land b > 0 \land d > 0 \land g > 0 \land \mathtt{x} \ge x_{\text{mn}} \land \mathtt{y} \ge y_{\text{mn}} \rightarrow \\
\textsf{ctrl} & 3 & \langle(\ \mathtt{x\_add} := *; ?\mathtt{x\_add} \ge 0\,;\ \mathtt{y\_add} := *; ?\mathtt{y\_add} \ge 0\,; \\
 & 4 & \quad \mathtt{x} := \mathtt{x} + \mathtt{x\_add}\,;\ \mathtt{y} := \mathtt{y} + \mathtt{y\_add}\,; \\
\textsf{plant} & 5 & \quad \{\mathtt{x}' = a\mathtt{x} - b\mathtt{x}\mathtt{y},\ \mathtt{y}' = d\mathtt{x}\mathtt{y} - g\mathtt{y}\}^{d} \\
\textsf{safe} & 6 & \ )^{\times}\rangle(\mathtt{x} \ge x_{\text{mn}} \land \mathtt{y} \ge y_{\text{mn}})
\end{array}$$

predators at unpredictable intervals, that could last arbitrarily long. This case study is inspired by an ARCH-COMP benchmark [55], but is modified by the addition of discrete control choices and adversarial game play.

Model 6 Explanation

Model 6 shows the dGL formula modeling this problem via a *game*, with two players, canonically called Angel (representing the forest department) and Demon (representing uncertainties)[3]. Angel chooses the number of animals to introduce (Lines 3 and 4)[4], while Demon adversarially chooses the length of the interval until the next opportunity to introduce animals (Line 5), as well as how many iterations the game runs for (Line 6), forcing Angel to maintain safe populations for arbitrarily long[5]. Angel wins if in the end populations remain above the minimum thresholds (Line 6). The overall modal formula of the form $\langle\alpha\rangle\phi$ is true in states where Angel has a strategy to play in α so that regardless of how Demon plays in α, she will win (i.e., postcondition ϕ holds). Model 6 asks us to prove that, under the assumptions of Lines 1 and 2, Angel can always win.

This case study is symbolic, with different behavior depending on the values of the parameters a, b, d, g, and variables $\mathtt{x}$ and $\mathtt{y}$. Our reasoning must account for all uncountably infinite possible values that these symbols can take. To verify this formula, we will necessarily have to reason about the *nonlinear dynamics of the Lotka-Volterra equation over unbounded time horizons* (Line 5). While only a few automatic solvers can reason about nonlinear dynamics, even these cannot reason about reachability over unbounded time horizons[6]. Then we must consider the discrete and adversarial dynamics: Angel chooses *some* non-negative real number of animals to introduce (Line 3) and we must verify whether *any* of her uncountably infinite choices will work over the nonlinear, unbounded-time dynamics. Additionally, we must consider what hap-

---

[3] We use the typographic convention of monospaced font for `variables` that change over the course of a game and math font for each *symbolic parameter* P that can represent any real number which remains constant over the course of a game.

[4] In the game $\mathtt{x\_add} := *$, Angel freely chooses any real number to assign to $\mathtt{x\_add}$. Then, $?\mathtt{x\_add} \ge 0$ asserts that $\mathtt{x\_add}$ must be non-negative.

[5] In the game $\{x' = f(x)\}^d$, the state evolves per the flow of the differential equation $x' = f(x)$ for a duration chosen by Demon. In the game $(\alpha)^{\times}$, Demon chooses how many times to repeat the game α.

[6] For instance, at ARCH-COMP 2025, where benchmark 5 involved Lotka-Volterra dynamics, the aim was only to solve for 3.64 time units.

pens if this entire process repeats *arbitrarily many times* in a loop (Line 6). This explosion in possible behaviors is hard for traditional automatic techniques to deal with.

Yet a symbolic proof is possible by translating a high-level physical understanding of the problem to a clever choice of deductive proof rules. Looking closely at the qualitative behavior of the Lotka-Volterra dynamics reveals a way for Angel to win. The dynamics have an *equilibrium point* at $(\mathrm{x}, \mathrm{y}) = (g/d, a/b)$, which is above the minimum populations threshold per assumptions $a/b \geq y_{\text{mn}}$ and $g/d \geq x_{\text{mn}}$ (Model 6, Line 1). If Angel manages to set the populations to this equilibrium point at the start of each iteration, then she will satisfy her winning condition at the end of the iteration, regardless of how long Demon runs the dynamics. Since Angel can introduce any non-negative number of animals, all she needs at the start of each iteration is that predator and prey populations are no greater than the equilibrium point. Finally, many iterations of this maneuver string together correctly over loop iterations because at the end of each iteration, populations are at the equilibrium point, satisfying the requirement that populations are no greater than the equilibrium point.

$$\text{(loop)}\quad \frac{\Gamma \vdash J, \Delta \quad J \vdash [\alpha]J \quad J \vdash P}{\Gamma \vdash [\alpha^*]P, \Delta} \qquad\qquad (\exists\text{R})\quad \frac{\Gamma \vdash p(e), \Delta}{\Gamma \vdash \exists x\, p(x), \Delta}\ \text{(arbitrary term } e)$$

$$\text{(dC)}\quad \frac{\Gamma \vdash [x' = f(x) \,\&\, Q]C, \Delta \quad \Gamma \vdash [x' = f(x) \,\&\, (Q \wedge C)]P, \Delta}{\Gamma \vdash [x' = f(x) \,\&\, Q]P, \Delta}$$

$$\text{(dI)}\quad \frac{Q \vdash [x' := f(x)](F)'}{F \vdash [x' = f(x) \,\&\, Q]F}$$

Figure 4.3: dGL Proof Rules and Axioms relevant to the proof of Model 6. The rules are as follows: differential invariant (dI), differential cut (dC), loop (loop), and exists right (existsr).

To translate this high-level physical reasoning to a formal proof, the right choice of dGL proof rules[7] is critical, and traditionally provided by a human expert who understands what proof rule is applicable when. To show the first step in our reasoning above, that $(\mathrm{x}, \mathrm{y}) = (g/d, a/b)$ is an equilibrium point of the dynamics, the rule most commonly used to prove invariants of differential equations (the *differential invariant* rule) does not immediately apply. Instead, a quick proof is possible using the more rarely used *differential radical invariant* [58] rule (Eq. (4.1)). It allows us to conclude that polynomial equation $h_j = 0$ is an invariant of the dynamics if we can show that its Lie derivatives $\mathcal{L}_d(h_j), \mathcal{L}_d^2(h_j), \ldots \mathcal{L}_d^N(h_j)$ are zero, where N is the finite derivative order at which the chain of ideals $\langle h \rangle \subseteq \langle h, \mathcal{L}_p(h) \rangle \subseteq \cdots$ reaches a fixed point.

$$\text{(dRI)}\quad \frac{\Gamma, Q \vdash \wedge_i^N \wedge_j \mathcal{L}_p^i(h_j) = 0}{\Gamma \vdash [x' = p \,\&\, Q] \wedge_j h_j = 0} \tag{4.1}$$

Eq. (4.2) shows how this applies to our case. Read from bottom to top, the differential radical invariant rule (dRI) applies to produce a sequent that is propositional in the decidable fragment of

---

[7] Fig. 4.3 lists all the rules referenced in this discussion; the intent is not to dwell on their individual details, but to highlight that the proof requires a sequence of complex rule applications that resist straightforward (traditional) automation.

real arithmetic, which an automatic prover like Z3 [39] can discharge, establishing an equilibrium point of these nonlinear dynamics.

$$\mathrm{dRI}\,\dfrac{\dfrac{*}{\mathtt{x} - g/d = 0 \land \mathtt{y} - a/b = 0 \vdash d(a\mathtt{x} - b\mathtt{x}\mathtt{y}) = 0 \land b(d\mathtt{x}\mathtt{y} - g\mathtt{y}) = 0}}{\mathtt{x} - g/d = 0 \land \mathtt{y} - a/b = 0 \vdash [\mathtt{x}' = a\mathtt{x} - b\mathtt{x}\mathtt{y},\ \mathtt{y}' = d\mathtt{x}\mathtt{y} - g\mathtt{y}](\mathtt{x} - g/d = 0 \land \mathtt{y} - a/b = 0)} \tag{4.2}$$

This reasoning step must be carefully combined with the other reasoning steps to complete the proof. In this example we also need to apply rules such as the *loop* rule with the correct invariant $J \equiv \mathtt{x} \geq x_{\text{mn}} \land \mathtt{y} \geq y_{\text{mn}} \land \mathtt{x} \leq g/d \land \mathtt{y} \leq a/b \land g/d \geq x_{\text{mn}} \land a/b \geq y_{\text{mn}} \land a > 0 \land b > 0 \land d > 0 \land g > 0$ to demonstrate the inductiveness of Angel's strategy over many iterations, and the *exists right* rule with a correct expression for Angel's choice of `x_add` and `y_add`. Identifying and applying these rules requires both an understanding of the high-level solution strategy of the problem as well as dGL's proof rules, leading to difficulties in automation using traditional techniques which lack the former and need careful, case-by-case encoding of the latter.

Proofs can be expressed succinctly as *tactic* scripts, which are sequences of proof rules with their relevant parameters that when applied in order, complete the proof. Our pipeline was able to automatically generate a 24 line tactic script that completed the proof for this case study, listed in Appendix C.1.

4.5.2 Verification Experimental Outcomes Discussion

This section provides some insight into the qualitative behavior of the pipeline, and the common failure modes.

Comparison to Manual Proofs

The proofs generated by the tool are generally longer than human-written proofs. For example, a proof for the Coolant problem (Model 25) written by a human consisted of 45 lines of proof tactic, compared to the 103, 96, and 104 line tactics generated by the tool in the GPT-5.5 high reasoning runs. Part of this, however, can be explained by the fact that the LLMs, because of guidance in the prompt, tend to produce more robust tactics that rely less on KeYmaera X's heuristics-based `auto` tactic.

Tracking State

The ability to reason about proof state accurately is important to writing correct Bellerophon tactics, and deteriorates in the absence of reasoning and with smaller models. This leads to two common failure modes: incorrect *positional arguments* and mishandled *proof branching*.

Bellerophon tactics often require *positional arguments* indicating where in the sequent a rule should be applied (e.g., suppose the sequent has the shape $\Gamma \vdash [\alpha_1^*]\phi_1, [\alpha_2^*]\phi_2$. Here, KeYmaera X needs to be told whether to apply the loop rule to the formula $[\alpha_1^*]\phi_1$ at position 1 or the formula $[\alpha_2^*]\phi_2$ at position 2). Identifying the right positional arguments requires tracking the proof state accurately, causing problems for GPT-5.5 without reasoning, GPT-5.4-mini and Qwen3.5.

Proof branching occurs when a proof rule has multiple premises, for example, applying the loop rule leads to three branches (for the base case, inductive case, and postcondition), each of which must receive their own proofs. A KeYmaera X proof tactic, upon applying the loop rule, must therefore specify three sub-tactics to handle the three branches. Getting this right syntactically requires accurate tracking of the proof state. GPT-5.4-mini and Qwen3.5 face difficulties with branching tactics, often specifying too few or too many sub-tactics.

Provability of Invariants

Even amongst valid loop invariants (or their continuous analogs), some are easier to prove than others. Invariants that are mathematically more compact can sometimes be harder to prove than "flat" invariants that are a long conjunction/disjunction of simple expressions. This is, for example, the reason that the GPT-5.4 model finds the Van der Pol Oscillator problem (Model 26) hard, failing one run and requiring a larger budget even for the successful runs. It has a tendency to propose a more complex loop invariant than GPT-5.5.

Cost and Token Considerations

Generally, GPT-5.4 takes slightly longer to find solutions than GPT-5.5, but often costs less because of its lower per-token cost. For easier problems, non-reasoning runs sometimes have the advantage of not "overthinking" their solutions, and can also perform more queries within the token budget because of no expenditure on reasoning tokens. However, overall, reasoning greatly improves performance for both GPT-5.5 and GPT-5.4-mini, largely because of improved state-tracking and error-recovery based on KeYmaera X feedback.

The verification pipeline is not yet able to automatically verify the full, most complex controller from Chapter 3. For this problem, the manually written tactic exceeded a thousand lines, making the proof so long that the pipeline "forgets" previous mistakes and is unable to complete the proof within the budget of our experiments. Future improvements to the pipeline, such as better summarization techniques, may help it scale to this proof as well. Nevertheless, the system is already useful in verification since it can still close the simpler lemmas that arise in the course of completing the larger proof. Further, the system is also useful for *synthesis*, which uses verification as a subroutine after breaking down the synthesis problem into smaller pieces. Because of how synthesis decomposes the problem, theorems to prove are generally shorter than the control envelopes, allowing this verification pipeline to help *synthesize* the full control envelope over the course of many smaller verification problems, which will be shown in Chapter 7.

4.6 Conclusion

This chapter created an automated theorem proving system that uses LLMs to prove theorems in KeYmaera X, and demonstrated its effectiveness on five challenging case studies. It already has practical utility for manual control envelope design, but becomes even more important in Chapter 7 as a subroutine for automatic control envelope synthesis.

Chapter 5

Control Envelope Synthesis via Angelic Refinement

This chapter presents an approach for synthesizing provably correct control envelopes for hybrid systems. The algorithm fills in the blanks of a hybrid system's sketch specifying the desired shape of the control envelope, the possible control actions, and the system's differential equations. In order to maximize the flexibility of the control envelope, the synthesized conditions saying which control action can be chosen when should be as permissive as possible while establishing a desired safety condition from the available assumptions, which are augmented if needed. An implicit, optimal solution to this synthesis problem is characterized using hybrid systems game theory, from which explicit solutions can be derived via symbolic execution and sound, systematic game refinements. Optimality can be recovered in the face of approximation via a dual game characterization. The resulting algorithm, *Control Envelope Synthesis via Angelic Refinements (CESAR)*, is demonstrated in a range of safe control envelope synthesis examples with different control challenges. This chapter solves the synthesis problem for *time-triggered* control, and will be generalized by the next chapter.

5.1 Introduction

Control envelope design is challenging. Engineers are good at specifying the *shape* of a model and listing the possible control actions by translating client specifications, which is crucial for the fidelity of the resulting model. But identifying the exact control conditions required for safety in such a model is much harder, requiring design insights and creativity, and is the main point of the deep area of control theory. Most initial system designs are incorrect and need to be fixed before verification succeeds. Fully rigorous justification of the safety of the control conditions requires full verification of the resulting controller in the hybrid systems model. This chapter presents a synthesis technique that addresses this hard problem by filling in the holes of a hybrid systems model to identify a correct-by-construction control envelope that is as permissive as possible.

Our approach is called *Control Envelope Synthesis via Angelic Refinements (CESAR)*. The idea is to implicitly characterize the optimal safe control envelope via hybrid games yielding maximally permissive safe solutions in differential game logic [123]. To derive explicit solutions

Model 7 The train ETCS model (slightly modified from [126]). Framed formulas are initially blank and are automatically synthesized by our tool as indicated.

```
   assum | 1  $A > 0 \land B > 0 \land T > 0 \land v \geq 0 \land$
ctrlable | 2  $\boxed{e - p > v^2/2B} \rightarrow [\{$
    ctrl | 3      $( \quad (?\boxed{e - p > vT + AT^2/2 + (v + AT)^2/2B}\,;\ a := A)$
         | 4      $\cup\ (?\boxed{\text{true}}\,;\ a := -B) \quad )\,;$
   plant | 5      $(t := 0\,;\ \{p' = v, v' = a, t' = 1 \,\&\, t \leq T \land v \geq 0\})$
    safe | 6  $\}^*](e - p > 0)$
```

used for controller monitoring at runtime, we successively refine the games while preserving safety and, if possible, optimality. Our experiments demonstrate that CESAR solves hybrid systems synthesis challenges requiring different control insights.

Contributions. The primary contributions of this chapter behind CESAR are:

- optimal hybrid systems control envelope synthesis via hybrid games.
- differential game logic formulas identifying optimal safe control envelopes.
- refinement techniques for safe control envelope approximation, including *bounded fixpoint unrollings* via a recurrence, which exploits *action permanence* (a hybrid analogue to idempotence).
- a primal/dual game counterpart optimality criterion.

5.2 Background

Control Safety Envelopes by Example. In order to separate safety-critical aspects from other system goals during control design, we abstractly describe the safe choices of a controller with safe control envelopes that deliberately underspecify when and how to exactly execute certain actions. They focus on describing in which regions it is safe to take actions. For example, Model 7 designs a train control envelope [126] that must stop the train by the *end of movement authority* e located somewhere ahead, as assigned by the train network scheduler. Past e, there may be obstacles or other trains. The train's control choices are to accelerate or brake as it moves along the track. The goal of CESAR is to synthesize the framed formulas in the model, that are initially blank.

Line 6 describes the *safety property* that is to be enforced at all times: the train driving at position p with velocity v must not go past position e. Line 1 lists *modeling assumptions*: the train is capable of both acceleration ($A>0$) and deceleration ($B>0$), the controller latency is positive ($T>0$) and the train cannot move backwards as a product of braking (this last fact is also reflected by having $v \geq 0$ as a domain constraint for the plant on Line 5). These assumptions are fundamentally about the physics of the problem being considered. In contrast, Line 2 features a *controllability assumption* that can be derived from careful analysis. Here, this synthesized assumption says that the train cannot start so close to e that it won't stop in time even if it starts braking immediately. Line 3 and Line 4 describe a train controller with two actions: accelerating ($a := A$) and braking ($a := -B$). Each action is guarded by a synthesized formula, called an *action guard* that indicates when it is safe to use. Angel has control over which action runs, and adversarially plays with the objective of violating safety conditions. But Angel's options are limited to only safe ones because of the synthesized action guards, ensuring that Demon still wins and the overall formula is valid. In this case, braking is always safe whereas acceleration can only be allowed when the distance to end position e is sufficiently large. Finally, the plant on Line 5 uses differential equations to describe the train's kinematics. A timer variable t is used to ensure that no two consecutive runs of the controller are separated by more than time T. Thus, this controller is *time-triggered*.

Overview of CESAR. CESAR first identifies the optimal solution for the blank of Line 2. Intuitively, this blank should identify a *controllable invariant*, which denotes a set of states where a controller with choice between acceleration and braking has some strategy (to be enforced by the conditions of Line 3 and Line 4) that guarantees safe control forever. Such states can be characterized by the following dGL formula where Demon, as a proxy for the controller, decides whether to accelerate or brake: $[((a := A \cap a := -B)\,; \mathsf{plant})^*]\,\mathsf{safe}$ where plant and safe are from Model 7. When this formula is true, Demon, who decides when to brake to maintain the safety contract, has a winning strategy that the controller can mimic. When it is false, Demon, a perfect player striving to maintain safety, has no winning strategy, so a controller has no guaranteed way to stay safe either.

This dGL formula provides an *implicit* characterization of the optimal controllable invariant from which we derive an explicit formula in $\mathcal{P}_{\mathbb{R}}$ to fill the blank with using symbolic execution. Symbolic execution solves a game following the axioms of dGL to produce an equivalent $\mathcal{P}_{\mathbb{R}}$ formula (Section 5.3.7). However, our dGL formula contains a loop, for which symbolic execution will not terminate in finite time. To reason about the loop, we *refine* the game, modifying it so that it is easier to symbolically execute, but still at least as hard for Demon to win so that the controllable invariant that it generates remains sound. In this example, the required game transformation first restricts Demon's options to braking. Then, it eliminates the loop using the observation that the repeated hybrid iterations $(a := -B; \mathsf{plant})^*$ behave the same as just following the continuous dynamics of braking for unbounded time. It replaces the original game with $a := -B\,; t := 0\,; \{p' = v, v' = a \;\&\; \wedge v \geq 0\}$, which is loop-free and easily symbolically executed. Symbolically executing this game to reach safety condition safe yields controllable invariant $e - p > \frac{v^2}{2B}$ to fill the blank of Line 2.

Intuitively, this refinement (formalized in Section 5.3.4) captures situations where the controller stays safe forever by picking a single control action (braking). It generates the optimal solution for this example because braking forever is the dominant strategy: given any state, if braking forever does not keep the train safe, then certainly no other strategy will. However, there are other problems where the dominant control strategy requires the controller to strategically switch between actions, and this refinement misses some controllable invariant states. So we introduce a new refinement: bounded game unrolling via a recurrence (Section 5.3.5). A solution generated by unrolling n times captures states where the controller can stay safe by switching control actions up to n times.

Having synthesized the controllable invariant, CESAR fills the action guards (Line 3 and Line 4). An action should be permissible when running it for one iteration maintains the controllable invariant. For example, acceleration is safe to execute exactly when $[a := A; \mathsf{plant}]e - p > \frac{v^2}{2B}$. We symbolically execute this game to synthesize the formula that fills the guard of Line 3.

5.3 Approach

This section formally introduces the *Control Envelope Synthesis via Angelic Refinements (CESAR)* approach for hybrid systems control envelope synthesis.

5.3.1 Problem Definition

We frame the problem of *control envelope synthesis* in terms of filling in holes $\sqcup$ in a problem of the following shape:

$$\mathsf{prob} \;\equiv\; \mathsf{assum} \wedge \sqcup \rightarrow \left[\left(\left(\cup_i \left(?\sqcup_i \,;\, \mathsf{act}_i\right)\right) ; \mathsf{plant}\right)^*\right] \mathsf{safe}. \tag{5.1}$$

Here, the control envelope consists of a nondeterministic choice between a finite number of guarded actions. Each action act_i is guarded by a condition $\sqcup_i$ to be determined in a way that ensures safety within a controllable invariant [9, 60] $\sqcup$ to be synthesized also. The plant is defined by the following template:

$$\mathsf{plant} \;\equiv\; t := 0\,;\, \{x' = f(x),\, t' = 1 \;\&\; \mathsf{domain} \wedge t \le T\}. \tag{5.2}$$

This ensures that the plant must yield to the controller after time T at most, where T is assumed to be positive and constant. In addition, we make the following assumptions:

1. Components assum, safe and domain are propositional arithmetic formulas.
2. Timer variable t is fresh (does not occur except where shown in template).
3. Programs act_i are discrete dL programs that can involve choices, assignments and tests with propositional arithmetic. Variables assigned by act_i must not appear in safe. In addition, act_i must terminate in the sense that $\vDash \langle \mathsf{act}_i \rangle$ true.
4. The modeling assumptions assum are invariant in the sense that

$$\vDash \mathsf{assum} \rightarrow [(\cup_i \, \mathsf{act}_i)\,;\, \mathsf{plant}]\, \mathsf{assum}.$$

   This holds trivially for assumptions about constant parameters such as $A > 0$ in Model 7 and this ensures that the controller can always rely on them being true.

Definition 5.1. A *solution* to the synthesis problem above is defined as a pair (I, G) where I is a formula and G maps each action index i to a formula G_i. In addition, the following conditions must hold:

1. Safety is guaranteed: $\mathsf{prob}(I, G) \equiv \mathsf{prob}[\sqcup \mapsto I, \sqcup_i \mapsto G_i]$ is valid and $(\mathsf{assum} \wedge I)$ is a loop invariant that proves it so.
2. There is always some action: $(\mathsf{assum} \wedge I) \rightarrow \bigvee_i G_i$ is valid.

Condition 2 is crucial for using the resulting nondeterministic control envelope, since it guarantees that safe actions are always available as a fallback.

5.3.2 An Optimal Solution

Solutions to a synthesis problem may differ in quality. Intuitively, a solution is better than another if it allows for a strictly larger controllable invariant. In case of equality, the solution with the more permissive control envelope wins. Formally, given two solutions $S = (I, G)$ and $S' = (I', G')$, we say that S' is better or equal to S (written $S \sqsubseteq S'$) if and only if $\models \mathsf{assum} \rightarrow (I \rightarrow I')$ and additionally either $\models \mathsf{assum} \rightarrow \neg(I' \rightarrow I)$ or $\models (\mathsf{assum} \land I) \rightarrow \bigwedge_i (G_i \rightarrow G'_i)$. Given two solutions S and S', one can define a solution $S \sqcap S' = (I \lor I',\, i \mapsto (I \land G_i \lor I' \land G'_i))$ that is better or equal to both S and S' ($S \sqsubseteq S \sqcap S'$ and $S' \sqsubseteq S \sqcap S'$). A solution S' is called the *optimal solution* when it is the maximum element in the ordering, so that for any other solution S, $S \sqsubseteq S'$. The optimal solution exists and is expressible in dGL:

$$I^{\,\mathrm{opt}} \equiv [((\cap_i\, \mathsf{act}_i)\,;\, \mathsf{plant})^*]\, \mathsf{safe} \tag{5.3}$$

$$G_i^{\,\mathrm{opt}} \equiv [\mathsf{act}_i\,;\, \mathsf{plant}]\, I^{\,\mathrm{opt}}. \tag{5.4}$$

Intuitively, $I^{\,\mathrm{opt}}$ characterizes the set of all states from which an optimal controller (played here by Demon) can keep the system safe forever. In turn, $G^{\,\mathrm{opt}}$ is defined to allow any control action that is guaranteed to keep the system within $I^{\,\mathrm{opt}}$ until the next control cycle as characterized by a modal formula. Section 5.3.3 formally establishes the correctness and optimality of $S^{\,\mathrm{opt}} \equiv (I^{\,\mathrm{opt}},\, G^{\,\mathrm{opt}})$.

While it is theoretically reassuring that an optimal solution exists that is at least as good as all others and that this optimum can be characterized in dGL, such a solution is of limited practical usefulness since Eq. (5.3) cannot be executed without solving a game at runtime. Rather, we are interested in *explicit* solutions where I and G are quantifier-free real arithmetic formulas. There is no guarantee in general that such solutions exist that are also optimal, but our goal is to devise an algorithm to find them in the many cases where they exist or find safe approximations otherwise.

5.3.3 Controllable Invariants

The fact that $S^{\,\mathrm{opt}}$ is a solution can be characterized in logic with the notion of a controllable invariant that, at each of its points, admits some control action that keeps the plant in the invariant for one round. All lemmas and theorems throughout this chapter are proved in Appendix D.

Definition 5.2 (Controllable Invariant)**.** A *controllable invariant* is a formula I such that $\models I \rightarrow \mathsf{safe}$ and $\models I \rightarrow \bigvee_i [\mathsf{act}_i\,;\, \mathsf{plant}]\, I$.

From this perspective, $I^{\,\mathrm{opt}}$ can be seen as the largest controllable invariant.

Lemma 5.1. *$I^{\,\mathrm{opt}}$ is a controllable invariant and it is optimal in the sense that $\models I \rightarrow I^{\,\mathrm{opt}}$ for any controllable invariant I.*

Moreover, not just $I^{\,\mathrm{opt}}$, but *every* controllable invariant induces a solution. Indeed, given a controllable invariant I, we can define $\mathcal{G}(I) \equiv (i \mapsto [\mathsf{act}_i\,;\, \mathsf{plant}]\, I)$ for the *control guards induced by I*. $\mathcal{G}(I)$ chooses as the guard for each action act_i the modal condition ensuring that act_i, preserves I after the plant.

Lemma 5.2. *If I is a controllable invariant, then $(I, \mathcal{G}(I))$ is a solution (Def. 5.1).*

Conversely, a controllable invariant can be derived from any solution.

Lemma 5.3. *If (I, G) is a solution, then $I' \equiv (\mathsf{assum} \wedge I)$ is a controllable invariant. Moreover, we have $(I, G) \sqsubseteq (I', \mathcal{G}(I'))$.*

Solution comparisons w.r.t. $\sqsubseteq$ reduce to implications for controllable invariants.

Lemma 5.4. *If I and I' are controllable invariants, then $(I, \mathcal{G}(I)) \sqsubseteq (I', \mathcal{G}(I'))$ if and only if $\vDash \mathsf{assum} \rightarrow (I \rightarrow I')$.*

Taken together, these lemmas allow us to establish the optimality of $S^{\,\mathrm{opt}}$.

Theorem 5.1. *$S^{\,\mathrm{opt}}$ is an optimal solution (i.e. a maximum w.r.t. $\sqsubseteq$) of Def. 5.1.*

This shows the roadmap for the rest of the chapter: finding solutions to the control envelope synthesis problem reduces to finding controllable invariants that imply $I^{\,\mathrm{opt}}$, which can be found by restricting the actions available to Demon in $I^{\,\mathrm{opt}}$ to guarantee safety, thereby *refining* the associated game.

5.3.4 One-Shot Fallback Refinement

The simplest refinement of $I^{\,\mathrm{opt}}$ is obtained when fixing a single fallback action to use in all states (if that is safe). A more general refinement considers different fallback actions in different states, but still only plays one such action forever.

Using the dGL axioms, any loop-free dGL formula whose ODEs admit solutions expressible in real arithmetic can be automatically reduced to an equivalent first-order arithmetic formula (in $\mathrm{FOL}_{\mathbb{R}}$). An equivalent propositional arithmetic formula in $\mathcal{P}_{\mathbb{R}}$ can be computed via quantifier elimination (QE). For example:

$$\begin{aligned}
& [(v := 1 \cap v := -1)\,;\, \{x' = v\}]\, x \neq 0 & \\
\equiv\; & [v := 1 \cap v := -1]\, [\{x' = v\}]\, x \neq 0 & \text{by } [;] \\
\equiv\; & [v := 1]\, [\{x' = v\}]\, x \neq 0 \;\vee\; [v := -1]\, [\{x' = v\}]\, x \neq 0 & \text{by } [\cap] \\
\equiv\; & [\{x' = 1\}]\, x \neq 0 \;\vee\; [\{x' = -1\}]\, x \neq 0 & \text{by } [:=] \\
\equiv\; & (\forall t {\geq} 0\, x + t \neq 0) \vee (\forall t {\geq} 0\, x - t \neq 0) & \text{by } ['],[:=] \\
\equiv\; & x > 0 \;\vee\; x < 0 & \text{by QE }.
\end{aligned}$$

Even when a formula features nonsolvable ODEs, techniques exist to compute weakest preconditions for differential equations, with conservative approximations [142] or even exactly in some cases [22, 128]. In the rest of this section and for most of this chapter, we are therefore going to assume the existence of a reduce oracle that takes as an input a loop-free dGL formula and returns a quantifier-free arithmetic formula that is equivalent modulo some assumptions. Section 5.3.7 shows how to implement and optimize reduce.

Definition 5.3 (Reduction Oracle)**.** A *reduction oracle* is a function reduce that takes as an input a loop-free dGL formula F and an assumption $A \in \mathcal{P}_{\mathbb{R}}$. It returns a formula $R \in \mathcal{P}_{\mathbb{R}}$ along with a boolean flag exact such that the formula $A \rightarrow (R \rightarrow F)$ is valid, and if exact is true, then $A \rightarrow (R \leftrightarrow F)$ is valid as well.

Back to our original problem, $I^{\,\mathrm{opt}}$ is not directly reducible since it involves a loop. However, conservative approximations can be computed by restricting the set of strategies that the Demon player is allowed to use. One extreme case allows Demon to only use a single action act_i repeatedly as a fallback (e.g. braking in the train example). In this case, we get a controllable invariant $[(\mathsf{act}_i\,;\,\mathsf{plant})^*]\,\mathsf{safe}$, which further simplifies into $[\mathsf{act}_i\,;\,\mathsf{plant}_\infty]\,\mathsf{safe}$ with

$$\mathsf{plant}_\infty \equiv \{x' = f(x), t' = 1 \;\&\; \mathsf{domain}\}$$

a variant of plant that never yields control. For this last step to be valid though, a technical assumption is needed on act_i, which we call *action permanence*.

Definition 5.4 (Action Permanence)**.** An action act_i is said to be *permanent* if and only if $(\mathsf{act}_i\,;\,\mathsf{plant}\,;\,\mathsf{act}_i) \equiv (\mathsf{act}_i\,;\,\mathsf{plant})$, i.e., they are equivalent games.

Intuitively, an action is *permanent* if executing it more than once in a row has no consequence for the system dynamics. This is true in the common case of actions that only assign constant values to control variables that are read but not modified by the plant, such as $a := A$ and $a := -B$ in Model 7.

Lemma 5.5. *If* act_i *is permanent,* $\vDash [(\mathsf{act}_i\,;\,\mathsf{plant})^*]\,\mathsf{safe} \leftrightarrow [\mathsf{act}_i\,;\,\mathsf{plant}_\infty]\,\mathsf{safe}$.

Our discussion so far identifies the following approximation to our original synthesis problem, where P denotes the set of all indexes of permanent actions:

$$\begin{aligned} I^0 &\equiv [(\cap_{i\in\mathsf{P}}\,\mathsf{act}_i)\,;\,\mathsf{plant}_\infty]\,\mathsf{safe}, \\ G_i^0 &\equiv [\mathsf{act}_i\,;\,\mathsf{plant}]\,I^0. \end{aligned}$$

Here, I^0 encompasses all states from which the agent can guarantee safety indefinitely with a single permanent action. G^0 is constructed according to $\mathcal{G}(I^0)$ and only allows actions that are guaranteed to keep the agent within I^0 until the next control cycle. Note that I^0 degenerates to false in cases where there are no permanent actions, which does not make it less of a controllable invariant.

Theorem 5.2. I^0 *is a controllable invariant.*

Moreover, in many examples of interest, I^0 and $I^{\,\mathrm{opt}}$ are equivalent since an optimal fallback strategy exists that only involves executing a single action. This is the case in particular for Model 7, where

$$\begin{aligned} I^0 &\equiv [a := -B\,;\,\{p' = v, v' = a \;\&\; v \geq 0\}]\, e - p > 0 \\ &\equiv e - p > v^2/2B \end{aligned}$$

characterizes all states at safe braking distance to the obstacle and G^0 associates the following guard to the acceleration action:

$$\begin{aligned} G^0_{a:=A} &\equiv [a := A\,;\,\{p' = v, v' = a, t' = 1 \;\&\; v \geq 0 \wedge t \leq T\}]\, e - p > v^2/2B \\ &\equiv e - p > vT + AT^2/2 + (v + AT)^2/2B \end{aligned}$$

That is, accelerating is allowed if doing so is guaranteed to maintain sufficient braking distance until the next control opportunity. Section 5.3.6 discusses automatic generation of a proof that (I^0, G^0) is an optimal solution for Model 7.

5.3.5 Bounded Fallback Unrolling Refinement

In Section 5.3.4, we derived a solution by computing an underapproximation of $I^{\,\mathrm{opt}}$ where the fallback controller (played by Demon) is only allowed to use a one-shot strategy that picks a single action and plays it forever. Although this approximation is always safe and, in many cases of interest, happens to be exact, it does lead to a suboptimal solution in others. In this section, we allow the fallback controller to switch actions a bounded number of times before it plays one forever. There are still cases where doing so is suboptimal (imagine a car on a circular race track that is forced to maintain constant velocity). But this restriction is in line with the typical understanding of a fallback controller, whose mission is not to take over a system indefinitely but rather to maneuver it into a state where it can safely get to a full stop [115].

For all bounds $n \in \mathbb{N}$, we define a game where the fallback controller (played by Demon) takes at most n turns to reach the region I^0 in which safety is guaranteed indefinitely. During each turn, it picks a permanent action and chooses a time θ in advance for when it wishes to play its next move. Because the environment (played by Angel) has control over the duration of each control cycle, the fallback controller cannot expect to be woken up after time θ exactly. However, it can expect to be provided with an opportunity for its next move within the $[\theta, \theta + T]$ time window since the plant can never execute for time greater than T. Formally, we define I^n as follows:

$$I^n \;\equiv\; [\mathsf{step}^{\times \leq n}\,;\, \mathsf{forever}]\,\mathsf{safe} \qquad \mathsf{forever} \;\equiv\; (\cap_{i \in \mathsf{P}}\, \mathsf{act}_i)\,;\, \mathsf{plant}_\infty$$
$$\mathsf{step} \;\equiv\; (\theta := *\,;\, ?\theta \geq 0)^d\,;\, (\cap_{i \in \mathsf{P}}\, \mathsf{act}_i)\,;\, \mathsf{plant}_{\theta+T}\,;\, ?\mathsf{safe}^d\,;\, ?t \geq \theta$$

where $\mathsf{plant}_{\theta+T}$ is the same as plant, except that the domain constraint $t \leq T$ is replaced by $t \leq \theta + T$. Equivalently, we can define I^n by induction as follows:

$$I^{n+1} \;\equiv\; I^n \,\vee\, [\mathsf{step}]\, I^n \qquad I^0 \;\equiv\; [\mathsf{forever}]\,\mathsf{safe}, \tag{5.5}$$

where the base case coincides with the definition of I^0 in Section 5.3.4. Importantly, I^n is a loop-free controllable invariant and so reduce can compute an explicit solution to the synthesis problem from I^n.

Theorem 5.3. *I^n is a controllable invariant for all $n \geq 0$.*

Theorem 5.3 establishes a nontrivial result since it overcomes the significant gap between the *fantasized* game that defines I^n and the *real* game being played by a time-triggered controller. Our proof critically relies on the action permanence assumption along with the following property of differential equations, which establishes that ODE programs preserve a specific form of reach-avoid property as a result of being deterministic.

Lemma 5.6. *Consider a property of the form $R(a,b) \equiv [\alpha_b]\,(S \wedge (t \geq a \rightarrow I))$ with $\alpha_b \equiv (t := 0\,;\, \{x' = f(x), t' = 1 \,\&\, Q \wedge t \leq b\})$. Then this formula is valid:*

$$c \leq b \wedge R(a,b) \rightarrow [\alpha_c]\, R(a-t, b-t).$$

Example. As an illustration, consider the example in Fig. 5.1 and Model 8 of a 2D robot moving in a corridor that forms an angle. The robot is only allowed to move left or down at a

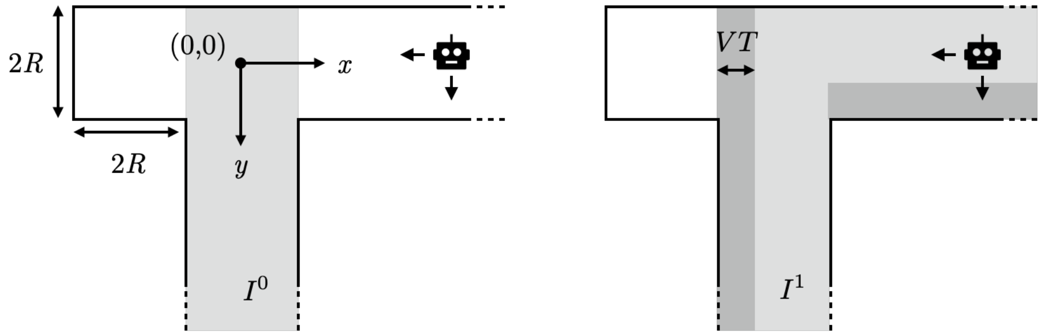


Figure 5.1: Robot navigating a corridor (Model 8). A 2D robot must navigate safely within a corridor with a dead-end without crashing against a wall. The corridor extends infinitely on the bottom and on the right. The robot can choose between going left and going down with a constant speed V. The left diagram shows I^0 in gray. The right diagram shows I^1 under the additional assumption $VT < 2R$ (I^1 and I^0 are otherwise equivalent). A darker shade of gray is used for regions of I^1 where only one of the two available actions is safe according to G^1.

constant velocity and must not crash against a wall. Computing I^0 gives us the vertical section of the corridor, in which going down is a safe one-step fallback. Computing I^1 forces us to distinguish two cases. If the corridor is wider than the maximal distance travelled by the robot in a control cycle ($VT > 2R$), then the upper section of the corridor is controllable (with the exception of a dead-end that we prove to be uncontrollable in Section 5.3.6). On the other hand, if the corridor is too narrow, then I^1 is equivalent to I^0. Formally, we have $I^1 \equiv (y > -R \wedge |x| < R) \vee (VT < 2R \wedge (x > -R \wedge |y| < R))$. Moreover, computing I^2 gives a result that is equivalent to I^1. From this, we can conclude that I^1 is equivalent to I^n for all $n \geq 1$. Intuitively, it is optimal with respect to *any* finite fallback strategy (restricted to permanent actions).

Model 8 Robot navigating a corridor with framed solutions of holes.

```
   assum | $V > 0 \wedge T > 0$
ctrlable | $\wedge\ \boxed{(y > -R \wedge |x| < R) \vee (VT < 2R \wedge (x > -R \wedge |y| < R))} \rightarrow [\{$
    ctrl |     $(\quad (?\boxed{x > -R + VT}\,;\ v_x := -V\,;\ v_y := 0)$
         |     $\cup\ (?\boxed{y < R - VT \vee x < R}\,;\ v_x := 0\,;\ v_y := V)\quad )\,;$
   plant |     $(t := 0\,;\ \{x' = v_x, y' = v_y, t' = 1\ \&\ t \leq T\})$
    safe | $\}^*]((x > -3R \wedge |y| < R) \vee (y > -R \wedge |x| < R))$
```

The controllable invariant unrolling I^n has a natural stopping criterion.

Lemma 5.7. *If $I^n \leftrightarrow I^{n+1}$ is valid for some $n \geq 0$, then $I^n \leftrightarrow I^m$ is valid for all $m \geq n$ and $I^n \leftrightarrow I^\omega$ is valid where $I^\omega \equiv [\mathsf{step}^\times\,;\ \mathsf{forever}]\,\mathsf{safe}$.*

5.3.6 Proving Optimality via the Dual Game

Suppose one found a controllable invariant I using techniques from the previous section. To prove it optimal, one must show that $\vDash \mathsf{assum} \rightarrow (I^{\,\mathrm{opt}} \rightarrow I)$. By contraposition and $[\alpha]\, P \leftrightarrow \neg\langle\alpha\rangle\, \neg P$ ([·]), this is equivalent to proving that:

$$\vDash \ \mathsf{assum} \wedge \neg I \rightarrow \underbrace{\langle((\cap_i\, \mathsf{act}_i)\,;\, \mathsf{plant})^*\rangle\, \neg\mathsf{safe}}_{\neg I^{\,\mathrm{opt}}}. \tag{5.6}$$

We define the largest *un*controllable region $U^{\,\mathrm{opt}} \equiv \neg I^{\,\mathrm{opt}}$ as the right-hand side of implication 5.6 above. Intuitively, $U^{\,\mathrm{opt}}$ characterizes the set of all states from which the environment (played by Angel) has a winning strategy against the controller (played by Demon) for reaching an unsafe state. In order to prove the optimality of I, we compute a sequence of increasingly strong approximations U of $U^{\,\mathrm{opt}}$ such that $U \rightarrow U^{\,\mathrm{opt}}$ is valid. We do so via an iterative process, in the spirit of how we approximate $I^{\,\mathrm{opt}}$ via bounded fallback unrolling (Section 5.3.5), although the process can be guided by the knowledge of I this time. If at any point we manage to prove that $\mathsf{assum} \rightarrow (I \vee U)$ is valid, then I is optimal.

One natural way to compute increasingly good approximations of $U^{\,\mathrm{opt}}$ is via loop unrolling. The idea is to improve approximation U by adding states from where the environment can reach U by running the control loop once, formally, $\langle(\cap_i\, \mathsf{act}_i)\,;\, \mathsf{plant}\rangle\, U$. This unrolling principle can be useful. However, it only augments U with new states that can reach U in time T at most. So it cannot alone prove optimality in cases where violating safety from an unsafe state takes an unbounded amount of time.

For concreteness, let us prove the optimality of I^0 in the case of Model 7. In [126] essentially the following statement is proved when arguing for optimality: $\vDash \mathsf{assum} \wedge \neg I^0 \rightarrow \langle(a := -B\,;\, \mathsf{plant})^*\rangle\, \neg\mathsf{safe}$. This is identical to our optimality criterion from Eq. (5.6), except that Demon's actions are restricted to braking. Intuitively, this restriction is sound since accelerating always makes things worse as far as safety is concerned. If the train cannot be saved with braking alone, adding the option to accelerate will not help a bit. In this work, we propose a method for formalizing such arguments within dGL to arbitrary systems.

Our idea for doing so is to consider a system made of two separate copies of our model. One copy has all actions available whereas the other is only allowed a single action (e.g. braking). Given a safety metric m (i.e. a term m such that $\vDash m \leq 0 \rightarrow \neg\mathsf{safe}$), we can then formalize the idea that "action i is always better w.r.t safety metric m" within this joint system.

Definition 5.5 (Uniform Action Optimality)**.** Consider a finite number of discrete dL programs α_i and $p \equiv \{x' = f(x)\ \&\ Q\}$. Let $V = \mathrm{BV}(p) \cup \bigcup_i \mathrm{BV}(\alpha_i)$ be the set of all variables written by p or some α_i. For any term θ and integer n, write $\theta^{(n)}$ for the term that results from θ by renaming all variables $v \in V$ to a fresh tagged version $x^{(n)}$. Using a similar notation for programs and formulas, define $p^{(1,2)} \equiv \{(x^{(1)})' = f(x^{(1)}), (x^{(2)})' = f(x^{(2)})\ \&\ Q^{(1)} \wedge Q^{(2)}\}$. We say that action j is *uniformly optimal* with respect to safety metric m if and only if:

$$\vDash \ m^{(1)} \geq m^{(2)} \rightarrow [{\alpha_j}^{(1)}\,;\, (\cup_i\, {\alpha_i}^{(2)})\,;\, p^{(1,2)}]\, m^{(1)} \geq m^{(2)}.$$

$\mathsf{best}_j((\alpha_i)_i, p, m)$ denotes that action j is uniformly optimal with respect to m for actions α_i and dynamics p.

With such a concept in hand, we can formally establish the fact that criterion Eq. (5.6) can be relaxed in the existence of uniformly optimal actions.

Theorem 5.4. *Consider a finite number of discrete* dL *programs* α_i *such that* $\vDash \langle \alpha_i \rangle$ *true for all* i *and* $p \equiv \{x' = f(x) \;\&\; q \geq 0\}$*. Then, provided that* $\mathsf{best}_j((\alpha_i)_i, p, m)$ *and* $\mathsf{best}_j((\alpha_i)_i, p, -q)$ *(no other action stops earlier because of the domain constraint), we have:*

$$\vDash \; \langle ((\cap \, \alpha_i) \, ; \, p)^* \rangle \, m \leq 0 \leftrightarrow \langle (\alpha_j \, ; \, p)^* \rangle \, m \leq 0 \;\; .$$

A general heuristic for leveraging Theorem 5.4 to grow U automatically works as follows. First, it considers $R \equiv \mathsf{assum} \wedge \neg I \wedge \neg U$ that characterizes states that are not known to be controllable or uncontrollable. Then, it picks a disjunct $\bigwedge_j R_j$ of the disjunctive normal form of R and computes a forward invariant region V that intersects with it: $V \equiv \bigwedge_j \{R_j \, : \, \mathsf{assum}, \, R_j \vdash [(\cup_i \, \mathsf{act}_i) \, ; \, \mathsf{plant}] \, R_j\}$. Using V as an assumption to simplify $\neg U$ may suggest metrics to be used with Theorem 5.4. For example, observing $\vDash V \rightarrow (\neg U \rightarrow (\theta_1 > 0 \wedge \theta_2 > 0))$ suggests picking metric $m \equiv \min(\theta_1, \theta_2)$ and testing whether $\mathsf{best}_j(\mathsf{act}, p, m)$ is true for some action j. If such a uniformly optimal action exists, then U can be updated as $U \leftarrow U \vee (V \wedge \langle (\mathsf{act}_j \, ; \, \mathsf{plant})^* \rangle \, m \leq 0)$. The solution I^1 for the corridor (Model 8) can be proved optimal automatically using this heuristic in combination with loop unrolling.

5.3.7 Implementing the Reduction Oracle

The CESAR algorithm assumes the existence of a *reduction oracle* that takes as an input a loop-free dGL formula and attempts to compute an equivalent formula within the fragment of propositional arithmetic. When an exact solution cannot be found, an implicant is returned instead and flagged appropriately (Def. 5.3). This section discusses our implementation of such an oracle.

As discussed in Section 5.3.4, exact solutions can be computed systematically when all ODEs are solvable by first using the dGL axioms to eliminate modalities (see Appendix D) and then passing the result to a *quantifier elimination algorithm* for first-order arithmetic [29, 150]. Although straightforward in theory, a naïve implementation of this idea hits two practical barriers. First, quantifier elimination is expensive and its cost increases rapidly with formula complexity [38, 160]. Second, the output of existing QE implementations can be unnecessarily large and redundant. In iterated calls to the reduction oracle, these problems can compound each other.

To alleviate this issue, our implementation performs *eager simplification* at intermediate stages of computation, between some axiom application and quantifier-elimination steps. This optimization significantly reduces output solution size and allows CESAR to solve a benchmark that would otherwise timeout after 20 minutes in 26s. Appendix A.4 further discusses the impact of eager simplification. Still, the doubly exponential complexity of quantifier elimination puts a limit on the complexity of problems that CESAR can currently tackle.

In the general case, when ODEs are not solvable, our reduction oracle is still often able to produce *approximate* solutions using differential invariants generated automatically by existing tools [142]. Differential invariants are formulas that stay true throughout the evolution of an ODE system. [1] To see how they apply, consider the case of computing $\mathsf{reduce}([\{x' = f(x)\}] \, P, A)$

---

[1] dGL provides ways to reason about differential invariants without solving the corresponding differential equa-

where P is the postcondition formula that must be true after executing the differential equation, and A is the assumptions holding true initially. Suppose that formula $D(x)$ is a differential invariant such that $D(x) \rightarrow P$ is valid. Then, a precondition sufficient to ensure that P holds after evolution is $A \rightarrow D(x)$. For a concrete example, Appendix A.2 shows how our reduction oracle computes the precondition for the dynamics of the `parachute` benchmark. It first uses the Pegasus tool [142] to identify a Darboux polynomial, suggesting an initial differential invariant D_0. Once we have D_0, the additional information required to conclude post condition P is $D_0 \rightarrow P$. To get an invariant formula that implies $D_0 \rightarrow P$, eliminate all the changing variables $\{x, v\}$ in the formula $\forall x \, \forall v \; (D_0 \rightarrow P)$, resulting in a formula D_1. D_1 is a differential invariant since it features no variable that is updated by the ODEs. Our reduction oracle returns $D_0 \wedge D_1$, an invariant that entails postcondition P. More details on our implementation of reduce and how it deals with ODEs in particular can be found in Appendix A.1.

5.3.8 The CESAR Algorithm

The CESAR algorithm for synthesizing control envelopes is summarized in Algorithm 1. It is expressed as a generator that yields a sequence of solutions with associated optimality guarantees. Possible guarantees include "*sound*" (no optimality guarantee, only soundness), "k-*optimal*" (sound and optimal w.r.t all k-switching fallbacks with permanent actions), "ω-*optimal*" (sound and optimal w.r.t all finite fallbacks with permanent actions) and "*optimal*" (sound and equivalent to S^{opt}). Line 11 performs the optimality test described in Section 5.3.6. Finally, Line 10 performs an important soundness check for the cases where an approximation has been made along the way of computing (I^n, G^n). In such cases, I is not guaranteed to be a controllable invariant and thus Case (2) of Def. 5.1 must be checked explicitly.

When given a problem with solvable ODEs and provided with a complete QE implementation within reduce, CESAR is guaranteed to generate a solution in finite time with an "n-*optimal*" guarantee at least (n being the unrolling limit).

5.4 Benchmarks and Evaluation

To evaluate our approach to the Control Envelope Synthesis problem, we curate a benchmark suite with diverse optimal control strategies. As Table 5.1 summarizes, some benchmarks have non-solvable dynamics, while others require a sequence of clever control actions to reach an optimal solution. Some have *state-dependent fallbacks* where the current state of the system determines which action is "safer", and some are drawn from the literature. We highlight a couple of benchmarks here. See Appendix A.3 for a discussion of the full suite and the synthesized results, and [71] for the benchmark files and evaluation scripts.

`Power Station` is an example where the optimal control strategy involves two switches, corresponding to two steps of unrolling. A power station can either produce power or dispense it to meet a quota, but never give out more than it has produced. Charging is the fallback action that is safe for all time *after* the station has dispensed enough power. However, to cover all

---

tion. For example, for an invariant of the form $e = 0$, the differential invariant axiom is $[\{x' = f(x)\}]\, e = 0 \leftrightarrow (e = 0 \wedge [\{x' = f(x)\}]\, e' = 0)$.

Algorithm 1 CESAR: Control Envelope Synthesis via Angelic Refinements

```
Input: a synthesis problem (as defined in Section 5.3.1), an unrolling limit $n$.
Remark: valid is defined as $\mathsf{valid}(F,\ A) \equiv (\texttt{first}(\mathsf{reduce}(\neg F, A)) = \text{false})$.
$k \leftarrow 0$
$I, e_I \leftarrow \mathsf{reduce}([\mathsf{forever}]\,\mathsf{safe},\ \mathsf{assum})$
while $k \leq n$ do
    $e_G \leftarrow$ true
    for each i do
        $G_i, e \leftarrow \mathsf{reduce}([\mathsf{act}_i\,;\ \mathsf{plant}]\,I,\ \mathsf{assum})$
        $e_G \leftarrow e_G$ and $e$
    if ($e_G$ and $e_I$) or $\mathsf{valid}(I \rightarrow \bigvee_i G_i,\ \mathsf{assum})$ then
        if $e_G$ and $\mathsf{optimal}(I)$ then
            yield $((I, G),$ "optimal")
            return
        else if $e_G$ and $e_I$ then yield $((I, G),$ "$k$-optimal")
        else yield $((I, G),$ "sound")
    $I', e \leftarrow \mathsf{reduce}(I \vee [\mathsf{step}]\,I,\ \mathsf{assum})$
    $e_I \leftarrow e_I$ and $e$
    if $e_G$ and $e_I$ and $\mathsf{valid}(I' \rightarrow I,\ \mathsf{assum})$ then
        yield $((I, G),$ "$\omega$-optimal")
        return
    $I \leftarrow I'$
    $k \leftarrow k + 1$
```

controllable states, we need to switch at least two times, so that the power station has a chance to produce energy and then dispense it, before settling back on the safe fallback. `Parachute` is an example of a benchmark with non-solvable, hyperbolic dynamics. A person jumps off a plane and can make an irreversible choice to open their parachute. The objective is to stay within a maximum speed that is greater than the terminal velocity when the parachute is open.

We implement CESAR in Scala, using Mathematica for simplification and quantifier elimination, and evaluate it on the benchmarks. Simplification is an art [80, 85]. We implement additional simplifiers with the `Egg` library [161] and SMT solver `z3` [39]. Experiments were run on a 32GB RAM M2 MacBook Pro machine. CESAR execution times average over 5 runs.

CESAR synthesis is automatic. The optimality tests were computed manually. Table 5.1 summarizes the result of running CESAR. Despite a variety of different control challenges, CESAR is able to synthesize safe and in some cases also optimal safe control envelopes within a few minutes. As an extra step of validation, synthesized solutions are checked by the hybrid system theorem prover KeYmaera X [52]. All solutions are proved correct, with verification time as reported in the last column of Table 5.1.

Table 5.1: Summary of CESAR experimental results

Benchmark	Synthesis Time (s)	Checking Time (s)	Optimal	Needs Unrolling	Non Solvable Dynamics
ETCS Train [126]	14	9	✓		
Sled	20	8	✓		
Intersection	49	44	✓		
Parachute [53]	46	8			✓
Curvebot	26	9			✓
Coolant	49	20	✓	✓	
Corridor	20	8	✓	✓	
Power Station	26	17	✓	✓	

5.5 Related Work

Hybrid controller synthesis has received significant attention [12, 91, 148], with popular approaches using temporal logic [5, 12, 164], games [108, 153], and CEGIS-like guidance from counterexamples [1, 36, 133, 143]. CESAR, however, solves the different problem of synthesizing control *envelopes* that strive to represent not one but *all* safe controllers of a system. Generating *valid* solutions is not an issue (a trivial solution always exists that has an empty controllable set). The real challenge is *optimality* which imposes a higher order constraint because it reasons about the relationship between possible valid solutions, and cannot, e.g., fit in the CEGIS quantifier alternation pattern $\exists\forall$. So simply adapting existing controller synthesis techniques does not solve symbolic control envelope synthesis.

Safety shields computed by numerical methods [2, 48, 82] serve a similar function to our *control envelopes* and can handle dynamical systems that are hard to analyze symbolically. However, they scale poorly with dimensionality and do not provide rigorous formal guarantees due to the need to discretize continuous systems. Compared to our symbolic approach, they cannot handle unbounded state spaces (e.g. our infinite corridor) nor produce shields that are parametric in the model's parameters without hopelessly increasing dimensionality.

On the optimality side, a systematic but manual process was used to design a safe European Train Control System (ETCS) and justify it as optimal with respect to specific train criteria [126]. Our work provides the formal argument filling the gap between such case-specific criteria and end-to-end optimality. CESAR is more general and automatic.

5.6 Conclusion

This chapter presents the CESAR algorithm for Control Envelope Synthesis via Angelic Refinements. It is the first approach to automatically synthesize symbolic control envelopes for hybrid systems. The synthesis problem and optimal solution are characterized in differential game logic. Through successive refinements, the optimal solution in game logic is translated into a controllable invariant and control conditions. The translation preserves safety. For the many cases where refinement additionally preserves optimality, an algorithm to test optimality of the result post translation is presented. The synthesis experiments on a benchmark suite of diverse control problems demonstrate CESAR's versatility.

Can CESAR be extended to handle more control shapes? How far does the idea of characterizing optimal control envelope solutions using game theory extend? Is it perhaps possible to solve control envelope synthesis for *all* the problems whose solutions are characterized by a hybrid game? Chapter 6 shows that the answer is yes. Since we become able to solve for *all* hybrid system sketches that mirror the structure of a game, it becomes convenient to treat *the hybrid game itself* as the specification of the shape of the control envelope. In the next chapter, we see control envelope synthesis for time-triggered control, event-triggered control and reach-avoid problems all handled by a single, extremely general framework.

Chapter 6

Generalized Hybrid Game Control Envelope Synthesis

Chapter 5 introduced control envelope synthesis for a class of hybrid systems by identifying a connection between the solution of a hybrid system sketch and hybrid games. In fact, the shape of control problems can be modeled directly by two-player hybrid games, which only indicate what moves players can make (what control actions are possible) without specifying the actual strategy that the players follow. This chapter synthesizes control envelopes for *all* dGL games, greatly extending the generality of control envelope synthesis.

6.1 Introduction

A hybrid *game* between the *controller* and the *environment* serves as a specification [104, 108, 153] for synthesis. The game models the *control possibilities* and the possible *environment behaviors*, with the controller winning if it meets the desired control contract. The synthesis task is to identify within the possible control space the control behaviors that win into the control contract. A control envelope corresponds to a *nondeterministic policy* in the specification game that indicates what actions a player can choose at any given decision point, such that *a player following it will never lose*, regardless of what the opponent does. Such a *nondeterministic* policy induces a *set* of control solutions because multiple actions are allowed at each decision point, and following each option leads to a *different* control solution. This chapter introduces *subvalue maps* as a symbolic, compositional representation of control envelopes that we can reason about recursively along the syntactic structure of the game. Via this representation we develop the theory to compare, verify, and synthesize symbolic control envelopes for hybrid games in their full generality.

Our formalization is in differential game logic (Chapter 2 [121]), which models two-player games with imperative program constructs like assignments, loops and branching. Subvalue map verification/synthesis composes at the level of such program constructs. This lets us leverage program analysis techniques like loop variants, invariants and their continuous analogs to solve control envelope synthesis. After defining subvalue maps, this chapter introduces the theory to verify that subvalue maps induce only policies that don't lose. To ensure that induced policies additionally never let the player get stuck, always allowing some action, the refined concept of *inductive* subvalue map is introduced.

We address control envelope synthesis. Since our ultimate goal is to use control envelopes to derive/check control actions, we consider a restricted category of inductive subvalue maps where checking if a given control action lies within the envelope is efficient (polynomial time). We present an algorithmic framework to synthesize such subvalue maps. Next, we formalize when subvalue maps are more permissive, permitting more control options, through a partial order. It is always possible to construct an optimal inductive subvalue map for a given game, establishing completeness.

The theoretical development needed is subtle. For example, at loops, a control envelope should allow *unboundedly* many repetitions so long as there is still a *finite* exit strategy *within* the same control envelope. The first few sections of the chapter build up the theoretical tools that make the theory for subvalue maps possible, and largely internalized within dGL. These include: 1. a dGL analog to Brzozowski derivatives [25] that let us step through a game and characterize gameplay after a given subgame (Section 6.4), and 2. dGL characterizations checking whether a set of states is reachable by following *some* strategy or *any* strategy in a given subvalue map (Section 6.5).

Subvalue maps are analogous to *value* functions in reinforcement learning [145]. A value function maps states to utility. Once it is known, an agent policy can be derived from it. Similarly, a player's subvalue map maps every subgame to a formula denoting a set of states from which that player can win the rest of the game (*sub*value because some maps map to a set of *known* winning states smaller than the theoretical maximum winning set). In dGL, *subgames* specify game decision points where at most one player makes a play. These subgames compose together per a recursive syntax to form large, arbitrarily complex games. A player's policy set can then be reconstructed by syntactically stepping through the game: at every subgame where the player must take a decision, it accepts only those decisions that reach the next subgame in a mapped winning state for that subgame per the subvalue map.

This chapter addresses control envelope synthesis for *all* of dGL: Synthesis for safety properties, liveness properties, and any combination of these are all solved by a single, natural framework without special handling of either. A special case of our technical development, which simply excludes subgames relating to continuous dynamics, solves the problem for purely discrete games. The flexibility of our final synthesis framework is demonstrated through examples with diverse control challenges, and performance is evaluated on benchmarks from the literature and a procedurally generated benchmark suite.

Our main contributions are to:

1. Introduce *inductive subvalue maps* as symbolic, composable representations of control envelopes for hybrid games (Section 6.6).
2. Present an algorithmic framework (Section 6.7) for *synthesis* of inductive subvalue maps where checking that control is within the induced nondeterministic policy is *computationally efficient*.
3. Order subvalue maps by permissiveness and prove completeness for optimal hybrid game inductive subvalue map synthesis via the construction of a *maximal subvalue map* (Section 6.8).

4. Evaluate the approach, demonstrating its application to representatives of various control theory problem classes (Section 6.10).

6.2 Differential Game Logic Background

This chapter synthesizes policies for games written in dGL. As discussed before (Chapter 2), a dGL game is played by two adversarial players, canonically called Angel and Demon. Various operators give either player opportunities to make control decisions to try to attain a winning condition, which is a dGL formula. *Unlike* Chapter 5, this chapter uses the convention that Angel is responsible for the controller's decisions, and wins when she maintains the desired system properties (e.g. safety), and Demon plays for the environment. dGL games are determined, meaning exactly one player has a winning strategy from any given state [121], and players use Markov strategies, relying only on the current state with no information about the strategy of the other player. dGL games are generated by a recursive grammar. In many of the game constructs, either Angel or Demon can make a choice, deciding how the game should proceed, and try to choose in a way that is most favorable to them. These decisions are represented using *actions*, which will later be used to keep track of state change, construct game trees and represent policies[1]. We recall the game constructs below and provide notation for *actions* that they correspond to. We also introduce explicit operators for some Demon-controlled actions (beyond the application of the dual operator shown in Chapter 2) that we use in this chapter for readability.

1. Angelic free assignment $x := *$ models Angel assigning any real value of her choice to variable x, and can be used, e.g., to model a controller's ranged choices. Angel's available actions are of the form $(x := e)$, where e is a (real polynomial) term. Dually, Demonic free assignment $x := \otimes$ models Demon assigning a real value of his choice to variable x. It models, e.g., environment disturbances that make control harder. Demon's available actions are of the form $(x := e)^d$, where e is a term.
2. Loop α^* runs α as many times as Angel wants. At the end of every iteration, Angel gets to choose whether to run the loop again or to exit it. This can model, e.g., a controller that repeatedly performs a task until a criterion is met. Angel's available actions are $\mathfrak{g}$ to *go* repeat the loop for one more iteration, or $\mathfrak{s}$ to *stop* and exit the loop. Dually, $\alpha^\times$ runs α as many times as Demon chooses. It can model a control loop that runs arbitrarily many times, and must stay safe forever. Demon's actions are $\mathfrak{g}^d$ to repeat the loop, or $\mathfrak{s}^d$ to exit it.
3. Branch choice $\alpha \cup \beta$ lets Angel choose to play either game α or game β. This can model a controller choosing between two modes of operation, for example accelerating or braking. Angel's available actions are $\mathfrak{l}$ to play *left* game α, or $\mathfrak{r}$ to play *right* game β. Dually, $\alpha \cap \beta$ lets Demon choose to play either game α or game β. This can model different modes of environment behavior, such as a car in the environment, that the controller must avoid colliding with, accelerating or braking. Demon's available actions are $\mathfrak{l}^d$ to play game α,

---

[1] The action notation presented here is a simplified form of the operational semantics of dGL [121][Appendix C]. Sequential compositions are given an action for uniformity but do not have an action in the original operational semantics.

or $\mathfrak{r}^d$ to play game β.

4. Continuous evolution $\{x' = f(x) \,\&\, Q\}$ modifies x per the solution of differential equation $x' = f(x)$. How long the ODE runs is determined by Angel, who must also maintain that domain constraint formula Q is true throughout. This can model, e.g., a control mode running till the controller interrupts. Angel's available actions are of the form $(x' = f(x) \,\&\, Q \mathbin{@} t)$, where t is a term representing the time Angel chooses to run the ODE for. Dually, in $\{x' = f(x) \,\&\, Q\}^d$, Demon chooses how long to run the ODE while ensuring that Q holds throughout. Demon's available actions are of the form $(x' = f(x) \,\&\, Q \mathbin{@} t)^d$, where t is the time Demon runs the ODE. This can model, e.g., control loop latency in controllers that periodically poll to take decisions. On omitting this construct and any cases it generates in definitions and theorems, we obtain results for purely discrete games.

Additionally, some game constructs do not involve Angel or Demon making any decisions.

1. Assignment $x := e$ assigns the expression e to variable x. The corresponding action that Angel performs to change state is written as $(x := e)$.
2. Sequential composition $\alpha; \beta$ runs game α followed by game β. The corresponding action that Angel performs is $\mathfrak{c}$ to start α.
3. Test $?Q$ makes Angel immediately lose the current game if formula Q is false, but has no effect if Q is true. The action representing Angel going through the test (and possibly losing) is $(?Q)$. Dually, $!Q$ makes Demon lose when Q is false. It can model physical environment constraints. Demon's only action, to take the test, is $(!Q)$.

As usual, state σ is a mapping from variables to real numbers, and $\sigma(x)$ is the real value associated with variable x. Eq. (6.1) shows the grammar of dGL hybrid games that we will use in this chapter. An alternative notation uses duals for Demon controlled constructs (Chapter 2), but for clarity in this chapter we use the explicit operators for some Demon controlled games shown below.

$$\begin{aligned} \alpha \equiv\ & x := e \mid \alpha; \beta \mid ?Q \mid \alpha^* \mid \alpha \cup \beta \mid x := * \mid \{x' = f(x) \& Q\} \\ & \mid !Q \mid \alpha^\times \mid \alpha \cap \beta \mid x := \otimes \mid \{x' = f(x) \& Q\}^d \end{aligned} \tag{6.1}$$

The *subgames* of dGL game α are the games that recursively compose to construct α, corresponding to the nodes of the abstract syntax tree induced by the grammar of Eq. (6.1).

We briefly describe the formal representation of actions and policies. Sections 6.4 and 6.5 will revisit these ideas with further detail. In any given initial state, a dGL game generates a game tree (Section 6.4.2) where each node corresponds to an action. A *play* of a game is a sequence of actions corresponding to a path from a root of the game tree to a leaf. Actions map old state to new state. For example, the action $(x := e)$ maps state σ to $\sigma(x \mapsto e)$, i.e., σ with value e replaced for x. Since agents and their gameplay are symmetric, in this chapter we often focus on the Angelic perspective without loss of generality.

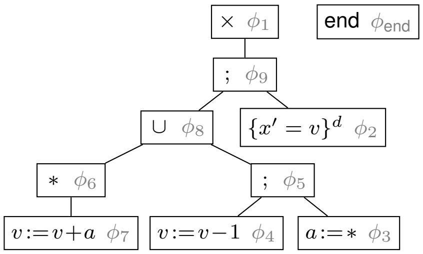
An Angelic (nondeterministic) *policy* is a function that maps every subgame and state to the set of actions that Angel can take at that subgame and state. *Henceforth in this chapter, policy will be used to mean nondeterministic policy*. Angel *follows* an Angelic policy if at every subgame she must take a decision, she plays one of the actions permitted by the policy. As usual, formulas can be interpreted as the set of states in which they are true. So, formula $\langle \alpha \rangle\, \phi$ represents the *winning region* of Angel for game α and winning condition ϕ. Notation $\models \phi$ means formula ϕ is valid (true in all states).

6.3 Overview

---

1. Input
Game: $(((v := v+a)^* \cup (v := v-1\,;\, a := *))\,;\, \{x' = v\}^d)^\times$ **Angel's Goal:** $x > 0$

2. Compute subvalue map



3. Policy for Angel derived from subvalue map

$\boxed{\cup}$	Taking the left (resp. right) branch is allowed if ϕ_6 (resp. ϕ_5) is true.
$\boxed{*}$	(Re-)entering the loop is allowed if ϕ_7 is true. Exiting is allowed if ϕ_2 is.
$\boxed{a := *}$	The value assigned to a must satisfy ϕ_2.

Computing the subvalue map:

$$\begin{aligned}
\textbf{Deduce } \phi_{\text{end}} &\equiv x > 0 && \text{(Goal)}\\
\textbf{Guess } \phi_1 &\equiv \big\langle (((v := v + a)^* \cup (v := v - 1\,;\, a := *))\,;\, \{x' = v\}^d)^\times \big\rangle\, x > 0 && \text{(Seek Invariant)}\\
&\Leftarrow \big\langle ((n{:=}*\,;?n{\geq}0\,;v{:=}v{+}an) \cup (v{:=}v{-}1\,;a{:=}*))\,;\{x'{=}v\}^d \big\rangle\, x{>}0 && \text{(Refinement)}\\
&\Leftrightarrow x > 0 \wedge (v \geq 0 \vee a > 0) && \text{(Axioms, QE)}\\
\textbf{Deduce } \phi_2 &\equiv \langle \{x' = v\}^d \rangle\, \phi_1 \Leftrightarrow \forall t\,(t \geq 0 \rightarrow \phi_1(x \mapsto x + vt)) && \text{(Solve ODE)}\\
&\Leftrightarrow x > 0 \wedge v \geq 0 && \text{(QE)}\\
\textbf{Deduce } \phi_3 &\equiv \exists a\, \phi_2 \Leftrightarrow x > 0 \wedge v \geq 0, \quad \phi_5 \equiv \phi_4 \equiv \phi_3(v{-}1 \mapsto v) \Leftrightarrow x > 0 \wedge v \geq 1 && \text{(Axioms, QE)}\\
\textbf{Guess } \phi_6 &\equiv \big\langle (v := v + a)^* \big\rangle\, \phi_2 \;\Leftrightarrow\; \big\langle n{:=}*\,;?n{\geq}0\,;v{:=}v{+}an \big\rangle\, \phi_2 && \text{(Invariant)}\\
&\Leftrightarrow x > 0 \wedge (v \geq 0 \vee a > 0) && \text{(QE)}\\
\textbf{Deduce } \phi_7 &\equiv \phi_6(v \mapsto v{+}a) \Leftrightarrow x > 0 \wedge (v + a \geq 0 \vee a > 0) &&\\
\textbf{Check } \phi_6 &\rightarrow \big\langle (?\phi_7\,;\, v := v + a)^*\,;\, ?\phi_2 \big\rangle \phi_2 \text{ valid} && \text{(Check Invariant)}\\
\textbf{Deduce } \phi_9 &\equiv \phi_8 \equiv \phi_6 \vee \phi_5 \Leftrightarrow x > 0 \wedge (v \geq 0 \vee a > 0) &&\\
\textbf{Check } \phi_1 &\rightarrow \phi_9 \wedge x > 0 \text{ valid} && \text{(Check Invariant)}
\end{aligned}$$

Figure 6.1: An *inductive subvalue map* for a simple hybrid game and the associated policy. The subvalue formulas are numbered in the order in which they are derived by the algorithm presented in Section 6.7. We will revisit this example throughout the chapter, providing more explanations.

To illustrate our new concept of an *inductive subvalue map*, let us consider the simple hybrid game from Fig. 6.1. In this game, Angel aims to ensure $x > 0$ after each iteration of a loop ($^\times$) controlled by Demon. At each iteration, Angel must choose ($\cup$) between two alternative paths, after which Demon continuously evolves x at rate v for a duration of his choice ($\{x' = v\}^d$). In the first path, Angel can increment v by a in a loop (*) as many times as she desires. In the second path, v is decremented by 1 and Angel gets an opportunity to arbitrarily set the value of a ($a := *$).

Our goal is to characterize not *one*, but *all* of the ways Angel can win (or at least *as many* as possible). At every choice point, we must determine which actions are compatible with Angel still winning the game and which actions are not. This way, Angel can pursue additional, unmodeled and possibly changing secondary goals without threatening her mission-critical objective of ensuring $x > 0$. In the traditional setting of control theory [10, 11] and reinforcement learning [145, 146, 147], this information is typically presented in the form of a *value function* that splits the state space into winning and losing states. This work proposes a representation of such value functions for hybrid games which is *sound*, *symbolic*, and which respects their *composable* structure.

Subvalue maps (Section 6.5, Section 6.6). A *subvalue map* associates *each* subgame of a hybrid game to a formula that *conservatively* estimates the set of states from which the overall game can be won, *starting from* this specific subgame. In the example from Fig. 6.1, ϕ_1 provides a sufficient initial condition for winning the game. More interestingly, ϕ_5 provides a sufficient condition for winning the subgame starting from the second Angel branch within the loop.

$$\vDash \ \phi_5 \ \rightarrow \ \big\langle v := v - 1 \,;\, a := * \,;\, \{x' = v\}^d \,;\, \alpha^{\times} \big\rangle \, x > 0 \tag{6.2}$$

where α stands for the full Demon loop body. At every control point for Angel, the subvalue map indicates which actions can be taken without forfeiting winning. For example, when reaching the $\cup$ decision point, Angel can safely take the left branch if ϕ_6 is true. Similarly, she can safely take the right branch if ϕ_5 is true. What if none of these hold though? Useful policies permit at least one action to be chosen. For this reason, Section 6.6 introduces the refined concept of an *inductive* subvalue map, whose local compatibility conditions ensure this property. In our example, these conditions mandate in particular that $\phi_8 \rightarrow \phi_6 \vee \phi_5$ is valid, ensuring that a safe action is always available to Angel at subgame $\cup$.

Internalizing policy reasoning (Section 6.5.2, 6.5.3). It is possible to reason about the policies represented by subvalue maps *within* dGL *itself*. The *universal projection* of a game onto a subvalue map constructs a dGL game that wins precisely when *all plays that respect the subvalue-map policy* are guaranteed to win. The *existential projection* of a game onto a subvalue map wins when *there exists some policy-compliant play* that wins. These projections let us use dGL to state and prove properties, keeping the theory tidy. The safety of the monitor derived from an inductive subvalue map that keeps the game on track for winning can similarly be expressed within dGL. In our example, the following formula is *guaranteed* to be valid:

$$\begin{aligned} \phi_1 \rightarrow \big\langle \big(\big(?(\phi_6 \vee \phi_5) \,;\, \big(!\phi_6 \,;\, ?\phi_6 \,;\, (!\phi_7 \,;\, v := v + a \,;\, ?\phi_6)^{\times} \,;\, !\phi_2 \big) \cap \\ \big(!\phi_5 \,;\, v := v - 1 \,;\, ?(\exists a \, \phi_2) \,;\, a := \otimes \,;\, !\phi_2 \big) \big) \,;\, \{x' = v\}^d \big)^{\times} \big\rangle \, x > 0. \end{aligned} \tag{6.3}$$

In the game above, every Angel choice from the original game has been transferred to Demon, on the condition that he complies with Angel's subvalue strategy monitor. Compliance is always possible, as ensured by proper Angel tests. A subtlety is the treatment of infinite gameplay.

Consider a car operating within a control envelope ensuring that it avoids obstacles and can eventually stop at a parking spot. The car should not be limited to only strategies with some fixed maximum number of moves so that it can travel arbitrarily long routes. Still, the car should be able to eventually park. Inductive subvalue maps enforce not *finite control* but the *existence of a finite path to victory*. They do not force the controller to take this finite path if it still has a way to win in the future, *allowing gameplay of unbounded length*.

Solving hybrid games via symbolic execution (Section 6.7). For practical applications, it is useful for subvalue maps to check if an action is acceptable at a given state *efficiently*. To compute such subvalue maps, with modality-and-quantifier-free subvalues, Section 6.7 presents an algorithmic framework. Like precondition calculus [40], the framework is parametrized on an invariant generation procedure. We propose *game rewriting* as a general principle for implementing this procedure (Section 6.7.1). Fig. 6.1 illustrates the resulting algorithm. The algorithm leverages the fact that any dGL formula that is free of loops and whose ODEs have polynomial solutions can be automatically rewritten into an equivalent modality-and-quantifier-free formula, by using the axioms of dGL along with quantifier elimination (QE) [150]. However, whenever a loop is hit, a guess is needed for a subvalue of the associated subgame. Such a guess is typically made by using heuristics to conservatively rewrite it into a loop-free dGL formula that can be symbolically evaluated. The algorithm then proceeds with the loop body, after which the validity of the guess is checked retrospectively. Section 6.7 provides more details.

Existence of an optimal subvalue map (Section 6.8). Subvalue maps can be ordered by how permissive their policy is. Section 6.8 formalizes such an ordering, where the main subtlety is comparing monitor guards *in the context in which they are reachable*. An optimal subvalue map can be constructed using dGL modalities. In our example, this solution maps ϕ_5 to the right-hand side of the implication in Eq. (6.2). The optimal subvalue map construction achieves completeness for the subvalue map synthesis problem. In practical subvalue maps like in Fig. 6.1, where modal subvalues are not permitted, the maps are synthesized using sound *lower* bounds of a game's value function. An *upper* bound can also be computed by considering the game from the opposing player's perspective. We can compare the dual subvalue maps and thereby symmetrically prove the optimality of the solution derived in Fig. 6.1. The proofs in this chapter largely follow from structural induction. *Appendix B.5 provides all proofs throughout this chapter.*

Remark 1. Assertion/invariant maps [49] for static analysis of imperative programs annotate every program point with a formula that must hold whenever execution reaches that program point. Analogously, an *Angelic subvalue map* annotates every subgame with a formula, and constrains Angel to play such that the formula must hold whenever execution reaches that subgame. Analogous to assertion maps [64], subvalue maps will be computed by backward symbolic execution.

Remarks, such as Remark 1 above, draw analogies to familiar concepts but are not critical to understanding the chapter.

6.4 Prefixes and Suffixes

An *Angelic subvalue map* maps every dGL subgame to a formula denoting Angel's known winning subregion for the rest of the game starting at that subgame. As it is not always possible to compute exact winning regions, the *known* winning *sub*regions in a subvalue map are conservative, safe subsets of the actual theoretical winning regions per dGL semantics. Fig. 6.1 shows an example where each subgame, corresponding to nodes of the syntax tree, is mapped to its corresponding winning subregion ($\phi_1, \cdots \phi_9$). In this example, the winning *sub*regions are maximal, *equal* to the winning regions.

This section develops the formalism necessary for defining subvalue maps. A subvalue map maps each subgame to a formula that must logically imply the winning region for playing the rest of the game *starting at that subgame*. To logically characterize this, we first introduce the *game suffix*, or game remainder of a subgame. The game suffix of a subgame features *all* possible future behaviors of the overall game from *any* point where an instance of the subgame is encountered. Surprisingly, we can characterize all these behaviors with a finite dGL game. Then the exact winning region starting from a subgame is defined as the winning region for the game suffix.

6.4.1 Game Suffix Construction

For disambiguation, each subgame is given a unique label. Notation $a{:}\alpha$ refers to the *named* game α where each subgame has a unique label, and the entire game α (corresponding to the root node of the abstract syntax tree) is labeled a. Even subgames that are identical in structure but appear at different places in the game are given different labels to distinguish multiple occurrences. $\mathsf{subgames}(a{:}\alpha)$ refers to the set of all named subgames of $a{:}\alpha$. Game suffix $a{:}\alpha_{b:}$ is a game modeling what remains to be played after seeing some occurrence of subgame $b{:}\beta$ while playing $a{:}\alpha$ (Def. 6.1)[2]. This is a complex idea: unboundedly many different plays of a game can lead to playing a given subgame b, and even within a single play, b can occur an unbounded number of times. The ability to succinctly characterize all suffix plays (leveraging dGL) is crucial to the definition of subvalue maps.

Example 6.1. Let the overall game of Fig. 6.1 be $a{:}\alpha$, and let the label of each subgame be the index i of the winning subregion formula ϕ_i shown on its corresponding node. Consider subgame 5 inside loop α. All execution traces after any instance of subgame 5 have the remainder of the loop body ($\{x' = v\}^d$) followed by potential future iterations of the loop (α). So the game suffix $a{:}\alpha_{5:}$ of subgame 5 is $\{x' = v\}^d\,;\,\alpha$.

Remark 2. The suffix characterizing behavior of a dGL game after a subgame *itself as a* dGL *game* is analogous to how Brzozowski derivatives reason about the behavior seen in a regular expression after a given string as a new regular expression [25].

Definition 6.1 (Game suffix)**.** The game suffix $a{:}\alpha_{b:}$ of subgame $b{:}\beta$ in $\mathsf{subgames}(a{:}\alpha)$ is con-

---

[2] The notation is analogous to Python list slicing. $a{:}\alpha_{b:}$ represents the part of $a{:}\alpha$ appearing *after* b. The complementary concept of *game prefix* is written as $a{:}\alpha_{:b}$, and consists of the part of the game appearing before subgame b.

structed as follows. If $a{:}\alpha$ is $b{:}\beta$ then $a{:}\alpha_{b:} = a{:}\alpha$. Otherwise, if $a{:}\alpha$ has structure:

$$a{:}(g{:}\gamma)^{*} \text{ or } a{:}(g{:}\gamma)^{\times} \text{ then } a{:}\alpha_{b:} = (g{:}\gamma_{b:})\ ;\ a{:}\alpha$$

$$a{:}(g{:}\gamma \cup d{:}\delta) \text{ or } a{:}(g{:}\gamma \cap d{:}\delta) \text{ then } \begin{cases} a{:}\alpha_{b:} = g{:}\gamma_{b:} & b \in \mathsf{subgames}(g{:}\gamma) \\ a{:}\alpha_{b:} = d{:}\delta_{b:} & b \in \mathsf{subgames}(d{:}\delta) \end{cases}$$

$$a{:}(g{:}\gamma\ ;\ d{:}\delta) \text{ then } \begin{cases} a{:}\alpha_{b:} = (g{:}\gamma_{b:})\ ;\ d{:}\delta & b \in \mathsf{subgames}(g{:}\gamma) \\ a{:}\alpha_{b:} = d{:}\delta_{b:} & b \in \mathsf{subgames}(d{:}\delta) \end{cases}$$

In the $a{:}(g{:}\gamma\ ;\ d{:}\delta)$ and $a{:}(g{:}\gamma \cup d{:}\delta)$ cases, b is either in $\mathsf{subgames}(g{:}\gamma)$ or $\mathsf{subgames}(d{:}\delta)$ because b is a subgame of $a{:}\alpha$ (and not a itself which is already handled initially). Cases where $a{:}\alpha$ is atomic, i.e., $\alpha \in \alpha \in \{x := e, x := *, ?Q, !Q, \{x' = f(x)\ \&\ Q\}, \{x' = f(x)\ \&\ Q\}^{d}\}$, b must be a, so these cases are already handled initially.

6.4.2 Game Trees and Plays

This section discusses dGL gameplay to precisely connect subgame suffixes to their defining property in dGL operational semantics. However, later in this chapter we will rely only on the syntactic suffix definition that has already been introduced. The background on dGL game trees is also useful for understanding *strategies*, which in Section 6.5 will provide a global view of the gameplay that a policy induces. This section first recounts which plays lie within a game and makes precise which suffix plays occur after a subgame. Then it defines a way to syntactically compare these play suffixes by the impact they have on state, *ignoring minor game structure differences*, to ensure that the *game suffix* has the same effect as these *suffix plays*, establishing that game suffixes reconstruct all gameplay that can occur after a subgame is seen. The definitions of game trees and plays that follow are from the existing operational semantics of dGL [121][Appendix C], extended to account for labels.

A labeled game $a{:}\alpha$, given initial state σ, corresponds to a *labeled game tree* $\mathcal{T}(a{:}\alpha, \sigma)$ where each node has a subgame label and an action. Per the conventions of descriptive set theory, the tree is a prefix-closed set of the paths originating at the root (Fig. 6.2).

Example 6.2. In Fig. 6.1 let the label of each subgame be the index i of the corresponding winning subregion formula ϕ_i. Consider subgame 6, the inner loop. The game tree for this subgame has the paths $6 \cdot \mathfrak{s}$ for 0 iterations, $6 \cdot \mathfrak{g} \cdot 7 \cdot (v := v + a) \cdot 6 \cdot \mathfrak{s}$ for one iteration, and so on. Operator $\cdot$ concatenates actions/sequences of actions. Fig. 6.2 visually represents this tree.

Def. B.2 in Appendix B.6.2 shows the full construction of trees. [3]. A path in a tree can be represented as the sequence of labels and actions encountered while traversing the path, where the subgame label appears immediately before the action that subgame corresponds to. Because of loops, subgame labels can repeat even within a single path. For example, in Fig. 6.2 because of the Angelic loop, labels 6 and 7 repeat.

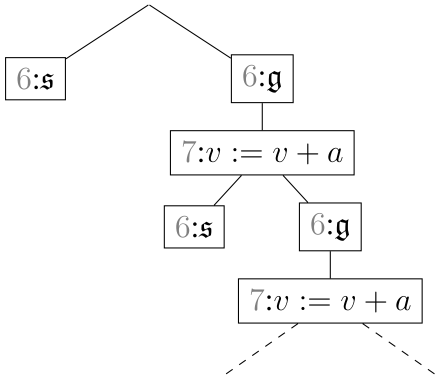
Tree nodes have actions, which correspond to transformations from old game state before the action to new state [121][Appendix C]. Appendix B.6.1 recalls the transformations. Notation

---

[3] This is similar to the standard dGL operational semantics [121, Appendix C], but with the addition of labels to keep track of subgames and some notation changes.

$$
\begin{aligned}
&\text{Paths} \equiv \{\\
&\quad (6{:}\,\mathfrak{s}), (6{:}\,\mathfrak{g}),\\
&\quad (6{:}\,\mathfrak{g} \cdot 7{:}\,v := v{+}a),\\
&\quad (6{:}\,\mathfrak{g} \cdot 7{:}\,v := v{+}a \cdot 6{:}\,\mathfrak{s}),\\
&\quad (6{:}\,\mathfrak{g} \cdot 7{:}\,v := v{+}a \cdot 6{:}\,\mathfrak{g} \cdot 7{:}\,v := v{+}a),\\
&\quad \cdots\}
\end{aligned}
$$

(a) Representation of game tree as (prefix-closed) set of paths through the tree.



(b) Visualization of the labeled game tree

Figure 6.2: The game tree for subgame 6 of the game in Fig. 6.1, i.e., $6{:}(7{:}v := v + a)^*$

$\lfloor \mathsf{act} \rfloor_\sigma$ is the state reached by running action act starting at state σ, e.g., $\lfloor (x := e) \rfloor_\sigma = \sigma(x \mapsto e)$ For tests and ODEs, it is possible for the transition to be undefined, e.g., for tests:

$$
\lfloor (?Q) \rfloor_\sigma = \begin{cases} \sigma & \text{if } \sigma \in [\![Q]\!] \\ \text{not defined} & \text{otherwise} \end{cases} \qquad \lfloor (!Q) \rfloor_\sigma = \begin{cases} \sigma & \text{if } \sigma \in [\![Q]\!] \\ \text{not defined} & \text{otherwise} \end{cases}
$$

In this case, the player responsible for the action loses. Labels, along with some actions whose only role is to restrict tree structure, have no impact on state, corresponding to the identity transformation. The set $\mathfrak{S}$ of such actions is $\{\mathfrak{g}, \mathfrak{s}, \mathfrak{l}, \mathfrak{r}, \mathfrak{c}, \mathfrak{g}^d, \mathfrak{s}^d, \mathfrak{l}^d, \mathfrak{r}^d\}$.

A path is executed by inductively applying the actions: $\lfloor a \cdot t \rfloor_\sigma = \lfloor t \rfloor_{\lfloor a \rfloor_\sigma}$. We syntactically compare paths and trees per the reachable states resulting from gameplay, ignoring structural differences, *by erasing the nodes that have no impact on state*. This lets us formalize how subgame suffixes preserve reachable states, but not structure. The action projection operator $\pi_{\neg\mathfrak{S}}$ returns the path with all labels and actions belonging to $\mathfrak{S}$ removed (Appendix B.6.6, Def. B.8). For paths p_1 and p_2, if $\pi_{\neg\mathfrak{S}}(p_1) = \pi_{\neg\mathfrak{S}}(p_2)$, then $\lfloor p_1 \rfloor_\sigma = \lfloor p_2 \rfloor_\sigma = \lfloor \pi_{\neg\mathfrak{S}}(p_1) \rfloor_\sigma$. $\pi_{\neg\mathfrak{S}}$ is overloaded to apply to game trees. For game tree t, $\pi_{\neg\mathfrak{S}}(t)$ is a game tree where all paths have only actions that can impact state, produced by applying $\pi_{\neg\mathfrak{S}}$ to every path in t. Two trees result in the same reachable states (even if their structure differs) if their projections are the same. We can now show the defining property of game suffixes.

Lemma 6.1 (Game Suffixes are Operational Gameplay Suffixes)**.** *For subgame $b{:}\beta$ in overall game $a{:}\alpha$, let the game suffix be $a{:}\alpha_{b:}$. Let σ be some state. The states reachable from playing $a{:}\alpha$ such that σ is reached at subgame $b{:}\beta$ at some point during the gameplay are the same as the states reachable from playing $a{:}\alpha_{b:}$ starting in σ. That is,*

$$
\{\pi_{\neg\mathfrak{S}}(s) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma'), \lfloor p \rfloor_{\sigma'} = \sigma \textit{ and } \pi_{\neg\mathfrak{S}}(s) \neq ()\} = \pi_{\neg\mathfrak{S}}(\mathcal{T}(a{:}\alpha_{b:}, \sigma)),
$$

where $()$ is the empty path.

In Lemma 6.1, the left-hand side considers all paths in the original game that contain b. These must be of the form $p{\cdot}b{\cdot}s$ where p is a prefix of b and s is a suffix. A suffix set is built by collecting the projection of all such suffixes, ignoring sequences that are empty after applying $\pi_{\neg\mathfrak{S}}$. The quantity on the right-hand side is the projection of the game tree of the subgame suffix at state σ. Suffix $a{:}\alpha_{b:}$ captures all remaining plays occurring after any occurrence of $b{:}\beta$ while playing $a{:}\alpha$, ignoring the impactless actions of $\mathfrak{G}$.

6.4.3 Game Prefix

The complementary concept of a *game prefix* computes *what contexts* a subgame can be seen in. For example, it will help to show that Angel, while following a subvalue map, only reaches a subgame within the region from which she can win the rest of the game (Section 6.5). The prefix of subgame b in the game $a{:}\alpha$, written $a{:}\alpha_{:b}$, features *all* behaviors of game α that can happen *before* an occurrence of subgame b within $a{:}\alpha$. Constructed analogously to the game suffix (Appendix B.6.8, Def. B.10), it satisfies an analogous property (Lemma 6.2).

Lemma 6.2 (Execution Prefixes are Original Gameplay Prefixes)**.** *For subgame $b{:}\beta$ in $a{:}\alpha$, for any initial state σ, the states reachable at $b{:}\beta$ while playing $a{:}\alpha$ are equal to those reachable by playing game prefix $a{:}\alpha_{:b}$. That is, prefix set $\{\pi_{\neg\mathfrak{G}}(p) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma)\}$ is equal, under prefix closure, to prefix game tree $\pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha_{:b}, \sigma))$*[4].

6.5 Subvalue Maps

We define subvalue maps, discuss their interpretation as policies, and these policies' properties.

6.5.1 Subvalue Map Definition

Def. 6.2 characterizes a subvalue map. Each subgame maps to a formula that must logically imply the winning region for playing the rest of the game starting at that subgame. The exact winning region starting from a subgame is defined as the winning region for the game suffix. A special end subgame additionally stores the terminal condition that the policy induced by the subvalue map tries to reach. Mathematically, when Angel is playing the overall game $a{:}\alpha$ with winning condition $S(\mathsf{end})$, her winning region at the point when she is about to play subgame $b{:}\beta$ to achieve $S(\mathsf{end})$ overall is $\langle a{:}\alpha_{b:}\rangle\, S(\mathsf{end})$.

Definition 6.2 (Subvalue maps)**.** A map S from the subgames of game $a{:}\alpha$ and special subgame end to formulas is an *Angelic subvalue map* of $a{:}\alpha$ when $\models S(b) \to \langle a{:}\alpha_{b:}\rangle S(\mathsf{end})$ for every subgame $b{:}\beta$ in $\mathsf{subgames}(a{:}\alpha)$. Dually, S is a *Demonic subvalue map* for game $a{:}\alpha$ when $\models S(\beta) \to [a{:}\alpha_{b:}]S(\mathsf{end})$ for every subgame $b{:}\beta$ in $\mathsf{subgames}(a{:}\alpha)$.

Interpretation as policy. Angelic subvalue maps provide Angel with a policy that describes how to play the game. For example, the map in Fig. 6.1 says that Angel at her choice $\cup$ should not transition to $(v := v + a)^*$ when ϕ_6 is false, and not transition to $v := v - 1\,;\, a := *$ when ϕ_5 is false. Intuitively, at any subgame where Angel has a choice (e.g. Fig. 6.1 at $\cup$), the *next*

---

[4] The prefix closure is required to recover tree structure. Unlike with the suffix construction, the prefix set builder collects only *parent* subgames of $b{:}\beta$ and *not all predecessors*. Expanding the prefix closure results in the condition: $\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma) \text{ and } p = t \cdot u\} = \pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha_{:b}, \sigma))$

subgame that she chooses to run (e.g., left branch $(v := v + a)^*$) should be such that the current state σ satisfies the winning subregion for that subsequent subgame (e.g., $\sigma \models \phi_6$). This next subgame to run after the current one ends is called the *successor* (Def. 6.3).

Definition 6.3 (Successor)**.** The successor of subgame ${\scriptstyle b:}\beta$ within the overall game ${\scriptstyle a:}\alpha$, written $\mathsf{succ}(b, {\scriptstyle a:}\alpha)$, is defined as follows. If ${\scriptstyle a:}\alpha$ is ${\scriptstyle b:}\beta$ then $\mathsf{succ}(b, {\scriptstyle a:}\alpha)$ is the special subgame end. Otherwise, if ${\scriptstyle a:}\alpha$ has structure:

$${\scriptstyle a:}({\scriptstyle g:}\gamma \,;\, {\scriptstyle d:}\delta) \text{ then}$$

$$\begin{cases} \mathsf{succ}(b, {\scriptstyle a:}\alpha) = {\scriptstyle d:}\delta & b \in \mathsf{subgames}({\scriptstyle g:}\gamma) \text{ and } \mathsf{succ}(b, {\scriptstyle g:}\gamma) = \mathsf{end} \\ \mathsf{succ}(b, {\scriptstyle a:}\alpha) = \mathsf{succ}(b, {\scriptstyle g:}\gamma) & \text{otherwise if } b \in \mathsf{subgames}({\scriptstyle g:}\gamma) \\ \mathsf{succ}(b, {\scriptstyle a:}\alpha) = \mathsf{succ}(b, {\scriptstyle d:}\delta) & \text{otherwise} \end{cases}$$

$${\scriptstyle a:}({\scriptstyle g:}\gamma)^* \text{ or } {\scriptstyle a:}({\scriptstyle g:}\gamma)^\times \text{ then}$$

$$\begin{cases} \mathsf{succ}(b, {\scriptstyle a:}\alpha) = a & b \in \mathsf{subgames}({\scriptstyle g:}\gamma) \text{ and } \mathsf{succ}(b, {\scriptstyle g:}\gamma) = \mathsf{end} \\ \mathsf{succ}(b, {\scriptstyle a:}\alpha) = \mathsf{succ}(b, {\scriptstyle g:}\gamma) & \text{otherwise if } b \in \mathsf{subgames}({\scriptstyle g:}\gamma) \end{cases}$$

$$\begin{matrix} {\scriptstyle a:}({\scriptstyle g:}\gamma \cup {\scriptstyle d:}\delta) \text{ or} \\ {\scriptstyle a:}({\scriptstyle g:}\gamma \cap {\scriptstyle d:}\delta) \end{matrix} \text{ then } \begin{cases} \mathsf{succ}(b, {\scriptstyle a:}\alpha) = \mathsf{succ}(b, {\scriptstyle g:}\gamma) & b \in \mathsf{subgames}({\scriptstyle g:}\gamma) \\ \mathsf{succ}(b, {\scriptstyle a:}\alpha) = \mathsf{succ}(b, {\scriptstyle d:}\delta) & \text{otherwise} \end{cases}$$

Using the successor definition, Def. 6.4 shows how in general, Angel can interpret the subvalue map as a (nondeterministic) policy indicating what control decisions are safe to take to eventually win based on the current state[5].

Definition 6.4 (Policy Interpretation)**.** *Policy interpretation function* $\mathcal{P}_{a:\alpha}(S)$ transforms an Angelic subvalue map S for game ${\scriptstyle a:}\alpha$ to a nondeterministic policy function that maps current state σ and subgame ${\scriptstyle b:}\beta$ to the set of acceptable control actions Angel can take. $\mathcal{P}_{a:\alpha}(S)(b, \sigma)$ is defined as below. When the current subgame ${\scriptstyle b:}\beta$ has structure:

$${\scriptstyle b:}({\scriptstyle d:}\delta \cup {\scriptstyle g:}\gamma) \text{ then } \begin{cases} \{\mathfrak{l}, \mathfrak{r}\} & \sigma \models S(d) \wedge S(g) \\ \{\mathfrak{l}\} & \text{else if } \sigma \models S(d) \\ \{\mathfrak{r}\} & \text{else if } \sigma \models S(g) \\ \emptyset & \text{otherwise} \end{cases}$$

$${\scriptstyle b:}({\scriptstyle g:}\gamma)^* \text{ then } \begin{cases} \{\mathfrak{s}, \mathfrak{g}\} & \sigma \models S(g) \wedge S(\mathsf{succ}(b, {\scriptstyle a:}\alpha)) \\ \{\mathfrak{g}\} & \text{else if } \sigma \models S(g) \\ \{\mathfrak{s}\} & \text{else if } \sigma \models S(\mathsf{succ}(b, {\scriptstyle a:}\alpha)) \\ \emptyset & \text{otherwise} \end{cases}$$

$${\scriptstyle b:}x := * \text{ then } \{(x := e) \mid \sigma(x \mapsto e) \models S(\mathsf{succ}(b, {\scriptstyle a:}\alpha))\}$$

$$\begin{aligned} {\scriptstyle b:}\{x' = f(x) \,\&\, Q\} \text{ then } \{&(x' = f(x) \,\&\, Q \,@\, t) \;:\; t \geq 0, \forall s \in [0, t], \varphi(s) \models Q \text{ and} \\ &\varphi(t) \models S(\mathsf{succ}(b, {\scriptstyle a:}\alpha)) \text{ where } \varphi : [0, t] \to \mathcal{S} \text{ is differentiable,} \\ &\varphi(0) = \sigma, \text{ and } \forall s \in [0, t],\ \varphi(s) \models x' = f(x) \wedge Q\} \end{aligned}$$

---

[5] The successor is not just the next label to occur in the game suffix. This is because the successor should not be a child of the current subgame. It should only start when the current subgame ends.

Relationship with strategies. We have focused on *policies* which provide local answers for what action to play next. *Strategies*, on the other hand, are global gameplay solutions. The Angelic strategy at a given initial state corresponds to a game tree at that state that has been pruned so that at every subgame where Angel has to take a decision, she takes exactly one decision. Thus, a strategy plans out all the future moves of its player for all potential actions of the opponent for all parts of the game at once. Viewing policies from the lens of strategies is useful for understanding the global behaviors that a policy induces, and will be relevant in Section 6.5.2 and Section 6.5.3. The set of strategies that Angel can play for the game $a{:}\alpha$ while following the policy of a subvalue map S starting in a state σ is written as $\mathcal{S}_{a:\alpha}(S)(\sigma)$. The construction is shown in Appendix B.6.6, Def. B.7. The intuition is to generate the game tree while restricting Angel to play per the policy. When the policy has multiple options at a given Angelic decision point, $\mathcal{S}_{a:\alpha}(S)(\sigma)$ has different strategies for each way to proceed. When the policy is deterministic, the set contains a single strategy.

Winning games. Subvalue maps induce policies that are suitable to play games with *compatible* winning conditions, i.e., winning conditions that contain the terminal subregion of the policy that end maps to (Def. 6.5).

Definition 6.5 (Compatible winning conditions)**.** An Angelic subvalue map S for game $a{:}\alpha$ is said to be *compatible with Angel winning condition* ϕ when $\models S(\mathsf{end}) \rightarrow \phi$. Dually, a Demonic subvalue map S for game $a{:}\alpha$ is compatible with Demon winning condition ϕ when $\models S(\mathsf{end}) \rightarrow \phi$.

If Angel plays a game for a compatible winning condition per a policy induced by an Angelic subvalue map, she always remains in a state where there *exists* a way for her to win the remaining game (Theorem 6.1 at the end of this section). However, the policy she needs to play to win *might not lie within the subvalue map*.

For example, in Fig. 6.1 at Angel choice $\cup$, consider the subvalue map in the figure except that ϕ_6 is $\bot$ instead of $x > 0 \wedge (v \geq 0 \vee a > 0)$. This is still an Angelic subvalue map. If we arrive at ϕ_8 in state $\sigma \equiv \{x = 0, v = 0, a = 0\}$, then Angel can win by choosing the left branch guarded by ϕ_6, but will lose if she chooses the right branch. The policy induced by the map at this point does not let Angel into the losing play of choosing the right branch. However it does not let her choose the left branch either since $\sigma \not\models \bot$. Its returned action set is $\emptyset$. Angel is stuck. This motivates the definition of an *inductive* subvalue map (Section 6.6), where not only does the policy keep Angel in a state where she has a way to win, but this victory can be achieved by continuing to follow the subvalue map policy.

We next formulate within dGL the formalism to say *for all ways to play the policy of the subvalue map* (Section 6.5.3) up to any subgame, *there exists a way to continue to play within the subvalue map* (Section 6.5.2).

6.5.2 Existential Subvalue Map Projection

The *existential projection* of game $a{:}\alpha$ onto Angelic subvalue map S, written $a{:}\alpha \underset{\downarrow}{\exists} S$, is the game that can be won exactly when there *exists* a way for Angel to win $a{:}\alpha$ *while restricted to stay within the (nondeterministic) policy of* S. Projection changes the game so that Angel is put on rails, limited to the policy induced by S while taking every decision, and is forced to manifest what she must do to win while playing within the policy.

Lemma 6.3 (Existential projection correspondence)**.** *For any Angelic subvalue map S for game $a{:}\alpha$ and compatible winning condition ϕ, in initial state $\sigma \models \langle\alpha\rangle\,\phi$, Angel has a winning strategy for the game $a{:}\alpha \mathbin{\overset{\exists}{\downarrow}} S$ if and only if Angel can win $\langle\alpha\rangle\,\phi$ while following the policy of S. That is, $\sigma \models \langle a{:}\alpha \mathbin{\overset{\exists}{\downarrow}} S\rangle\,\phi$ iff $\mathcal{S}_{a:\alpha}(S)(\sigma) \neq \emptyset$.*

Def. 6.6 defines existential projection by inserting Angelic tests to guard *every* Angelic decision that would make Angel lose if her choice would result in playing into the subsequent subgame in a state outside that subgame's winning subregion. For example, in Fig. 6.1, the subvalue map says that ϕ_2 is the winning subregion of the subgame that is played after subgame $3{:}a := *$. The projection of the subvalue map onto subgame 3 is $a := *\,;\ ?\phi_2$, thus *locally* ensuring that Angel makes a correct assignment to a that will let Angel win the rest of the game. When Angel has no decisions to take, e.g., $g{:}\gamma \cap d{:}\delta$ where *Demon* chooses between $g{:}\gamma$ and $d{:}\delta$, projection does not interfere, only recursively adding guards for any Angelic decisions within subgames γ and δ. During recursive calls, we sometimes have to update the end subvalue to reflect the appropriate terminal condition for the subgame under consideration. Notation $S(\mathsf{end} \mapsto Q)$ refers to S with the subvalue of end updated to Q.

Definition 6.6 (Existential projection)**.** The *existential projection* of dGL game $a{:}\alpha$ onto Angelic subvalue map S, written $a{:}\alpha \mathbin{\overset{\exists}{\downarrow}} S$, is generated recursively from the structure of $a{:}\alpha$ as follows. If $a{:}\alpha$ has structure:

$$a{:}x := * \text{ then } x := *\,;\ ?S(\mathsf{end}). \qquad a{:}(g{:}\gamma \cup d{:}\delta) \text{ then } (?S(g)\,;\ g{:}\gamma \mathbin{\overset{\exists}{\downarrow}} S) \cup (?S(d)\,;\ d{:}\delta \mathbin{\overset{\exists}{\downarrow}} S)$$

$$a{:}(g{:}\gamma)^* \text{ then } \big(?S(g)\,;\ (g{:}\gamma \mathbin{\overset{\exists}{\downarrow}} (S(\mathsf{end} \mapsto S(a))))\big)^*\,;?S(\mathsf{end})$$

$$a{:}(g{:}\gamma;\, d{:}\delta) \text{ then } (g{:}\gamma \mathbin{\overset{\exists}{\downarrow}} (S(\mathsf{end} \mapsto S(d))))\ ;\ (d{:}\delta \mathbin{\overset{\exists}{\downarrow}} S)$$

$$a{:}(g{:}\gamma \cap d{:}\delta) \text{ then } g{:}\gamma \mathbin{\overset{\exists}{\downarrow}} S \cap d{:}\delta \mathbin{\overset{\exists}{\downarrow}} S \qquad a{:}(g{:}\gamma)^\times \text{ then } \big(g{:}\gamma \mathbin{\overset{\exists}{\downarrow}} (S(\mathsf{end} \mapsto S(a)))\big)^\times$$

$$a{:}\{x' = f(x)\ \&\ Q\} \text{ then } \{x' = f(x)\ \&\ Q\}\,;\ ?S(\mathsf{end})$$

atomic and not controlled by Angel, i.e.,

$$\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x)\ \&\ Q\}^d\}, \text{ then } \alpha$$

where game labels are preserved and fresh labels are used for newly introduced subgames. The existential projection of a Demonic subvalue map is symmetric (Appendix B.6.4), with the addition of Demonic assertions after subgames controlled by Demon.

Remark 3. $a{:}\alpha \mathbin{\overset{\exists}{\downarrow}} S$ is an *Angelic refinement* of $a{:}\alpha$, that is, for any winning condition ψ, $\models \langle a{:}\alpha \mathbin{\overset{\exists}{\downarrow}} S\rangle\,\psi \rightarrow \langle\alpha\rangle\,\psi$ (Lemma B.2). Intuitively, if Angel has a winning strategy within S, she can win with the same strategy in the unconstrained game.

6.5.3 Universal Subvalue Map Projection

The *universal projection*, written $a{:}\alpha \mathbin{\overset{\forall}{\downarrow}} S$, explores a subvalue map maximally flexibly by making Demon play Angel's policy in Angelic subvalue map S to look for loopholes. It changes the game to put *Demon* on rails forcing him to play within Angel's S, as he looks for *any* strategy in the policy that falsifies Angel's winning condition.

Lemma 6.4 (Universal projection correspondence)**.** *For any Angelic subvalue map S for game $a{:}\alpha$, in initial state $\sigma \models \langle\alpha\rangle\, a$, for compatible winning condition ϕ, Angel has a winning strategy for the game $a{:}\alpha \underset{\downarrow}{\forall} S$ if and only if all ways to pursue the policy of S for game $a{:}\alpha$ that complete the game end in Angel's winning region. That is, $\sigma \models \langle a{:}\alpha \underset{\downarrow}{\forall} S\rangle\, \phi$ iff all elements of set $\mathcal{S}_{a:\alpha}(S)(\sigma)$ are Angel winning strategies.*

Def. 6.7 defines universal projection by replacing all Angelic games ($\cup, x := *, \alpha^*, \{x' = f(x)\&Q\}$) by their dual Demonic games and inserting Demonic tests to ensure Demon plays these decisions within Angel's control envelope. Like in projection, control choices that were already Demon's remain unguarded. Eq. (6.3) shows universal projection for the example subvalue map and game in Fig. 6.1.

Definition 6.7 (Universal projection)**.** The *universal projection* of Angelic subvalue map S onto dGL game $a{:}\alpha$, written $a{:}\alpha \underset{\downarrow}{\forall} S$, is generated recursively per the structure of $a{:}\alpha$ as follows. If $a{:}\alpha$ has structure:

$a{:}x := *$, then $x := \otimes\,;\, !S(\mathsf{end})$. $\quad a{:}(g{:}\gamma;\, d{:}\delta)$, then $(g{:}\gamma \underset{\downarrow}{\forall} (S(\mathsf{end} \mapsto S(d))))\ ;\ (d{:}\delta \underset{\downarrow}{\forall} S)$.

$a{:}(g{:}\gamma \cup d{:}\delta)$, then $(!S(g)\,;\, g{:}\gamma \underset{\downarrow}{\forall} S) \cap (!S(d)\,;\, d{:}\delta \underset{\downarrow}{\forall} S)$.

$a{:}(g{:}\gamma)^*$, then $(!S(g)\,;\, (g{:}\gamma \underset{\downarrow}{\forall} (S(\mathsf{end} \mapsto S(a)))))^{\times}\ ;\ !S(\mathsf{end})$.

$a{:}(g{:}\gamma \cap d{:}\delta)$, then $(g{:}\gamma \underset{\downarrow}{\forall} S) \cap (d{:}\delta \underset{\downarrow}{\forall} S)$. $\quad a{:}(g{:}\gamma)^{\times}$, then $(g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))^{\times}$.

$a{:}\{x' = f(x)\ \&\ Q\}$, then $\{x' = f(x)\ \&\ Q\}^{d}\,;\, !S(\mathsf{end})$.

atomic and not controlled by Angel, i.e.,

$\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x)\ \&\ Q\}^{d}\}$, then α.

where game labels are preserved (including for Angel-controlled subgames transformed to their Demon-controlled equivalent) and fresh labels are used for newly introduced subgames. The universal projection of a Demonic subvalue map is symmetric (see Appendix B.6.5).

We can now show the defining property of subvalue maps.

Theorem 6.1 (Subvalue Map stays in Winning Region)**.** *Suppose S is an Angelic subvalue map for game $a{:}\alpha$ compatible with winning condition ϕ, i.e., for every subgame $b \in \mathsf{subgames}(a{:}\alpha)$, $\models S(b) \rightarrow \langle a{:}\alpha_{b:}\rangle\, S(\mathsf{end})$ and $\models S(\mathsf{end}) \rightarrow \phi$. Upon starting in any state $\sigma \in [\![S(a)]\!]$, and reaching subgame b by following any strategy induced by the subvalue map S, there exists a winning strategy for Angel to win the remainder of the game. That is, $S(a) \models \langle (a{:}\alpha \underset{\downarrow}{\forall} S)_{:b}\rangle\, \langle a{:}\alpha_{b:}\rangle\, \phi$.*

As discussed in Section 6.5.1, the property of Theorem 6.1 is critical but not alone strong enough, since it does not guarantee that following the policy will ensure Angel wins. The property it maintains is only that Angel always continues to have some winning strategy ($\langle a{:}\alpha_{b:}\rangle\, \phi$), not that such a strategy is within the policy ($\langle a{:}\alpha \underset{\downarrow}{\exists} S_{b:}\rangle\, \phi$). Angel might get stuck without a valid choice permitted by the map S. The *inductive* subvalue map introduced next prevents this situation, disqualifying subvalue maps where Angel gets stuck.

6.6 Inductive Subvalue Maps

To correct for the shortcoming of subvalue maps by preventing players from getting stuck, we introduce *inductive* subvalue maps. Notation $a{:}\alpha \models S$ indicates S is an *inductive* subvalue map for game $a{:}\alpha$. Theorem 6.2 shows the characteristic property of an inductive subvalue map: its policy always keeps the player in a region where the player can win the rest of the game *by following the subvalue map policy.*

Theorem 6.2 (Inductive subvalue map ensures winning actions)**.** *Suppose S is an inductive Angelic subvalue map for game $a{:}\alpha$ and compatible with winning condition ϕ, i.e., $a{:}\alpha \models S$ and $S(\mathsf{end}) \models \phi$. Upon starting in any state $\sigma \models S(a)$, and reaching subgame b by following any strategy induced by the subvalue map S, there is a way for Angel to win by continuing to follow S. That is, $S(a) \models \langle (a{:}\alpha \underset{\downarrow}{\forall} S)_{:b}\rangle\, \langle (a{:}\alpha \underset{\downarrow}{\exists} S)_{b:}\rangle\, \phi$.*

It is possible to characterize an inductive subvalue map using recursive conditions expressed via dGL formulas (Def. 6.8).

Definition 6.8 (Inductive subvalue maps)**.** Let S be a map from the subgames of $a{:}\alpha$ to winning subregions. S is an *inductive Angelic subvalue map* for game $a{:}\alpha$, written $a{:}\alpha \models S$, when the following holds. If $a{:}\alpha$ has structure:

$$
\begin{aligned}
a{:}(g{:}\gamma \cup d{:}\delta) &\text{ then } \models S(a) \to S(g) \lor S(d) \text{ and } g{:}\gamma \models S \text{ and } d{:}\delta \models S.\\
a{:}(g{:}\gamma \cap d{:}\delta) &\text{ then } \models S(a) \to S(g) \land S(d) \text{ and } g{:}\gamma \models S \text{ and } d{:}\delta \models S.\\
a{:}(g{:}\gamma\, ;\, d{:}\delta) &\text{ then } \models S(a) \to S(g) \text{ and } g{:}\gamma \models S(\mathsf{end} \mapsto S(d)) \text{ and } d{:}\delta \models S.\\
a{:}(g{:}\gamma)^{*} &\text{ then } \models S(a) \to \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\, S(\mathsf{end}) \text{ and } g{:}\gamma \models S(\mathsf{end} \mapsto S(a)).\\
a{:}(g{:}\gamma)^{\times} &\text{ then } \models S(a) \to S(g) \land S(\mathsf{end}) \text{ and } g{:}\gamma \models S(\mathsf{end} \mapsto S(a)).\\
\text{atomic, i.e., } \alpha \in &\{x := e, x := *, x := \otimes, ?Q, !Q, \{x' = f(x)\ \&\ Q\},\\
&\quad \{x' = f(x)\ \&\ Q\}^{d}\} \text{ then } \models S(a) \to \langle \alpha\rangle\, S(\mathsf{end}).
\end{aligned}
$$

Symmetric conditions characterize when S is an *inductive Demonic subvalue map* for game $a{:}\alpha$ (Appendix B.6.7).

Example 6.3. Consider again the Angelic choice in Fig. 6.1. To establish $S \models 8{:}(6{:}(v := a)^{*} \cup 5{:}(v := v - 1\, ;\, a := *))$, Def. 6.8 first checks that if $S(8)$ (i.e., ϕ_8) holds then either Angel can win by choosing to run subgame 6 (when ϕ_6 holds) or subgame 5 (when ϕ_5 holds). It then also recursively ensures that after going to subgame 5 or 6, S continues to provide a valid Angelic strategy ($6{:}(v := v + a)^{*} \models S$ and $5{:}(v := v - 1\, ;\, a := *) \models S$).

We discuss the subtle loop cases of Def. 6.8. For Demonic loop validity $a{:}(g{:}\gamma)^{\times} \models S$, the following conditions should hold.

1. $\models S(a) \to S(\mathsf{end})$, and $\models S(a) \to \langle\gamma\rangle S(a)$, i.e. $S(a)$ is an invariant that holds inductively and implies the postcondition. Inductiveness condition $\models S(a) \to \langle\gamma\rangle S(a)$ does not appear in Def. 6.8 because it is already implied by later conditions $\models S(a) \to S(g)$ and $g{:}\gamma \models S(\mathsf{end} \mapsto S(a))$.
2. $g{:}\gamma \models S(\mathsf{end} \mapsto S(a))$, i.e. the subvalue map for the loop body ensures finishing within the safe invariant. Using instead the more liberal condition $g{:}\gamma \models S$ to finish the loop body within the postcondition is unsound: Demon can choose to repeat the loop indefinitely, so Angel must play to remain within the inductive region. Alternative condition $g{:}\gamma \models S(\mathsf{end} \mapsto S(g))$ is also unsound since then $S(g)$ is merely inductive and does not have to imply reaching desired postcondition $S(\mathsf{end})$.
3. $\models S(a) \to S(g)$. In case Demon chooses to run the loop, Angel is guaranteed to not get stuck while playing the loop body in winning subregion $S(g)$, so overall loop winning

subregion $S(a)$ should not exceed $S(g)$. On the other hand, $S(g)$ can be weaker than $S(a)$ as it can be one loop iteration away from implying $S(\mathsf{end})$, while $S(a)$ must already imply $S(\mathsf{end})$.

For Angelic loop validity $a{:}(g{:}\gamma)^* \models S$, the following conditions must hold.

1. $\models S(a) \rightarrow \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\, S(\mathsf{end})$, i.e., Angelic loop invariant $S(a)$ ensures that there is a *well-founded* strategy to exit while staying within the control envelope. Angel should not get stuck in a state where her only option is playing the loop forever. Using instead the more liberal condition $\models S(a) \rightarrow \langle\alpha\rangle\, S(\mathsf{end})$ is not enough, since Angel can get stuck in a state where there is a way to exit the loop but only when going outside the subvalue map (see Appendix B.2.1).
2. $g{:}\gamma \models S(\mathsf{end} \mapsto S(a))$, i.e., the subvalue map for the loop body ensures that after playing the loop body, there is still a well-founded exit strategy within the subvalue map ($S(a)$). Using condition $g{:}\gamma \models S(\mathsf{end} \mapsto S(g) \vee S(\mathsf{end}))$ instead is unsound as then $S(g)$ is merely inductive and does not have to imply an exit strategy to reach postcondition $S(\mathsf{end})$ (see Appendix B.2.2).

Theorem 6.3 below proves that *all* inductive subvalue maps are subvalue maps, indicating that the validity conditions compose to produce correct Angel and Demon winning subregions. Thus, while following an inductive subvalue map, the player will never reach a state from which it cannot win (Theorem 6.1). Additionally, Theorem 6.2 at the start of this section shows that following the policy induced by an inductive subvalue map ensures that the agent will never reach a state where it is stuck. In combination, these mean that starting in a suitable initial state, while following an inductive subvalue map, a player cannot lose.

Theorem 6.3 (Inductive subvalue maps are subvalue maps)**.** *For* dGL *game $a{:}\alpha$, if S is an inductive Angelic subvalue map ($a{:}\alpha \models S$), then it is also an Angelic subvalue map for $a{:}\alpha$, i.e., for every subgame $b{:}\beta$ in* $\mathsf{subgames}(a{:}\alpha)$*, $\models S(b) \rightarrow \langle a{:}\alpha_{b:}\rangle S(\mathsf{end})$. Dually, if S is an inductive Demonic subvalue map for $a{:}\alpha$, then S is a Demonic subvalue map for $a{:}\alpha$.*

Def. 6.9 shows how an inductive subvalue map can capture strategies to win a dGL game from every state in the theoretical maximal winning region, indicating that Def. 6.2 is sufficiently liberal. It sets subvalues via modal dGL formulas. The winning subregion for the overall game is maximal.

Definition 6.9 (Model predictive subvalue map)**.** For the game $a{:}\alpha$ and win condition ϕ, the *model predictive Angelic subvalue map* $\{\mathsf{end} \mapsto \phi\} \cup \bigcup_{b\in\mathsf{subgames}(a:\alpha)}\{b \mapsto \langle a{:}\alpha_{b:}\rangle\, \phi\}$ maps every subgame b of $a{:}\alpha$ to the optimal winning regions of its game suffix. Dually, the *model predictive Demonic subvalue map* is $\{\mathsf{end} \mapsto \phi\} \cup \bigcup_{b\in\mathsf{subgames}(a:\alpha)}\{b \mapsto [a{:}\alpha_{b:}]\phi\}$.

Example 6.4. For the game of Fig. 6.1, the model predictive Angelic subvalue map maps subgame 5 to $\langle\{x' = v\}^d\,;\,\alpha\rangle\, x > 0$, which per the semantics of dGL, precisely means the condition under which Angel can win by reaching her assigned winning conditions after playing the remainder of the game $\{x' = v\}^d\,;\,\alpha$.

Remark 4. Model Predictive Control (MPC) [34, 54, 135] is a well-known controller design principle that uses a mathematical model to predict controller behavior and back-computes the control conditions that respect the desired system constraints. Def. 6.9 extends this construction to hybrid games logic. Game suffix $\langle a{:}\alpha_{b:}\rangle\phi$ serves as an exact model of future game behavior after subgame b[6].

Next, Section 6.7 defines a natural, symbolic execution based framework to synthesize subvalue maps that generate policy decisions in polynomial time. Then, Def. 6.10 provides a partial ordering on inductive subvalue maps where more permissive maps are better, and Theorem 6.5 shows that Def. 6.9 is a maximally permissive inductive subvalue map.

6.7 Synthesis Framework for Inductive Subvalue Map with Efficient Checks

To check whether a given control decision lies within the policy of a subvalue map, Def. 6.4 checks whether the resulting state is within a subvalue (e.g., for $b{:}(d{:}\delta \cup g{:}\gamma)$, the policy checks $\sigma \models S(d)$ and $\sigma \models S(g)$). This section synthesizes inductive subvalue maps where state checks are efficient, for quick responses in real systems. In the model predictive solution from the previous section (Def. 6.9), such checking is not efficient because of subvalues that are dGL formulas, for which checking is undecidable [121]. In this section we restrict subvalues to formulas consisting of equations/inequalities of real polynomials along with propositional logic connectives $(\wedge, \vee, \neg, etc.)$ but no quantifiers or modalities, henceforth, *propositional real arithmetic* formulas ($\mathcal{P}_\mathbb{R}$). For these, checking the truth value at a state takes polynomial time. *Given a* dGL *game and winning condition in* $\mathcal{P}_\mathbb{R}$*, this section synthesizes inductive subvalue maps where subvalues are in* $\mathcal{P}_\mathbb{R}$.

Synthesizing such a map is challenging because the subvalues must remain *mutually compatible*, indicating only the winning regions achievable while following the other subvalues. That is, they must follow the inductive subvalue map conditions Def. 6.8, where there are circular dependencies between subvalues. Consider, for example, the case of Demonic loop $a{:}(g{:}\gamma)^\times$. Subvalue $S(a)$ must imply $S(g)$, but $S(g)$ depends on $S(a)$ via condition $g{:}\gamma \models S(\mathsf{end} \mapsto S(a))$. In particular, starting from the optimal subvalue map construction (Def. 6.9) and changing individual dGL formulas to under-approximations in $\mathcal{P}_\mathbb{R}$ would not alone work because cyclically interdependent soundness conditions would need to be reestablished. 1. We address the challenge of interdependent conditions by isolating their effect with a hybrid games analog of loop invariants. 2. To soundly lower formulas to $\mathcal{P}_\mathbb{R}$ while maintaining permissiveness, we use a *predicate transformer* based approach that given a subgame and winning condition in $\mathcal{P}_\mathbb{R}$ computes subvalues in $\mathcal{P}_\mathbb{R}$. This lets us present a backwards symbolic execution [8, 40] based *algorithmic framework* to compute inductive subvalue maps with subvalues in $\mathcal{P}_\mathbb{R}$.

Remark 5. Weakest precondition calculi [40] generate weak *preconditions*. To reason about loops, they generally *check* loop invariants, and rely on external loop invariant generation. Analogously, our framework generates permissive *subvalues*. To reason about loops, it *provides checks for invariants of Angel and Demon loop subvalue maps*. It is *parametric* in the generation of such invariants.

We define predicate transformers over subgames such that *given a postcondition in* $\mathcal{P}_\mathbb{R}$, they *output a subvalue in* $\mathcal{P}_\mathbb{R}$ to recursively produce an entire subvalue map with subvalues in $\mathcal{P}_\mathbb{R}$.

---

[6] Though unlike standard MPC, as we are computing maximal control sets, there is no optimization step to identify the best control solution within the set.

The transformers for the loop-free fragment of dGL are derived from a closely related definition in Chapter 5 [70], which we discuss later in this section. For loops, our framework uses the approach of heuristically generating *invariant candidates* and checking their correctness, which has worked well for hybrid systems [125, 142]. We characterize the necessary and sufficient conditions for a sound invariant (for Angel loops, $\models \mathsf{Inv} \rightarrow \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\phi$, for Demon loops, $\models \mathsf{Inv} \rightarrow S(g) \wedge \phi$). The framework generates invariant candidate Inv (details in (Section 6.7.1)), computes the subvalue map of the loop body, and then checks whether the invariant and loop body map are compatible[7]. Proposed invariants must be in $\mathcal{P}_{\mathbb{R}}$.

Algorithm 2 puts together the predicate transformers to define a recursive solving function $\langle\rangle\mathsf{map}(a{:}\alpha, \phi)$ that computes an inductive Angelic subvalue map with winning regions in $\mathcal{P}_{\mathbb{R}}$ for game $a{:}\alpha$ that is compatible with winning condition $\phi \in \mathcal{P}_{\mathbb{R}}$. Operator $\uplus$ computes the disjoint union of subvalue maps. $S \setminus \mathsf{end}$ indicates the subvalue map S with the mapping for end removed. Such deletion is sometimes necessary to ensure that two maps to which we apply the disjoint union operator are genuinely disjoint. The algorithm is parametrized on *invariant candidate* generation function genInv. $[]\mathsf{map}$ similarly computes an inductive Demonic subvalue map (Appendix B.6.9).

Algorithm 2 Inductive subvalue map synthesis framework

```
function $\langle\rangle\mathsf{map}(a{:}\alpha, \phi)$                    ▷ Parametric Input: genInv
    $S_e \leftarrow \{\mathsf{end} \mapsto \phi\}$;
    if $\alpha \in \{x := e, x := *, x := \otimes, ?Q, !Q, \{x' = f(x) \& Q\}, \{x' = f(x) \& Q\}^d\}$ then
        return $\{a \mapsto \mathsf{simpl}(\langle\alpha\rangle\phi)\} \uplus S_e$
    else if $\alpha = a{:}(g{:}\gamma \cup d{:}\delta)$ then
        $S_1 \leftarrow \langle\rangle\mathsf{map}(g{:}\gamma, \phi)$; $S_2 \leftarrow \langle\rangle\mathsf{map}(d{:}\delta, \phi)$;
        return $S_1 \uplus (S_2 \setminus \mathsf{end}) \uplus \{a \mapsto S_1(g) \vee S_2(d)\}$
    else if $\alpha = a{:}(g{:}\gamma \cap d{:}\delta)$ then
        $S_1 \leftarrow \langle\rangle\mathsf{map}(g{:}\gamma, \phi)$; $S_2 \leftarrow \langle\rangle\mathsf{map}(d{:}\delta, \phi)$;
        return $S_1 \uplus (S_2 \setminus \mathsf{end}) \uplus \{a \mapsto S_1(g) \wedge S_2(d)\}$
    else if $\alpha = a{:}(g{:}\gamma; d{:}\delta)$ then
        $S_1 \leftarrow \langle\rangle\mathsf{map}(d{:}\delta, \phi)$; $S_2 \leftarrow \langle\rangle\mathsf{map}(g{:}\gamma, S_1(d))$;
        return $S_1 \uplus (S_2 \setminus \mathsf{end}) \uplus \{a \mapsto S_2(g)\}$
    else if $\alpha = a{:}(g{:}\gamma)^*$ then
        while true do
            $\mathsf{Inv} \leftarrow \mathsf{genInv}(\langle (g{:}\gamma)^* \rangle\phi)$; $S \leftarrow \langle\rangle\mathsf{map}(g{:}\gamma, \mathsf{Inv} \vee \phi)$
            if $\models \mathsf{Inv} \rightarrow \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\phi$ then return $(S \setminus \mathsf{end}) \uplus \{a \mapsto \mathsf{Inv}\} \uplus S_e$
    else $\alpha = a{:}(g{:}\gamma)^\times$
        while true do
            $\mathsf{Inv} \leftarrow \mathsf{genInv}(\langle (g{:}\gamma)^\times \rangle\phi)$; $S \leftarrow \langle\rangle\mathsf{map}(g{:}\gamma, \mathsf{Inv})$
            if $\models \mathsf{Inv} \rightarrow S(g) \wedge \phi$ then return $(S \setminus \mathsf{end}) \uplus \{a \mapsto \mathsf{Inv}\} \uplus S_e$
```

$\langle\rangle\mathsf{map}$ recursively computes the subvalue for each subgame. The terminal winning subregion

---

[7] Checking Angel loop invariants ($\models \mathsf{Inv} \rightarrow \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\phi$) can be difficult to automate using traditional variant generation techniques. We use refinement based proofs in practice (Appendix B.4.5).

$S(\mathsf{end})$ is always set to the target winning condition ϕ. In the atomic cases, there is only one subgame, and thus only one additional subvalue to compute. Here, the predicate transformer sets the subvalue per dGL winning region semantics (e.g., for $a{:}x := e$, the winning subregion is $\langle x := e\rangle\phi$). But $\langle\alpha\rangle\phi$ is a modal dGL formula which needs to be further simplified to a $\mathcal{P}_{\mathbb{R}}$ formula (in this case, to $\phi(x \mapsto e)$, i.e., the formula $\phi \in \mathcal{P}_{\mathbb{R}}$ with every occurrence of variable x substituted by e). This simplification is written as $\mathsf{simpl}(\langle\alpha\rangle\phi)$. Chapter 5 and Appendix A.1 [70] show how to implement simpl[8].

While $\mathsf{simpl}(\langle\alpha\rangle\phi)$ usually computes an exact expression in $\mathcal{P}_{\mathbb{R}}$, this is sometimes not possible for subgames with complicated ODEs. In such situations, a conservative, sound approximation is made such that $\models \mathsf{simpl}(\langle\alpha\rangle\phi) \rightarrow \langle\alpha\rangle\phi$, i.e. the simplified expression is true only in the states where the original expression was true. Theorem 6.4 shows that the solving framework is robust to such conservative approximations, producing inductive subvalue maps despite them. In the compositional cases, e.g., $a{:}(g{:}\gamma \cup d{:}\delta)$, $\langle\rangle\mathsf{map}$ first computes the subvalues of the composing subgames (in this case S_1 for $g{:}\gamma$ and S_2 for $d{:}\delta$), and then the subvalue for the outer subgame (in the case, $S(g) \vee S(d)$). For the loop cases, $\langle\rangle\mathsf{map}$ generates invariants until it finds one that passes the check.

Fig. 6.1 shows an example of computing a subvalue map using $\langle\rangle\mathsf{map}$. As before, label i denotes the subgame with winning subregion ϕ_i. At the top level, subgame 1 is a Demon loop. For this structure, after setting ϕ_{end}, in second step **Guess** ϕ_1, $\langle\rangle\mathsf{map}$ guesses invariant Inv using a *refinement* heuristic (discussed in Section 6.7.1), computing answer $x > 0 \wedge (v \geq 0 \vee a > 0)$. This invariant is used to recursively compute an inductive Angelic subvalue map for the loop body consisting of subgame 9. A final **Check** step ensures that $\models \psi \rightarrow \phi_9 \wedge x > 0$, i.e., the invariant is compatible with the inner subvalue map so Angel won't get stuck regardless of whether Demon runs the loop or exits.

Theorem 6.4 (Sound solving)**.** *$S := \langle\rangle\mathsf{map}(a{:}\alpha, \phi)$ is an inductive Angelic subvalue map for game $a{:}\alpha$ ($a{:}\alpha \models S$) with all subvalues in $\mathcal{P}_{\mathbb{R}}$, compatible with Angel winning condition $\phi \in \mathcal{P}_{\mathbb{R}}$. Dually, $S := [\,]\mathsf{map}(a{:}\alpha, \phi)$ is an inductive Demonic subvalue map with all subvalues in $\mathcal{P}_{\mathbb{R}}$, compatible with Demon winning condition ϕ.*

6.7.1 Invariant Generation

The subvalue map synthesis framework of Algorithm 2 provides the conditions to check if a candidate formula is a sound invariant. Any combination of invariant generation heuristics [70, 125, 142] can be used to generate candidate invariants. To solve for games where no other techniques apply, we propose a *game rewriting* framework. This framework uses a library of case-based syntactic transformations to rewrite loop game α^* or $\alpha^\times$ as a simpler game β that emulates aspects of its behavior. Then, if β is loop-free, $\mathsf{simpl}(\langle\beta\rangle\phi)$ becomes an invariant guess, otherwise other invariant generation heuristics are tried on β. Checks in Algorithm 2, Line 17 and Line 21 recover soundness.

For example, the step **Guess** ϕ_1 in Fig. 6.1 shows a rewrite when computing an invariant of the outer loop (ϕ_1). The rewritten game (on the second line) emulates inner loop $(v := v + a)^*$

---

[8] The function $\mathsf{reduce}(\langle\alpha\rangle\,\phi, A)$ (Appendix A.1) produces a $\mathcal{P}_{\mathbb{R}}$ formula that is equivalent to $\langle\alpha\rangle\,\phi$ under assumption formula A. $\mathsf{simpl}(\langle\alpha\rangle\,\phi)$ is $\mathsf{reduce}(\langle\alpha\rangle\,\phi, \top)$.

by letting Angel decide in advance how many loop iterations to run and assign her decision freely to new variable n. Then, setting $v := v + an$ conservatively estimates the effect of running the loop $\lceil n \rceil$ times (when n is fractional, v ends up smaller, and thus conservatively less favorable to Angel than after $\lceil n \rceil$ loop iterations). The rewrite also eliminates the outer loop using a generalized form of one-shot refinement (Chapter 5 [70]) where running the ODE emulates the worst case behavior of multiple loop iterations. In general, different rewrite rules are tried heuristically based on the shape of a problem until an answer is found. New rewrite rules can be added to the library as needed. Appendix B.4.4 shows the rewrite rules that solve the problems of Section 6.10.

6.8 Maximal Solution

For a given game and winning condition, there are many possible correct inductive subvalue maps. Most useful are the *more permissive* subvalue maps that allow as many control actions as possible. More permissive subvalue maps leave more control options open, permitting more space to optimize for secondary goals when identifying a concrete, specialized controller within the envelope. We formalize subvalue map permissiveness with a partial ordering where more permissive subvalue maps are greater than less permissive subvalue maps.

A natural candidate for ordering the subvalue maps of game $a{:}\alpha$ would be to say that S is at least as permissive as S' ($S \sqsupseteq S'$) when for each subgame b in $\mathsf{subgames}(a{:}\alpha)$, *every state in subregion $S'(b)$ is also within subregion $S(b)$*, i.e., $\models S'(b) \to S(b)$, so that $S(b)$ locally *permits* the agent to play into subgame b in all the states that $S'(b)$ does. However, this ordering fails to order cases like the following. Let S' set the winning region for the overall game to be empty ($S'(a) = \bot$), meaning that Angel is never allowed to play at all. For *any* other inductive subvalue map S, a good ordering should indicate that $S \sqsupseteq S'$ since S cannot possibly be less permissive, but the proposed naïve ordering does not. The winning subregion for some later subgame $b \neq a$ can be such that the check $\models S'(b) \to S(b)$ fails (see Appendix B.1 for details). An improved ordering makes the following modified check for every subgame b. $S \sqsupseteq S'$ when, *amongst the states that are reachable at subgame b while playing per subvalue map S'*, every state that is in subregion $S'(b)$ should also be within subregion $S(b)$. To express this logically, we first characterize the states that are reachable at subgame b while playing per subvalue map S'.

The game prefix (Def. B.10) of subgame b in the game $a{:}\alpha$, written $a{:}\alpha_{:b}$, features *all* possible behaviors that can happen *before* an occurrence of subgame b within $a{:}\alpha$. To characterize all the states reachable at subgame b while following the policy induced by S', we use the universal projection of S' onto the game prefix $(a{:}\alpha_{:b}) \forall\!\!\downarrow S'$. Def. 6.10 compares the winning subregions of S and S' after running the projected game to filter out unreachable states.

Definition 6.10 (Permissiveness ordering)**.** For two inductive Angelic subvalue maps S and S' for the game $a{:}\alpha$, S is at least as good as S', written $S \sqsupseteq S'$, iff for each subgame b of $a{:}\alpha$, $\models \langle a{:}\alpha_{:b} \forall\!\!\downarrow S' \rangle (S'(b) \to S(b))$ and $\models \langle a{:}\alpha \forall\!\!\downarrow S' \rangle (S'(\mathsf{end}) \to S(\mathsf{end}))$. Dually, for inductive Demonic subvalue maps, $S \sqsupseteq S'$ iff for subgame b of α, $\models [(a{:}\alpha_{:b})[]\forall\!\!\downarrow S'](S'(b) \to S(b))$ and $\models [a{:}\alpha[]\forall\!\!\downarrow S'](S'(\mathsf{end}) \to S(\mathsf{end}))$.

Remark 6. For every game and winning condition combination, the model predictive subvalue map (Def. 6.9) is a maximally permissive solution (Theorem 6.5). As Def. 6.9 always constructs an optimal subvalue map, it optimally solves the dGL control envelope synthesis problem, achieving completeness for symbolic hybrid games synthesis (when modal, dGL subvalues are permitted).

Theorem 6.5 (Maximal inductive subvalue map)**.** *Amongst the inductive Angelic subvalue maps for game* $a{:}\alpha$ *compatible with winning condition* ϕ*, the model predictive Angelic map given by Def. 6.9 is maximal under the ordering of Def. 6.10. That is, for all* S *such that* $a{:}\alpha \models S$*, model predictive Angelic subvalue map* S' *satisfies* $S' \sqsupseteq S$*. Dually, the model predictive Demonic subvalue map (Def. 6.9) is maximal amongst inductive Demonic subvalue maps per the ordering of Def. 6.10.*

6.8.1 Recovering Solution Optimality

When constructing subvalue maps where subvalues must be in $\mathcal{P}_{\mathbb{R}}$, Algorithm 2 is not guaranteed to construct an optimal subvalue map. This section describes a retrospective check that can be used to test whether the synthesis procedure failed to produce a maximally permissive envelope, indicating that more aggressive synthesis/invariant generation is required for more permissiveness. First, use Algorithm 2 to independently construct strategies for Angel and Demon for the same game $a{:}\alpha$. Let $S_{\langle\rangle}$ be the computed Angelic subvalue map for winning condition ϕ and $S_{[]}$ be a Demonic subvalue map for the opposite objective $\neg\phi$. Cross-checking these subvalue maps can recover whether they are optimal, capturing all possible ways for their players to win. If for each subgame b, $S_{[]}(b) \vee S_{\langle\rangle}(b)$ is valid, then intuitively, $S_{\langle\rangle}(b)$ and $S_{[]}(b)$ together cover every possible state, identifying it either as one where Angel has a winning strategy or where Demon does. Further weakening $S_{\langle\rangle}(b)$ would violate consistency (i.e., it is never possible for both Angel and Demon to win)[9]. So $S_{\langle\rangle}(b)$ must be a maximally permissive, optimal Angel winning region, and $S_{\langle\rangle}$, an optimal Angelic subvalue map.

6.9 Relationship to Runtime Monitoring

A runtime monitor (constructed similarly to existing work on monitoring dL [99] and constructive dGL [20]) raises an alarm or intervenes to enforce a correct fallback when gameplay deviates from the control envelope. It is possible to model the use of a subvalue map as a *runtime monitor* in a single dGL formula. For an Angelic subvalue map, this *monitored game* is constructed by 1. giving Demon control over Angelic decisions while restricting him to play per the subvalue map (like in universal projection) to mimic an unverified controller that plays unreliably while being forced to stay within the control envelope by a runtime monitor and 2. ensuring before each such monitored subgame that the subvalue map allows *some* control action using Angelic tests. Eq. (6.3) shows the monitored game for the running example. The *monitored game* for dGL game $a{:}\alpha$ and Angelic subvalue map S, written $a{:}\alpha \lightning S$, is generated recursively per the structure of $a{:}\alpha$ as follows. If $a{:}\alpha$ has structure:

$$a{:}x := *,\ \text{then}\ ?(\exists x\, S(\mathsf{end}))\,;\ x := \otimes\,;\ !S(\mathsf{end}).$$

---

[9] This is analogous to how in α-β pruning [81], if MIN's action results in a position where MAX wins, then that action cannot be part of MIN's strategy.

$$a{:}(g{:}\gamma;\, d{:}\delta),\ \text{then } g{:}\gamma \lightning S(\mathsf{end} \mapsto S(d))\,;\, d{:}\delta \lightning S.$$

$$a{:}(g{:}\gamma \cup d{:}\delta),\ \text{then } ?(S(g) \vee S(d))\,;\, ((!S(g)\,;\, g{:}\gamma \lightning S) \cap (!S(d)\,;\, d{:}\delta \lightning S))\,.$$

$$a{:}(g{:}\gamma)^*,\ \text{then } ?S(a)\,;\, (!S(g)\,;\, g{:}\gamma \lightning S(\mathsf{end} \mapsto S(a))\,;\, ?S(a))^\times\ ;\, !S(\mathsf{end}).$$

$$a{:}(g{:}\gamma \cap d{:}\delta),\ \text{then } (g{:}\gamma \lightning S) \cap (d{:}\delta \lightning S). \quad a{:}(g{:}\gamma)^\times,\ \text{then } (g{:}\gamma \lightning S(\mathsf{end} \mapsto S(a)))^\times.$$

$$a{:}\{x' = f(x)\ \&\ Q\},\ \text{then } ?\langle \alpha \rangle\, S(\mathsf{end})\,;\, \{x' = f(x)\ \&\ Q\}^d\,;\, !S(\mathsf{end}).$$

atomic and not controlled by Demon, i.e.,

$$\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x)\ \&\ Q\}^d\},\ \text{then } \alpha.$$

Monitoring a Demonic subvalue map is symmetric (see Appendix B.6.5). Every inductive subvalue map S is a sound runtime monitor, i.e., $\models S(a) \rightarrow \langle a{:}\alpha \lightning S \rangle\, S(\mathsf{end})$ (Theorem D.1).

6.10 Evaluation

To test the generality and flexibility of our approach, we demonstrate it on examples from classic control theory problem classes (event-triggered control, reach-avoid problems, nonlinear control, infinite horizon switching). We compare to CESAR (Chapter 5 [70]), which synthesizes symbolic control envelopes as well, but for hybrid *systems* with a *fixed, time-triggered template*. NYCS [13] is a tool that can synthesize control envelopes for hybrid games (expressed via hybrid automata), but does not solve parametrically for symbolic constraints[10]. We compare performance.

CESAR and NYCS cannot solve the new examples, each of which displays different problem features, demonstrating the greater generality of our approach, which solves them within a few minutes (Fig. 6.3a). We also evaluate our approach on the CESAR benchmark suite (Appendix A.3 [70]). Our implementation's performance is similar to CESAR despite the significantly greater generality (Fig. 6.3b). To compare to NYCS, we procedurally generate 25 instantiations of a reach-avoid problem. The implementation finds control envelopes for 17 of them, timing out for the rest, while NYCS solves all of them, suggesting potential for future optimizations for this problem class.

The experiments were run on a 32GB RAM M2 MacBook Pro. The implementation uses Pegasus [142], CESAR (Chapter 5 [70]), and additional rewriting heuristics for invariant generation. It uses Mathematica for simplification and quantifier elimination. Our examples favor interesting control challenges over dynamic complexity to avoid the computer algebra bottleneck, which is an avenue for future research. Appendix B.4.3 provides dGL formulas for all the new examples.

Event-triggered ETCS (Model 9) is an example of an *event-triggered* control problem, in contrast with the time-triggered control handled by Chapter 5 [70]. In time-triggered control, the controller repeatedly polls with some maximum time latency. In event-triggered control, the controller is instead triggered at *events*, when specific conditions are met. Identifying what these

---

[10] Our problems have symbolic parameters that can represent any real number, e.g., in event-triggered ETCS (Model 9), A, B and T. We generate an envelope that is parametric in these symbols, equivalent to making uncountably infinitely many calls to NYCS corresponding to every possible parameter value assignment. Such parametricity is crucial in some applications (e.g., when parameters are unknown statically and only identified at runtime [51]).

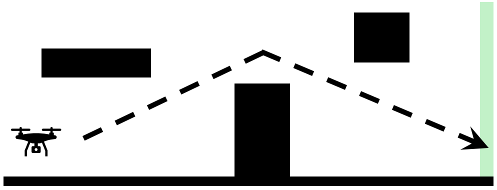
(a) Summary of new examples. Appendix B.4 discusses them.

Benchmark	**Ours**	**CESAR**	**NYCS**	**Problem Type**
Event ETCS	23s	∞	∞	Event-triggered control
Infinite Track	233s	∞	∞	Infinite horizon switching
Surgical Robot [83]	168s	∞	∞	Nonlinear control
Highway Driving [94]	91s	∞	∞	Time-triggered control with adversarial agent
Reach-avoid Robot	21s	∞	∞	Reach-avoid problem

Benchmark	**Ours**	**CESAR**	**NYCS**
ETCS Train	12s	15s	∞
Sled	64s	64s	∞
Intersection	104s	104s	∞
Curvebot	111s	112s	∞
Parachute	113s	123s	∞
Corridor	40s	40s	∞
Power Station	56s	56s	∞
Coolant	311s	312s	∞

(b) Running time comparison on the CESAR benchmark suite (Appendix A.3 [70]).

Tool	**Success**	**Failure**
Ours	17	8
NYCS	25	0
CESAR	0	25



(c) Setup and outcome of the quadcopter reach-avoid problem with procedurally generated obstacle placement.

Figure 6.3: Benchmark summaries and results.

events are that should trigger controller intervention is a part of the synthesis challenge. Our example problem (Model 9) translates the ETCS Train benchmark of the CESAR benchmark suite from the time-triggered to the event-triggered paradigm. It describes a train controller that must stop the train by point e, and has the choice to either accelerate or to brake. Line 3 shows the plant which Angel (emulating the controller) chooses how long to run. The synthesized subvalue map indicates when Angel must stop continuous evolution to enforce braking, characterizing the event at which to trigger the controller. Modeling an event-triggered system as a game is subtle: the controller has the ability to choose how long to run the plant, and should not be permitted to win by interrupting infinitely often and inducing Zeno behavior. dGL's flexibility allows a correct encoding using nested loops and time progress assertions.

Surgical Robot shows an example of nonlinear control (Model 19). It expresses the core control challenge in a case study [83] modeling force feedback in surgical robots. The objective is to dampen the force applied by the surgeon to ensure that a surgical tool stays within bounding planes called *virtual fixtures*. The damping factor g (Line 3) that we synthesize for can be *any* nonnegative real number, implying infinitely many control possibilities.

Highway Driving (Model 22) expresses the core control challenge of a case study [94] where on a highway, a controlled car must follow the car ahead at a safe distance. The car ahead can behave adversarially by braking suddenly. *Reach-avoid Robot* (Model 21) demonstrates an example of envelope synthesis for reach-avoid problems, where an agent must *reach* an objective while *avoiding* unsafe situations. *Infinite Track* (Model 20) is an example of infinite horizon switching. To remain safe for arbitrarily long time, the controlled vehicle must keep switching directions at the right moments, and make *infinitely* many switches. Appendix B.4.1 discusses these examples further, and Appendix B.4.3 lists the models. The synthesized control envelopes for *Infinite Track*, *Surgical Robot*, and *Reach-avoid Robot* are maximally permissive. The envelopes for *Highway Driving* and *Event-triggered ETCS* are *not* maximally permissive because of the rewrite heuristics used.

To compare the algorithm to NYCS, we procedurally generate instantiations of *quadcopter*, replacing symbolic parameters with concrete numbers, to create a reach-avoid challenge. Each instantiation describes a scenario like in Fig. 6.3c, where a quadcopter must reach a target safe area while avoiding obstacles placed randomly in the environment. The quadcopter constantly moves forward and can choose to either move upwards or downwards, but cannot revise its decision, once made, for one second. The objective is to find a control envelope showing from which points the quadcopter has a way to reach the target safe zone while avoiding the obstacles. The implementation finds control envelopes for 17 problems. The other 8 problems time out after 20 minutes. CESAR cannot solve any of these problems because they do not fit its template. NYCS, with its polytope representation and optimizations for this problem class, solves all of the problems. Appendix B.4.2 provides further details.

6.11 Related Work

Subvalue maps represent nondeterministic policies for hybrid games in a way that is compositional at the level of imperative program constructs such as loops and branches. They can be verified/synthesized leveraging program analysis techniques like loop invariants, making it possible to answer control envelope synthesis questions that no previous work solves.

- *Program logics* like game logic and dGL express games as imperative programs with Angelic and Demonic nondeterminism [111, 114, 121]. Their semantics compositionally define when *there exists* a way to win a game, but not when a given policy wins.
- Refinement [16, 105], Hoare logic [95], *Differential Floyd-Hoare logic* [61], and constructive proof [20] based approaches soundly construct deterministic policies (i.e., programs with Angelic nondeterminism fully resolved) but do not study *maximal sets of solutions*.

Model 9 Event-triggered ETCS

$$\begin{array}{r|l} \textsf{assume} & A > 0 \land B > 0 \land T > 0 \rightarrow \Big\langle \\ \textsf{choice} & \Big(t := 0\,;\, \big((a := A \cup a := -B)\,; \\ \textsf{plant} & \{p' = v, v' = a, t' = 1 \,\&\, e - p > 0\}\big)^*\,;\, ?t \geq 1\Big)^{\times}\Big\rangle \mathsf{True} \end{array}$$

They do not have a direct representation of nondeterministic policies from which to derive *sets* of permissible control actions, or a formal comparison of permissiveness of such policies, both of which are crucial parts of the permissive policy synthesis problem that we solve.

- A line of research solves games specified by *hybrid automata* with objectives expressed in logics like LTL [13, 41, 63, 154, 166]. Our approach targets specifications at a higher level of abstraction, where the game has imperative programming constructs like loops and branches. Compositionality is also at the level of these constructs, and the theoretical development required is different (e.g., the subtle Angel loop case in Def. 6.8).
- Traditional *game theory* [139, 156], and *reinforcement learning* [145] have recursive solving techniques and representations of nondeterministic policies (value functions in RL and value tables in tabular games [10, 146, 147]). But these works operate at a different level of abstraction than our imperative program-like hybrid games. Also, our work is symbolic rather than numeric, and creates envelopes parametric in the symbolic parameters of the input game. Finally, our subvalue maps are logically formalized, suitable for computerized verification and formally justified synthesis.
- *Compositional game theory* [7, 57] also has a compositional representation of nondeterministic policies but also operates at a different level of abstraction, solving not hybrid games but abstract games with sequential and parallel composition.
- Compared to safe set and *barrier function* approaches [4, 6, 33, 35, 97, 130, 144], we again solve for the higher level of abstraction of dGL, with program structures such as loops, as well as the adversarial dynamics of games.
- Our control envelopes are also conceptually different from *permissive strategies* [15]. A permissive strategy typically needs to ensure the player wins, but a control envelope just needs to ensure that the player remains in states from where a winning strategy exists.

6.12 Conclusion

We introduce a generalized symbolic control envelope synthesis approach for hybrid games. Such envelopes identify what control action is safe to take when. We represent the envelopes using inductive subvalue maps and provide an algorithmic framework to generate them. Our approach allows the expression and synthesis of a variety of control problems. The approach should generalize for non-dGL Markovian games, where the policy depends only on state and not the history of previous moves, which is an avenue for future research.

An artifact is available [77]. This artifact contains:

1. A version of KeYmaera X extended with implementations of our synthesis framework, and of the CESAR algorithm from Chapter 5.
2. KeYmaera X Specification files for the examples introduced by this chapter.
3. NYCS specifications for the quadcopter instances.
4. Instructions to set up and run the code.

A limitation that arises in the chapter is the bottleneck of computer algebra, and the requirement to identify good rewrite heuristics to cover new problem classes. The next chapter addresses these challenges by leveraging large language models as a general heuristic for generating invariants.

Chapter 7

Automatic Synthesis

Chapter 6 introduced a framework for synthesizing symbolic control envelopes for differential game logic games. It implemented the framework by symbolic execution which relied on expensive computer algebra operations such as quantifier elimination. For performance, it used a library of rewrite rules for different problem types. This chapter increases scalability: it reduces dependence on expensive computer algebra operations and eliminates the need for case-by-case heuristics by using large language models (LLMs). It also introduces the ability to focus on a *specific class* of solutions, which often, in real-world applications, results in a sufficiently useful control envelope while making the synthesis process more efficient. In this chapter, the user provides the game and describes their desired solution informally, and our tool computes the corresponding verified control envelope.

7.1 Overview

As before, we represent nondeterministic policies as *subvalue maps*. Once the LLM guesses a policy (subvalue map), we can verify if it is correct (non-losing, never stuck) by checking some dGL formulas identified by Chapter 6. For example, in the subvalue map of Fig. 7.1, to check that the subvalue of the loop, ϕ_{inv}, is correct, we make sure $\phi_{\mathsf{inv}} \rightarrow \phi_{\mathsf{safe}} \wedge \phi_{\mathrm{x}}$ holds, that is, Angel won't lose regardless of whether Demon continues the loop or not. If he ends the loop, Angel will win into ϕ_{safe}, and if he repeats the loop, Angel will still win because ϕ_{x} holds, which inductively tells us that she has a winning strategy from the start of the loop body $\mathtt{x} := *$. While this check was simple (no modalities), for other subgames, checks can involve modal dGL formulas, for which we use the verification tool of Section 4.4.

For synthesis, the same five case studies as in Chapter 4 are used for evaluation (Model 6, Model 25, Model 23, Model 26 and Model 24 listed in Appendix C.2) but with some initial assumptions missing (it will be the task of the synthesis pipeline to come up with all conditions necessary to make a strategy succeed). Additionally, each is associated with an informal natural language description of which of the many possible control solutions to synthesize (listed in Appendix C.2). Given the example of Model 6 (illustrated by Fig. 7.1) and a request to synthesize the solution that sets the system to an equilibrium point as soon as possible, our tool automatically identifies the corresponding subvalue map. It computes that Angel must pick `x_add` and

1. Input:

1. **dGL Game**: *Model 6* (Angel freely chooses $\mathtt{x\_add}$ and $\mathtt{y\_add}$, and wins if she keeps populations x and y above the minimum thresholds).
2. **Desired Solution (Informal)**: *"Set the system to its equilibrium point."*

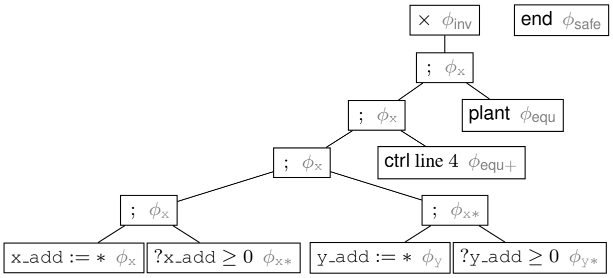
3. Control policy derived from subvalue map

In state σ:

1. At subgame $\mathtt{x\_add} := *, \mathrm{x} := e$ permitted iff $\sigma(\mathrm{x} \mapsto e) \models \phi_{\mathrm{x}}$.
2. At subgame $\mathtt{y\_add} := *, \mathrm{y} := e$ permitted iff $\sigma(\mathrm{y} \mapsto e) \models \phi_{\mathrm{y}}$.

where $\sigma(\varphi \mapsto e)$ is the state obtained by updating σ to map variable φ to value e.

2. Synthesized subvalue map (some subgames unexpanded)



where

$\phi_{\mathsf{const}} \equiv a > 0 \land b > 0 \land d > 0 \land g > 0,$
$\phi_{\mathsf{pre}} \equiv x_{\mathrm{mn}} d \leq g \land y_{\mathrm{mn}} b \leq a,$
$\phi_{\mathsf{safe}} \equiv \mathrm{x} \geq x_{\mathrm{mn}} \land \mathrm{y} \geq y_{\mathrm{mn}},$
$\phi_{\mathsf{equ}} \equiv \phi_{\mathsf{const}} \land \phi_{\mathsf{pre}} \land d\mathrm{x} = g \land b\mathrm{y} = a,$
$\phi_{\mathsf{inv}} \equiv \phi_{\mathsf{const}} \land \phi_{\mathsf{safe}} \land \phi_{\mathsf{pre}} \land d\mathrm{x} \leq g \land b\mathrm{y} \leq a \land \mathrm{x} > 0 \land \mathrm{y} > 0,$
$\phi_{\mathsf{equ+}} \equiv \phi_{\mathsf{const}} \land \phi_{\mathsf{pre}} \land d \cdot (\mathrm{x} + \mathtt{x\_add}) = g \land b \cdot (\mathrm{y} + \mathtt{y\_add}) = a,$
$\phi_{\mathrm{y}}* \equiv \phi_{\mathsf{equ+}} \land \mathtt{y\_add} \geq 0,$
$\phi_{\mathrm{y}} \equiv \phi_{\mathsf{const}} \land \phi_{\mathsf{pre}} \land d \cdot (\mathrm{x} + \mathtt{x\_add}) = g \land b\mathrm{y} \leq a,$
$\phi_{\mathrm{x}}* \equiv \phi_{\mathrm{y}} \land \mathtt{x\_add} \geq 0,$
$\phi_{\mathrm{x}} \equiv \phi_{\mathsf{const}} \land \phi_{\mathsf{pre}} \land d\mathrm{x} \leq g \land b\mathrm{y} \leq a.$

ctrl line 4 and plant refer to the corresponding parts of Model 6, and end is the special node representing the end of the game. ϕ_{safe} is the same correct control condition from Model 6, that is, $\mathrm{x} \geq x_{\mathrm{mn}} \land \mathrm{y} \geq y_{\mathrm{mn}}$. The other formulas are synthesized by the pipeline.

Figure 7.1: Example of envelope synthesis shown on Model 6

$\mathtt{y\_add}$ such that $\mathrm{x} + \mathtt{x\_add} = g/d$ and $\mathrm{y} + \mathtt{y\_add} = a/b$, and that this policy only works when, initially, $\mathrm{x} \leq g/d$, $\mathrm{y} \leq a/b$, $g/d \geq x_{\mathrm{mn}}$, and $a/b \geq y_{\mathrm{mn}}$.

7.2 Pipeline Description

Weakest precondition calculus for normal programs [40] starts at the end of the game and works backwards, requiring loop invariant guesses and retrospective checks at loops. Similarly, our pipeline executes the precondition calculus for subvalue maps (Chapter 6) on the game backwards from end to start, computing subvalues as preconditions. Some subgames, such as simple assignments, are easy to symbolically execute. Computing the precondition of others: loops, continuous dynamics, and nondeterministic choices, is expensive or undecidable in general. For these, the LLM is prompted to guess subvalue map entries which encode Angel's winning strategy. A retrospective check using Z3 [39] when possible and KeYmaera X when necessary recovers soundness. Fig. 7.2 shows the synthesis pipeline.

Initially, the LLM analyzes the game, identifying player actions and control modes, and comes up with a detailed strategy for Angel based on the informal guideline provided by the user. The strategy is expressed as a sequence of steps that Angel will take, stored to provide context in later steps. In the main subvalue map computation phase, the *Next Subvalue* step goes backwards to the next unprocessed subgame, and either symbolically computes the subvalue if it is easy, or prompts the LLM to guess it otherwise. In the latter case, the subvalue is checked using Z3 or KeYmaera X with existing, LLM-free automation. Guesses and their outcomes are

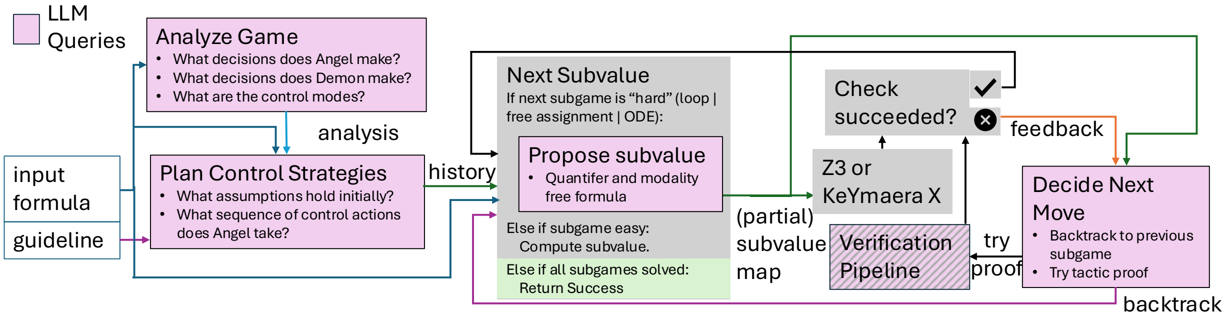


Figure 7.2: Synthesis Pipeline. Appendix C.4 shows prompts.

Configuration	**Solved**	**Pass@1**	**Avg Cost ($)**	**Avg Calls**	**Avg Output (k Tokens)**
Baseline	0/5	—	—	—	—
Ablation	2/5	0.40 ± 0.24	0.80 ± 0.06	19.5 ± 2.5	76 ± 7
Full Pipeline	**4/5**	$\mathbf{0.80 \pm 0.20}$	$\mathbf{2.19 \pm 0.73}$	$\mathbf{41.0 \pm 10.0}$	$\mathbf{203 \pm 67}$

Table 7.1: Synthesis results across configurations. *Solved* indicates the number of case studies where at least one run succeeded. *Pass@1* is the mean, over case studies, of the empirical success rate of a single tool attempt (each attempt having four parallel threads of the pipeline) ± standard error across case studies. Cost, calls, and output tokens are averaged *only over successful runs* ± standard error. Output token counts are reported in thousands. The ablation disables LLM-assisted proving and custom backtracking.

stored to provide context for future guesses[1].

If the check succeeds, the pipeline continues to the next unprocessed subgame. Otherwise, there are two possible recovery strategies, that the LLM chooses between. If the lemma was too complicated to verify using existing automation, it should try using the automatic theorem proving system from Section 4.4. Otherwise, the subvalue may be genuinely incorrect. The incorrectness may lie at the current subgame, or result from a bad earlier subvalue guess that makes the current subgame unsolvable (e.g., a non-inductive loop invariant causing problems in the loop body subvalues). The LLM should choose to backtrack to the subgame where the bad guess was made, and subvalue computation resumes from there.

The interactive theorem proving sub-system used in this pipeline has a modification relative to the one described in Chapter 4. There is an additional step at the end of every verification loop iteration where the LLM can decide whether the goal it was trying to prove was simply invalid, and it should escape the proof attempt and backtrack to guess a different subvalue.

Configuration	**Chem. Reaction**		**Coolant**		**Train**		**Lotka-Volterra**		**Van der Pol**	
	Succ.	Cost	Succ.	Cost	Succ.	Cost	Succ.	Cost	Succ.	Cost
Ablation	1/1	0.86	0/1	—	0/1	—	0/1	—	1/1	0.74
Full Pipeline	**2/2**	**0.75**	0/2	—	**2/2**	**5.35**	**2/2**	**1.90**	**2/2**	**0.75**

Table 7.2: Per-benchmark synthesis results. *Succ.*: successful runs out of total runs. *Cost*: average dollar cost of *successful runs*.

7.3 Outcomes

We apply the synthesis pipeline to all five case studies, using GPT-5 (default reasoning) to make all LLM calls. The implementation and experiments (LLM responses cached and re-playable) are available online at `https://doi.org/10.1184/R1/32248389` [74]. Each experimental run of the tool consists of four parallel runs of the pipeline of Fig. 7.2 that terminate when any one of them succeeds, or when they collectively run out of a nine hundred thousand output token budget. Table 7.1 summarizes the results of our experiments. A case study is solved when one of the runs of the synthesis tool produces a correct subvalue map. For each benchmark, we perform two runs of the tool (each run consisting of four parallel threads running the synthesis pipeline). In addition, we perform an ablation experiment where we do not allow the LLM to perform interactive theorem proving or backtracking to the subgame of its choosing. In this variant, the LLM must rely only on KeYmaera X's automation to check the guessed subvalues and on a depth-first search strategy for backtracking on subvalue guesses.

As a baseline, we compare to the tool from Chapter 6, which is the only one general enough to synthesize control envelopes for the problems of this chapter. The tool's existing template-based case-by-case heuristics and exact symbolic execution are not yet general or scalable enough to solve any of the five case studies (although it is possible that the tool could succeed if extended with the right custom heuristics on a case-by-case basis).

In contrast, the LLM-powered pipeline is very general, free of the need for case-by-case encoding of heuristics, and solves four out of five case studies. Further, the pipeline is able to account for the one-sentence informal guidelines describing the desired control solutions, shown in Appendix C.2.1, that are part of the input problem. This ability is useful in practice because control engineers often know what kind of solution they are looking for at an informal level, but getting the *exact mathematical details* right can be challenging. A control envelope that focuses on all the solutions *in a specific class of interest* (rather than as many solutions as possible, some of which are ultimately not useful in light of the secondary control objectives) can be desirable because it avoids carrying unnecessary arithmetic complexity, making it easier to understand and use downstream. For example, simpler expressions would be more efficient to check when monitoring a control system at runtime for compliance with the envelope. The control envelope is *not formally guaranteed* to match the informal description (it is not clear what such a guarantee would mean given the ambiguity of natural language), but the synthesized map *is* guaranteed to

[1] Control flow is more complicated for the special case of loops, where the loop invariant is guessed first, then the subvalue map of the loop body is computed assuming the invariant holds at the end of the body, and finally the loop invariant is retrospectively checked for inductiveness given the computed subvalue map of the body.

be a correct (non-losing, never stuck) control solution of the game. In practice, in our experiments the synthesized control envelopes do match the informally described solutions.

In the ablation, without the LLM-assisted proving, only two out of the five benchmarks are solved. The lower average cost/number of calls for successful benchmarks in the ablation is explained by only simpler problems being solved and contributing to the average. Table 7.2 shows the per-benchmark outcomes.

The synthesis pipeline failed only on the Coolant problem. The coolant has a subtle loop invariant (subvalue corresponding to the main loop) that is easy to get wrong (and hard to prove right). The LLM did reach nearly-correct loop invariants, dove into the loop body and generated subvalues (differential invariants) for the dynamics, decided to prove that its inner choice of subvalue was correct with an interactive proof since an automatic proof would generally fail, only to ultimately fail the final loop correctness check because the original loop invariant was missing something. It backtracked to make adjustments to the invariant, and got stuck in a series of expensive failed proof attempts. In the ablation experiment, the inability to do LLM-assisted tactical proofs became a bottleneck, since for benchmark problems Train and Lotka-Volterra, the nonlinear dynamics led to proof obligations that would be hard to discharge without tactics.

7.4 Discussion

Notice that in solving the benchmark Model 23, we have come full-circle by effectively synthesizing the conservative train control envelope (Model 1 in Chapter 3) automatically. Synthesizing the more complex solutions later in Chapter 3 is beyond the current abilities of the pipeline, but remains an interesting avenue for future work.

7.4.1 Agentic Approach

The pipeline described in this chapter restricts the LLM to a strictly structured sequence of steps. Within these steps, the LLM has the flexibility to decide on the direction of problem solving, for example, by deciding where to backtrack to and when to give up on an interactive proof attempt. What would it look like to take this flexibility to its extreme and give the LLM maximal freedom in guiding the synthesis process? This would be an *agentic* approach, where an LLM agent, given only an input game and envelope verification tool, would be tasked to come up with a control envelope that the tool verifies as correct. Such an approach would be of practical importance because it would benefit from the improvement in LLM agents that industry is pursuing today, and it would also be easy to use for practitioners who already have access to an LLM agent.

We develop an envelope verification tool that enables this agentic approach, available online [67]. Structured much like the pipeline of Fig. 7.2, it is based on the theory of Chapter 6. The tool takes in an envelope file that contains a game and maps its subgames to formulas. The tool then verifies that they form an inductive subvalue map (Def. 6.8) using KeYmaera X and Z3. Envelope files can also optionally include helper tactics marked as helping specific steps in the verification process go through. If the envelope file fails verification, the tool provides feedback such as where the verification failed and what the prover output was. An LLM agent can then be tasked with coming up with an envelope file that the tool verifies as correct. In preliminary experiments, we found that, given a game similar to Model 23, with the addition of slope and curve as in Chapter 3, the OpenAI codex agent using GPT-5.4 was able to use the tool to come up with the conservative train control envelope in that chapter. Further investigation of the costs and benefits of the agentic approach is another avenue for future work.

7.4.2 Conclusion

This concludes the second part of the dissertation, on automatic synthesis of control envelopes given the shape of the control problem. The remaining bottleneck for control envelope design becomes how to come up with this shape, the hybrid game specifying the control possibilities, in the first place. This is the subject of Chapter 8.

Chapter 8

Autoformalization

Previous chapters have shown how to synthesize control envelopes once the shape of the control problem has been formalized as a hybrid game. A final challenge that remains is coming up with the right hybrid game in the first place.

In fact, in general, writing a formal model of reliable cyber-physical systems (CPSs) is a bottleneck preventing broader industry adoption [137]. Unlike program verification, where a mathematically meaningful object to verify already exists in the form of a program, for CPSs, we must first create a formal model of the physical system and environment. This process is notoriously difficult, time-consuming, and error-prone.

Can large language models (LLMs) help autoformalize physical systems? They have shown autoformalization abilities for mathematics [89, 162] and CPS contracts [30, 110]. This chapter presents an initial, experiment-focused exploration of the ability of LLMs to autoformalize the underlying physics problems, faithfully preserving their exact continuous dynamics and discrete transitions in differential game logic (Chapter 2 [121]). We propose 20 benchmarks derived from hard undergraduate kinematics problems [66, 90, 96]. We find that, supported by few-shot prompting and parser feedback, OpenAI o3 has a top-5 accuracy of 70% (success rate on selecting the best out of 5 attempts per benchmark). The main cause of failure is checker limitations for the most complex problems.

Our benchmarks and pipeline are available as an online artifact at `https://doi.org/10.1184/R1/28934195`. Evaluating autoformalization of a physical system is challenging because every problem has many valid formal representations, and conversely, models can be incorrect for many subtle reasons. To automatically assess the semantic proximity of generated formal models to the original physics problem, we provide a checker that:

1. performs *symbolic execution* over the generated formal model, examining whether the expected symbolic solution to the original physics problem is recovered.
2. ensures the model is not in stasis by requiring that a minimum number of variables are mutated, indicating they are subject to physical effects.

Recovering the expected solution to the original natural language problem is a strong indication that the formalization models the problem correctly, though the checker may sometimes reject answers that are in principle correct but missed some implicit assumptions that the original solution made.

Autoformalization of physical models can have tremendous impact on reliable autonomy. Unlike in mathematics, validating a hypothesized CPS formalization is possible with comparisons between real measurements and predicted measurements during test runs [99]. Easy formalization would allow every modular component to be modeled formally, with all assumptions explicit. Verification could lead to safer systems and synthesis could make control system design easier while ensuring no edge cases are missed relative to the formal model.

8.1 Overview

This section explains by example what the process of autoformalizing a physics problem involves. Consider the following physics question [66, problem 1.2], which will serve as a running example:

> "A point traversed half the distance with a velocity v_0. The remaining part of the distance was covered with velocity v_1 for half the time, and with velocity v_2 for the other half of the time. Find the mean velocity of the point averaged over the whole time of motion."
> *Solution:* "$v_{\mathsf{avg}} = \frac{2v_0(v_1+v_2)}{2v_0+v_1+v_2}$"

The physical motion described here is hybrid, with continuous dynamics (motion at velocity v_0, v_1, and finally v_2) as well as discrete transitions (changing velocity under specified conditions). Additionally, there are non-trivial conditions describing *when* the discrete transitions happen. We seek to symbolically model this motion formally.

We use hybrid games [63, 153] to express the control problems. Like in the rest of this thesis, we use the logic formalization of hybrid games [121] (as opposed to the automaton-based formalization [63]). This may be beneficial because it is closer to code, which LLMs have been more exposed to and fine-tuned for. Model 10 shows an example of a dGL hybrid game which formalizes the running example.

Model 10 Running Example: Finding Mean Velocity

```
      setup | 1  $\langle\, x := 0\,;\, d_{\mathsf{vavg}} := 0\,;$
    phase 0 | 2    $\{x' = v_0, d'_{\mathsf{vavg}} = v_{\mathsf{avg}}\}\ ;\ \ d_h := x\,;$
    phase 1 | 3    $t := 0\,;\ \{x' = v_1, d'_{\mathsf{vavg}} = v_{\mathsf{avg}}, t' = 1\}\ ;$
    phase 2 | 4    $t_h := t\,;\ \{x' = v_2, d'_{\mathsf{vavg}} = v_{\mathsf{avg}}, t' = 1\}\ ;$
transitions | 5    $?t = 2t_h\,;\ ?x = 2d_h$
   win when | 6  $\rangle\, d_{\mathsf{vavg}} = x$
```

In this game, a player (canonically called Angel) runs physical dynamics per the problem specification, moving the point's position x with velocity v_0 in phase 0 on Line 2, with velocity v_1 in phase 1 on Line 3, and with velocity v_2 in phase 2 on Line 4. Angel is forced to transition between phases under the exact conditions specified by the problem because of *tests* (assertions) on Line 5. These assertions use *auxiliary variables* t_h to keep track of the time elapsed in phase 1,

t to keep track of time elapsed in phase 1 and phase 2 combined, and d_h to keep track of half the distance covered. On Line 5, test $?x = 2d_h$ ensures that the first phase transition occurred when half the distance was covered. The test $?t = 2t_h$ ensures that the second transition happened at half the total run time of phases 1 and 2. If Angel fails either test condition, she immediately loses the game.

Notice that there is a variable v_{avg} representing average velocity, and Line 6 says that Angel will win the game only if v_{avg} was correct, as checked using auxiliary variable d_{vavg}, which tracks the displacement covered by a particle traveling at v_{avg} through all phases. The value that v_{avg} must hold for Angel to win matches the solution to the original problem, $v_{\mathsf{avg}} = \frac{2v_0(v_1+v_2)}{2v_0+v_1+v_2}$. In this example, there happens to be no adversarial dynamics (Angel is the only player making decisions). However, in problems with adversarial nondeterminism such as unpredictable environmental factors or controller latency, a second, symmetric, adversarial player (canonically called Demon) models the situation by resolving nondeterminism with the goal to make Angel lose. Adversarial dynamics are also useful for optimization problems, e.g., "choose the launch angle that minimizes drift", which can be modeled as Angel and Demon competing to find this angle.

Having formalized the problem as a dGL hybrid game, we next describe how symbolic execution enables a check for whether the model aligns with the natural language question. For the fragment of dGL where most of the chosen benchmark problems lie (loop-free, polynomial solutions), symbolic execution is decidable and Chapter 5 describes how to perform it. Backwards symbolic execution over a game lets us compute the conditions under which Angel (or dually, Demon) can win the game. For example, symbolic execution of the subgame $\langle Line\ 5\rangle d_{\mathsf{vavg}} = x$ at the end of Model 10 evaluates to $x = d_{\mathsf{vavg}} \land t = 2t_h \land x = 2d_h$ which is precisely the weakest precondition starting from which Angel passes the tests of Line 5 and ends the game in a state where she wins. Consider the precondition ϕ of Line 1 computed in this way. Angel should be able to win the game exactly when v_{avg} is set to the correct solution, i.e., $v_{\mathsf{avg}} = \frac{2v_0(v_1+v_2)}{2v_0+v_1+v_2} \leftrightarrow \phi$. Such a check is implemented using Z3 and Mathematica.

Observe that even a minor modeling mistake can result in this check failing. In fact, symbolically executing Model 10 reveals that it is actually equivalent to $\top$. It suffers from a subtle exploit where Angel can always run each phase for 0 time, in which case $x_{\mathsf{avg}} = x = 0$ regardless of what v_{avg} is set to. The problem is that the model lets Angel choose the distance that the particle will travel, as her choice at Line 2 determines the value of half-distance d_h, which in turn determines the distance that phases 1 and 2 must cover. The fix is simple: the assignment $d_h := x$ on Line 2 should be turned to a test $?d_h = x$. The automated check was able to catch this difference and is a strong signal of correctness, especially in combination with manual review. The automated checker has a second test to ensure the system is not in stasis, ruling out empty games that exploit the symbolic execution test by setting the win condition directly to the desired solution. For this example, the second test checks via static analysis that at least two variables are written to, since *any* reasonable model will at least modify the variables representing the particle's position and time. The checker is provided with the minimum number of variables that must be written to and the expected solution for every benchmark problem.

Finally, we see the benefit of formal modeling when the model with Line 2 corrected is still not equivalent to the expected solution. The problem is that the textbook solution is correct *only* *under some unstated assumptions*: distance d is non-zero, velocity v_0 is positive, and $v_1 + v_2$ is also positive. Finally, Model 11 makes these assumptions. Under the assumptions of Model 11, Line 1, the modal formula from Model 11, Line 2 to Line 7 is equivalent to $\frac{2v_0(v_1+v_2)}{2v_0+v_1+v_2}$. In unverified or informal physics, subtle modeling glitches are widespread. As a human, it is easy to neglect a measure zero case like $d_h = 0$ during an intermediate calculation, but formalization forces this case to be explicit.

Model 11 Repaired Model: Finding Mean Velocity

```
     assum2 | 1  $(d_h > 0 \land v_0 > 0 \land v_1 + v_2 > 0) \rightarrow$
      setup | 2  $\langle x := 0 ; d_{\mathsf{vavg}} := 0 ;$
    phase 0 | 3    $\{x' = v_0, d'_{\mathsf{vavg}} = v_{\mathsf{avg}}\}$ ; $?d_h = x ;$
    phase 1 | 4    $t := 0 ; \{x' = v_1, d'_{\mathsf{vavg}} = v_{\mathsf{avg}}, t' = 1\}$ ;
    phase 2 | 5    $t_h := t ; \{x' = v_2, d'_{\mathsf{vavg}} = v_{\mathsf{avg}}, t' = 1\}$ ;
transitions | 6    $?t = 2t_h ; ?x = 2d_h$
   win when | 7  $\rangle\, d_{\mathsf{vavg}} = x$
```

This example begins to demonstrate why formalizing a hybrid physical problem is *not* a simple translation task. Introducing the right auxiliary variables, absent in the specification, is often necessary for a clean, tractable formalization. All the pieces of the model must fit together precisely, e.g., even making a weak inequality strict can transform a correct model into one admitting no runs [149]. Correctness requires careful attention to the subtleties of the problem specification, e.g., benchmark problem 10 (available in the online artifact [73]) has one variable representing a *displacement* and another, representing a *distance*. The former is a signed number while the latter must be modeled as non-negative. Models must prune mathematically degenerate cases using background knowledge of the physical world, e.g., by eliminating imaginary roots of polynomials computing real quantities. Additionally, there are many ways to model the same problem, and the right choice of coordinate system or frame of motion can be crucial to writing a model for which symbolic reasoning is tractable.

Given the complexity of the task, it is unclear whether LLMs can succeed at all, making a careful evaluation necessary. We create a benchmark suite drawing from textbooks and problem sets [66, 90, 96], choosing tricky problems with varying structures to systematically identify the strengths and weaknesses of LLMs on autoformalization.

8.2 Methodology

The autoformalization system, shown by Fig. 8.1, takes as input the natural language specification of the physical problem to model, and produces either a correct dGL formalization or a failure (no formalization found that passed the checker). It uses two types of LLM queries: one is *Propose Formalization*, and the second one is *Revise Formalization*. The artifact [73] shows all the prompt templates used.

The autoformalization system first queries the LLM to propose the formalization using the *Propose Formalization* query, providing up to four solved examples as part of the prompt. It provides an LLM with the natural language description of the problem and asks it to produce a dGL formula, including in the prompt common dGL syntax mistakes and their remedies. An example of a problem description the prompt would use based on the running example is

> "An object A starts from rest and travels a distance d. For the first half of the distance, it travels with velocity v0. For the second half of the distance, it spends half the time traveling with velocity v1 and half the time traveling with velocity v2. Another object B tracks A. Its motion has the same duration, starting, and ending points as A, but it travels with uniform velocity vavg. Let vavg remain a free variable."

The prompt also includes up to four solved examples. The returned formal model is passed to the KeYmaera X dGL parser for a syntax check.

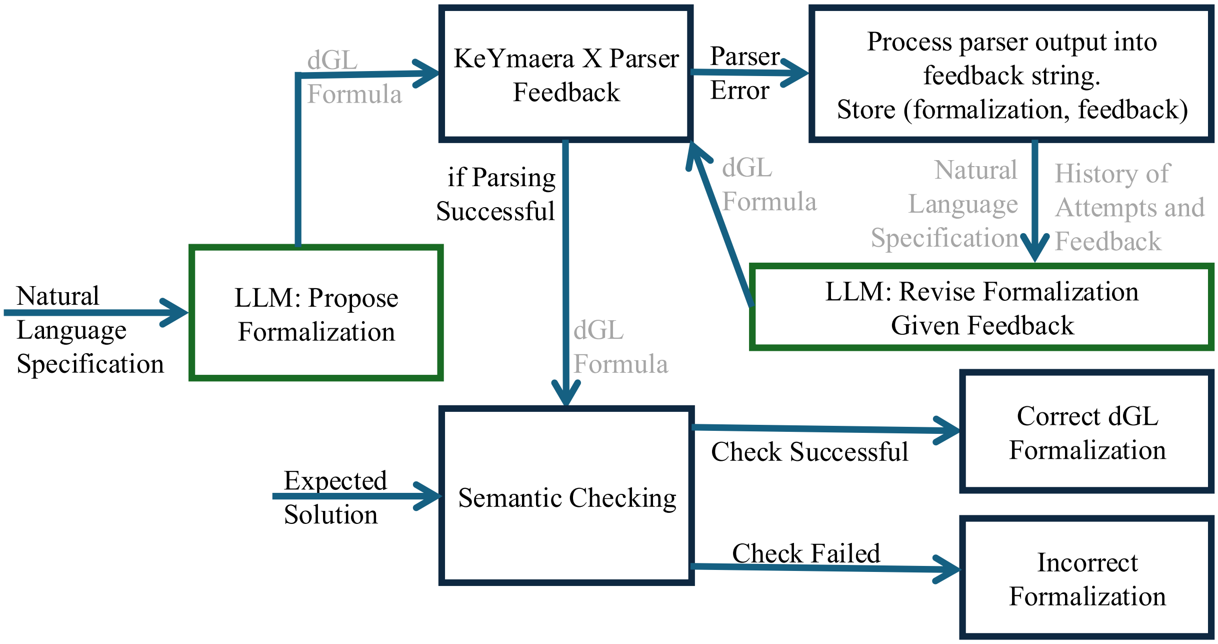


Figure 8.1: Autoformalization pipeline

If the syntax check fails, then the parser output is processed into short feedback strings. An example of a parser feedback string is

> "The input formula contains an unsupported Unicode character (possibly {`bad_char`}). Use only ASCII characters."

Where `bad_char` is a parameter extracted from parser output. The incorrect formalization along with the feedback is then appended to a list of previous failed formalization attempts.

The LLM is prompted to repair the formalization given past attempts and feedback using the *Revise Formalization* query. The prompt first provides up to two examples of repair. Then it provides the natural language question, and finally past incorrect proposals along with the feedback indicating why these proposals were wrong. The LLM replies with a new, repaired dGL game, which is again passed to the parser for checking. This syntactic repair loop is allowed to run at most three times, after which autoformalization fails. A full implementation, along with examples, is available in the online artifact [73].

If the syntax check succeeds, then the dGL formula is passed on to the semantic evaluation system. This system takes as input the expected symbolic solution to the problem along with the proposed dGL formula. For our running example, the expected solution consists of the formula $v_{\mathsf{avg}} = \frac{2v_0(v_1+v_2)}{2v_0+v_1+v_2}$, to test against for equivalence, and minimum expected number of variables that are written to, 2. If the checker finds that the dGL formula is equivalent to the expected solution under the initial assumptions of the formula and has enough variable writes, then it declares that the formalization is acceptable. The pipeline is implemented using Delphyne [87] and KeYmaera X [52], which in turn uses Z3 [39] and Mathematica.

8.3 Related Work

Given a natural language description, formal physics models are generally written manually. Prior work has studied the use of structured natural language to perform formalization by rule-based transformations [28, 109], but this limits flexibility in input specification. Recent progress in NLP (LLMs) raises the prospect of automating formalization directly from unstructured natural language.

The use of LLMs for autoformalizing mathematics has received significant research attention [89, 162], including for purely continuous systems with partial differential equation [42, 65]. Recent work also uses LLMs to synthesize contracts for cyber-physical systems [30, 107, 110]. However, we focus on LLM-based autoformalization for *physics models* of hybrid systems, with discrete transitions and continuous dynamics faithfully represented, which to the best of our knowledge has not yet been studied. Compared to general mathematics and LTL specifications, we expect this domain to be more data-scarce, with few examples of differential dynamic logic system formalizations and even fewer examples of differential game logic games available on the Internet. Additionally, we expect CPS model formalization to have a more operational flavor, modeling physical realities required to make a system work rather than abstract concepts that can be simplified for theoretical convenience (mathematics) or focused on only output behavior (LTL contracts). This difference makes it worth investigating the hybrid systems modeling domain.

8.4 Evaluation

We propose 20 benchmarks derived from challenging problems in undergraduate physics textbooks and problem sets [66, 90, 96]. The online artifact [73] lists them. Phrasing is changed to focus on modeling the physical situation described in the problem (rather than on solving for a variable) and the numbers are changed to induce rational answers to avoid failures of the automated check over numerical imprecision. Each benchmark is accompanied by an expected symbolic solution that the model should be equivalent to and the minimum expected number of variables that must be written to. Four additional problems, separate from the benchmarks, serve as examples for few-shot prompting.

We run the LLM autoformalization system using OpenAI GPT 4o, GPT 4.1 (best available

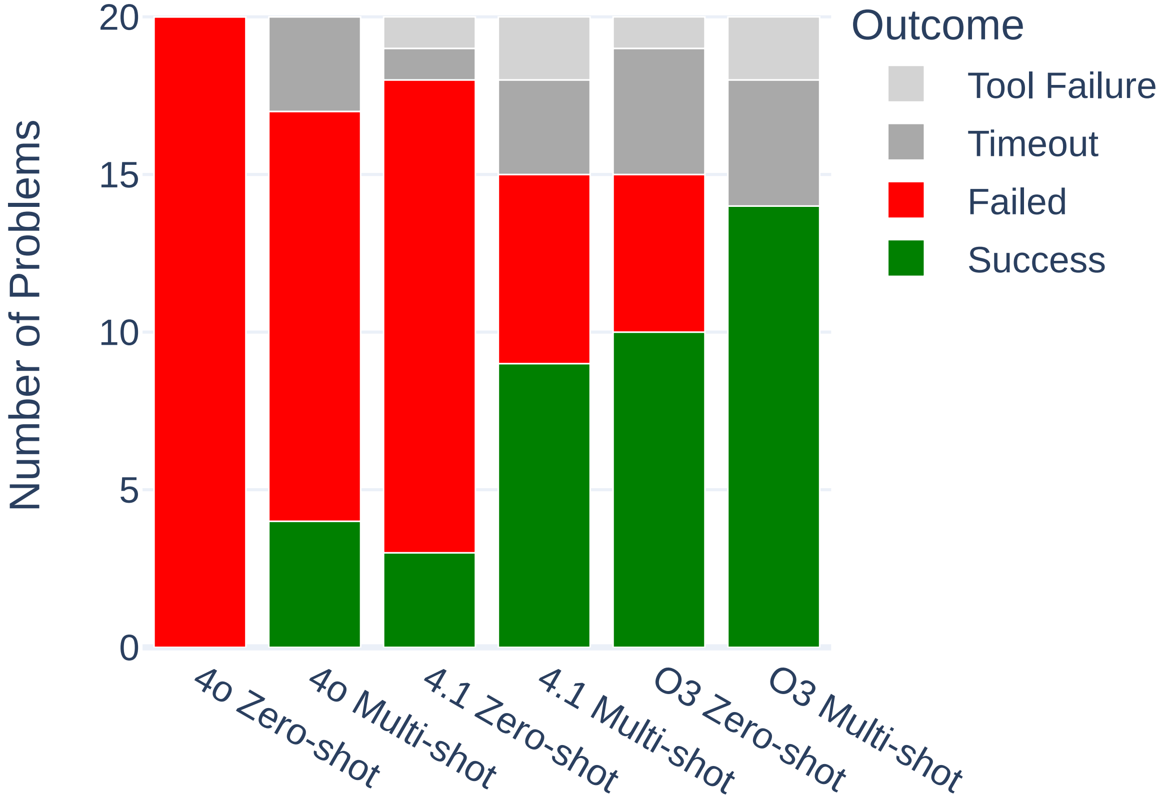


Figure 8.2: Outcomes of autoformalization by GPT 4o, GPT-4.1 and o3, with zero-shot and multi-shot prompting.

OpenAI non-reasoning model) and OpenAI o3 (best available OpenAI reasoning model) on the benchmark set, getting successively improving results. Outcomes are classified into 4 buckets:

1. *Success*: Correct, checker-certified formalization.
2. *Failed*: autoformalization failed either the syntax or semantic check.
3. *Timeout*: The solver timed out after 3 minutes of symbolic execution. Symbolic execution uses real quantifier elimination, which is doubly exponential [38], and can result in timeouts.
4. *Tool Failure*: The solver encountered an expression that it could not automatically symbolically execute (e.g., an ODE with a square root on the right-hand side).

Fig. 8.2 shows the outcome. The autoformalization system is sampled five times independently with temperature 1, and the best outcome is reported (where tool failure and solver timeout are considered better than failure). For each sample, the LLM is given three chances to repair syntax if the KeYmaera X parser reported an error. We evaluate two different prompts, one using few-shot prompting considering four examples (*multi-shot*) and another one without examples (*zero-shot*).

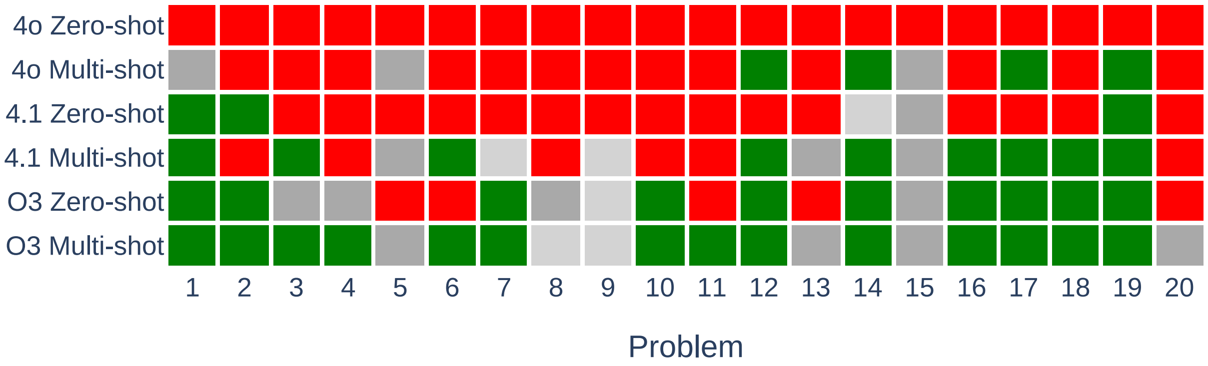


Figure 8.3: Problem outcome by model.

GPT 4o without few-shot prompting fails on every benchmark. Few-shot prompting significantly improves performance for all models. With few-shot prompting o3 has a success rate of 70%, which is higher than GPT 4.1's 45% and GPT 4o's 20%. Our results support that the improvements and the introduction of reasoning LLM models significantly increase their success rate on mathematical tasks, enabling strong performance in the autoformalization of hybrid games.

8.5 Discussion

Fig. 8.3 shows the outcome of each problem by model. We can observe that models found the same problems hard. There are six problems that no model solves. For these, the best performing model produces a formalization that could potentially be correct but is too complex for the checker to verify. The problems can be classified into three groups that explain where their complexity arises from.

1. Optimization Problems: Problem 5 and 20.
2. Non-polynomial continuous dynamics: Problem 8 and 9.
3. Complicated dynamics: Problem 13 and 15.

The first category, for example, includes problem 5, which is about identifying the launch angle of a boat that minimizes drift, modeled by adversarial Angel trying to pick a better angle than Demon, who must set the optimal angle to win. The interplay of the two players leads to combinatorial branching over possible sequence of events, while continuous dynamics occur under constraints preventing control blocking. These constraints make symbolic reasoning about the differential equations particularly computationally expensive.

An example of the second category is problem 8, which is about drones arranged in an equilateral triangle that cyclically pursue each other with constant radial velocity, collapsing inwards in increasingly rapid revolutions, till they crash. Their dynamics, often modeled with a non-polynomial square root expression on the right side of an ODE, cannot be handled automatically by the tool. Problem 13 is an example from the third category. It has three phases of dynamics, including one where acceleration has a continuous dependence on time. Such a differential equation is more expensive to symbolically evaluate, making a timeout the likely outcome.

Sometimes a problem fundamentally requires capabilities that a solver lacks, and the solver must be extended. Other times a difficult but crucial part of formalizing problems is choosing the right abstractions and representations that let proofs succeed. For the six failing problems, the solution likely lies in a combination of these methods. The solver must be extended to better automatically handle, for instance, non-polynomial dynamics by using continuous invariant generation [142]. Complementing this, the LLM should be guided to use tricks like polar coordinates and auxiliary variables to rewrite dynamics in friendlier forms, and to also provide checking hints such as continuous invariants.

Unsolved benchmarks provide a roadmap for improvement in both these directions. Additionally, the rapid improvement in success rate displayed by better LLM models suggests that autoformalization of hybrid games will continue to improve with advancements in LLMs.

8.6 Conclusion

This work experimentally studies the ability of LLMs to formally model physical behavior given a natural language specification. The results (70% success rate) are encouraging, and suggest that CPS model autoformalization using LLMs could be an impactful direction for future research.

Beyond the broader adoption of formal methods in offline controller and control envelope design, physics autoformalization enables a new class of applications, where autonomous systems autoformalize unfamiliar situations they face in the open world and derive formally justified control decisions to respond to them. For such applications, where the *physical environment* must be formalized at runtime, the input to autoformalization would be vision and sensor data rather than a natural language description. LLM autoformalization from such multimodal input data provides an interesting future research direction.

With this chapter, we explore the possibility of extending the synthesis pipeline all the way back to an informal natural language description of the control problem, asking whether the formal game specification itself can be generated automatically before the envelope is synthesized and verified. An open challenge remains recovering soundness guarantees for this step. In the course of this exploration, interesting future research directions are revealed, which we discuss in Chapter 9.

Chapter 9

Conclusion

We briefly summarize the contributions of this thesis, and then discuss interesting directions for future work.

9.1 Summary

Control envelopes are sets of control solutions. Verified control envelopes provide a path to verifiably safe control. When developing a formally verified controller (e.g., a train control system) one generally has to consider primary, mission-critical requirements (e.g., safety) along with secondary or evolving requirements (e.g., smoothness of the ride). A possible approach to synthesizing a controller that is formally correct with respect to the primary requirements is: (1) first identify a maximal *set* of controllers formally verified to satisfy the primary requirement, i.e., a *verified control envelope* for the requirement, and then (2) optimize within this set for the controller choice that best meets the secondary requirements (e.g., using machine learning [50, 131, 152]).

This thesis made symbolic verified control envelope synthesis possible for richly expressive control system models, and developed methods to increase scalability. Given the physically possible control possibilities and environment behavior, its techniques computed a set of control solutions that satisfied the control contract. It introduced the first approach applicable at the level of imperative program-like specifications[1], enabling synthesis for richly expressive control system models.

9.1.1 Train Control

As a concrete example of control envelopes for real-world applications, we considered a train controller that must stay within its *motion authority* (stop before some stopping points) to avoid collisions with other trains (Chapter 3 [69]). The controller can freely choose some engine acceleration or dynamic braking, but is not able to revise its decision for some bounded time latency

---

[1] There is envelope synthesis work in the automaton community [13], though the input is at a different level of abstraction (automata, not program-like logic). It is interesting future work to investigate what ideas can transfer between the very different formalisms in the future.

(typically on the order of seconds) after its decision. We created a control envelope that answered: given the current state, what values of acceleration/braking are acceptable so that the train has a way to always remain safe? We had to account for all the competing influences that govern train motion. For example, uphill slope decreases velocity, which decreases resistance, which permits a more rapid increase in velocity, slope and curve resistance effects. These complex interactions illustrate why safe and efficient control envelope synthesis is hard, and why it requires precise formal reasoning.

Our envelope was *symbolic*, supporting unbounded state space using variables that can represent any real number, and *parametric* in these variables (e.g., maximum train braking magnitude b_{max}). The control envelope captured an entire *set* of control solutions, so in the same state, it allowed multiple different choices for acceleration or braking, and each choice led to a different correct controller.

The thesis then turned to making control envelope synthesis possible, which involved answering questions such as the following. Given a control system as input[2], how can we *represent* its control envelope, a potentially infinite control solution set, in a way that is *compositional* (synthesized/verified by the synthesis/verification of its parts)? How do we *define mathematically* what it means for this control envelope to be *correct*? How can we perform *automatic synthesis*? Are there ways to make the synthesis *scalable*? The thesis answered these questions as summarized below.

9.1.2 Subvalue Maps

The input to the control envelope synthesis problem is a *two-player game* between the controller and the environment [70, 108] (Chapter 6 [76]). This is because, intuitively, a game defines the shape of a control problem without specifying control strategies (which must be synthesized). The controller has some control moves, the environment adversarially resolves nondeterminism, and the controller's winning condition consists of meeting some critical control requirements. Synthesizing a control envelope corresponds to identifying the nondeterministic policy indicating what plays the controller can make at each step so that it can still win the game. Input games are represented in an imperative program-like language, *differential game logic* [121], that decomposes into subgames recursively along familiar syntactic structures: loops, sequential composition, and branching. As a symbolic, compositional representation of a control envelope that we can reason about recursively along the syntactic structure of the game, the thesis proposes the *inductive subvalue map* (Chapter 6 [76]), a map from subgames to formulas that hold exactly when it is possible to play the corresponding subgame and win. This map induces a nondeterministic policy for the controller: to check if playing into a subgame g is permitted in a given state, we only need to check that the subvalue associated with g holds in that state. We define a *correct* control envelope using recursive logical constraints on the subvalue formulas that ensure playing per *any* controller in the envelope guarantees that the controller player never gets stuck with no acceptable next action and can *always continue towards winning the game within the envelope*. We identify a backward precondition calculus that derives the subvalue map

---

[2] We considered a very general class of input control problems: those that can be expressed as hybrid games, in differential game logic [121].

for a given differential game logic game and winning condition.

9.1.3 Scaling with LLMs

The precondition calculus gives us a general, formally justified symbolic control envelope synthesis framework. But scalability remains an issue because of dependence on difficult subroutines like quantifier elimination (doubly exponential) [38], and invariant generation (undecidable, infinite search space). We attack the scalability bottlenecks in three different ways (Chapter 7). (1) Invariant generation is performed by LLMs, leveraging their physics problem-solving abilities, coupled with backtracking search. (2) Quantifier elimination and simplification are also replaced with LLM queries, with soundness maintained by retrospective formal verification. (3) Verification can itself become a bottleneck, for which our solution is to have the LLM use an interactive theorem prover to break down the proof into tractable subgoals (Chapter 4).

9.1.4 Control Envelope Synthesis via Angelic Refinement (CESAR).

We identify another useful technique to synthesize for complicated games: *refinement*, or rewriting the game so that it is harder for the controller (maintaining soundness) but easier to symbolically reason about. For controllers that poll repeatedly in a single loop of some maximum time latency, we in fact identify systematic refinements via the CESAR algorithm (Chapter 5 [70]). We find that for controllers with *action permanence* (a hybrid analog of idempotence) and dynamics with polynomial solutions, CESAR is decidable.

9.2 Future Work

Control envelopes provide a practical path towards verifiably safe autonomy. They make it possible to safeguard controllers, even those based on legacy code or machine learning, by runtime monitoring against the verified envelope. Synthesis removes a large barrier to adoption: designing envelopes in the first place. However, some barriers remain.

9.2.1 Formalizing for the Complexities of Reality.

Formalization work with a similar flavor to this thesis will let control envelopes model increasingly complex real-world situations (partial observability, multiplayer coordination) with increasing scalability (efficiency improvements to the synthesis procedure for various problem classes, harnessing advances in AI via intersymbolic algorithms). How should we define probabilistic control envelopes that assure safety with some confidence interval given a probabilistic model of the environmental uncertainties? Extending the precondition calculus to follow the rules of stochastic reasoning [119] and statistics provides a path. How do we model control envelope synthesis for multi-agent, distributed systems where each agent may be open to collaborating? Statically agreed-upon contracts can be synthesized by treating all the collaborative control agents as a *single* controller. After carefully formalizing the answer to these questions, how do we perform scalable synthesis for the new kinds of envelopes? Building on Chapter 7, an answer lies in careful algorithm design that combines the use of AI and verification (intersymbolic AI [124]).

9.2.2 Control Envelope Synthesis for the Masses.

Reliable autoformalization tools can put envelopes within reach of even audiences without formal methods expertise. Before synthesizing/verifying an envelope, one still needs a *formal model* of their system (the environment, control possibilities, and the desired guarantees). Today, writing such models is hard, a bottleneck for broader adoption of control envelopes, and more generally, formal methods. Meanwhile, potential users often already have the relevant information in a different form, e.g., design documents, log data, or simulation code. LLMs now show promise at writing formal models from informal natural language descriptions (Chapter 8 [72]). If we can use LLMs to *automatically* create formal models, with iterative feedback, users outside formal methods could access verified control envelopes and use them for downstream applications. Two flavors of challenges arise for future work: making autoformalization *reliable*, e.g., with formal consistency checks and comparison to logs, and making it *usable*, e.g., with visual representations that help users spot errors.

9.2.3 Applications.

Initial case studies demonstrate the benefits and costs of using control envelopes, and pave a path for future adopters. The verified *train* control envelope [69] provides a concrete example of a control envelope with the complexities of real-world dynamics. Similar projects would make control envelopes useful for other applications. Another future application is that, when *fine-tuning foundation models* for robot control using reinforcement learning (RL), feedback on the model's control decisions should reason over unbounded time to catch a mistake now even if its consequences won't be seen till far in the future. Control envelopes batch-compute the solutions for the entire control problem over infinite time once and for all, so that later checking a single action for RL feedback is efficient.

9.2.4 Optimized Controller Extraction.

Finally, how to retrieve a controller from the synthesized symbolic envelope while optimizing for secondary requirements is an interesting future research direction. A technical path lies in extending work that quantitatively estimates how well states adhere to an end objective in a dL system [99] to also handle games. Controllers that achieve high performance on secondary requirements while still ensuring correctness would incentivize adoption in industry. Ultimately, optimized controller extraction techniques together with the work of this thesis can help realize the goal of deploying formally guaranteed, high-performance controllers in complex systems.

Appendix A

CESAR (Appendix)

A.1 Reduce Operation

We define reduce after first introducing two helper functions that it requires. Function $\triangleright(a, b)$ attempts to simplify $\mathrm{FOL}_{\mathbb{R}}$ formula a to $\mathcal{P}_{\mathbb{R}}$ assuming that b holds. The second helper function, `odereduce` (Def. A.1), isolates the action of reduce on differential equations. Since it is solely continuous programs that could lead to reduce failing to produce an exact solution, the exact bit of reduce depends on `odereduce`. Fig. A.1 shows the definition of reduce, eliding the exact bit, which is simply true if all of the `odereduce` calls that reduce makes are exact, and false otherwise.

Definition A.1 (ODE reduction)**.** Let $\langle\![\alpha]\!\rangle \in \{\langle\alpha\rangle, [\alpha]\}$, $\bowtie \in \{\leftrightarrow, \rightarrow\}$, and A, Q, and P be formulas in quantifier-free real arithmetic. An *ODE reduction oracle* `odereduce` is a function such that

$$A \rightarrow (\mathtt{odereduce}(\langle\![\{x' = f(x) \& Q\}]\!\rangle P, A) \bowtie \langle\![\{x' = f(x) \& Q\}]\!\rangle P)$$

is valid. When $\bowtie$ is $\leftrightarrow$ then `odereduce` is exact, otherwise it is approximating.

For solvable ODEs, `odereduce` is implemented as an exact oracle in Eq. (A.1).

$$\begin{aligned}
&\mathtt{odereduce}([\{x_1' = \theta_1, \cdots, x_n' = \theta_n \& Q\}]P, A) = \\
&\qquad \forall t\big(\mathrm{subst}(A, t) \land t \geq 0 \land \forall 0{\leq}s{\leq}t\,\mathrm{subst}(Q, s)\big) \rightarrow \mathrm{subst}(P, t) \\
&\mathtt{odereduce}(\langle\{x_1' = \theta_1, \cdots, x_n' = \theta_n \& Q\}\rangle P, A) = \\
&\qquad \exists t\big(\mathrm{subst}(A, t) \land t \geq 0 \land \forall 0{\leq}s{\leq}t\,\mathrm{subst}(Q, s)\big) \land \mathrm{subst}(P, t) \\
&\text{where } \mathrm{subst}(f, t) = f\{\int_{x_i}^{t} \theta_i \cdot dt \,/\, x_i\},\ t \text{ fresh},\ \langle\![\alpha]\!\rangle \in \{\langle\alpha\rangle, [\alpha]\}
\end{aligned} \tag{A.1}$$

In the general case, Pegasus [142], a tool that automatically generates *ODE invariants*, can often produce a formula satisfying the specification of `odereduce`. This may come at the cost of lost precision, possibly requiring reduce to set exact to false.

Theorem A.1 (Correctness of reduce)**.** *For any loop-free* dGL *formula F and assumptions $A \in \mathcal{P}_{\mathbb{R}}$ the function* `odereduce` *either sets* exact=$true$ *and the formula $A \rightarrow (\mathsf{reduce}(F, A) \leftrightarrow F)$ is valid, or else it sets* exact=$false$ *and the formula $A \rightarrow (\mathsf{reduce}(F, A) \rightarrow F)$ is valid.*

$$
\begin{aligned}
&\text{Let } \langle\!\![\alpha]\!\!\rangle \in \{\langle\alpha\rangle, [\alpha]\}\\
\mathsf{reduce}(\langle\!\![\alpha;\beta]\!\!\rangle P, A) &= \mathsf{reduce}(\langle\!\![\alpha]\!\!\rangle(\mathsf{reduce}(\langle\!\![\beta]\!\!\rangle P, \top)), A)\\
\mathsf{reduce}([\alpha\cup\beta]P, A) &= \mathsf{reduce}([\alpha]P, A) \wedge \mathsf{reduce}([\beta]P, A)\\
\mathsf{reduce}(\langle\alpha\cup\beta\rangle P, A) &= \mathsf{reduce}(\langle\alpha\rangle P, A) \vee \mathsf{reduce}(\langle\beta\rangle P, A)\\
\mathsf{reduce}(\langle\!\![x := e]\!\!\rangle P, A) &= \triangleright(P\{e/x\}, A)\\
\mathsf{reduce}([?f]P, A) = \triangleright(f \rightarrow P, A) &\quad \mathsf{reduce}(\langle ?f\rangle P, A) = \triangleright(f \wedge P, A)\\
\mathsf{reduce}([\{x' = f(x) \& Q\}]P, A) &= \triangleright(\texttt{odereduce}([x' = f(x)\&Q]P, A), \top)\\
\mathsf{reduce}(\langle\{x' = f(x) \& Q\}\rangle P, A) &= \triangleright(\texttt{odereduce}(\langle\{x' = f(x)\&Q\}\rangle P, A), \top)\\
\mathsf{reduce}([\alpha^d]P) = \mathsf{reduce}(\langle\alpha\rangle P) &\quad \mathsf{reduce}(\langle\alpha^d\rangle P) = \mathsf{reduce}([\alpha]P)\\
\mathsf{reduce}(P \wedge Q, A) = \triangleright(P \wedge Q, A) &\quad \mathsf{reduce}(P \vee Q, A) = \triangleright(P \vee Q, A)\\
\mathsf{reduce}(P \rightarrow Q, A) = \triangleright(P \rightarrow Q, A) &\quad \mathsf{reduce}(\neg P, A) = \triangleright(\neg P, A)
\end{aligned}
$$

Figure A.1: Definition of reduce (exact elided). Notation $P\{e/x\}$ indicates P with unbound occurrences of x replaced by expression e. `odereduce` isolates the effect of reduce on ODEs. $\triangleright$ simplifies and quantifier eliminates P assuming A.

A.2 Parachute: reduce of Non-solvable Dynamics

The parachute benchmark (Appendix A.3.3) has non-solvable dynamics whose exact solution involves hyperbolic functions. The differential equations are $\{x' = -v, v' = -rv^2 + g\}$ and admit the solution:

$$\left\{ v(t) \rightarrow \frac{\sqrt{g}\tanh\left(\sqrt{g}\sqrt{r}t + c_1\sqrt{g}\sqrt{r}\right)}{\sqrt{r}}, x(t) \rightarrow c_2 - \frac{\log\left(\cosh\left(\sqrt{g}\sqrt{r}(t + c_1)\right)\right)}{r} \right\}.$$

This solution does not belong to the decidable fragment of arithmetic, and QE is not guaranteed to terminate. Reduce is able to still return preconditions over this ODE by using differential invariants generated by Pegasus.

The first call to reduce is for program $[\{x' = -v, v' = -rv^2+g, t' = 1 \,\&\, v > 0 \wedge t \leq T\}](x < 0 \vee v < m)$. A call to Pegasus identifies the first degree Darboux polynomials of this ODE to be $\left\{\sqrt{g}, \sqrt{g} - \sqrt{r}v, \sqrt{g} + \sqrt{r}v\right\}$. As a heuristic to retain only the most useful polynomials, we filter out terms that are purely constant. The Darboux rule of dL tells us that the following formulas are now invariants of the differential equation system: $\left\{\sqrt{g} - \sqrt{r}v \geq 0, \sqrt{g} + \sqrt{r}v \geq 0\right\}$.

When do these invariants imply the desired post condition, $x < 0 \vee v < m$? Mathematically, the answer is the formula

$$\sqrt{g} - \sqrt{r}v \geq 0 \wedge \sqrt{g} + \sqrt{r}v \geq 0 \rightarrow x < 0 \vee v < m.$$

To find an invariant that implies this formula, we eliminate variables x and v using quantifier elimination. The resulting expression is $m > 0 \wedge g \geq 0 \wedge g < m^2 r$. Thus, reduce returns $\sqrt{g} - \sqrt{r}v \geq 0 \wedge \sqrt{g} + \sqrt{r}v \geq 0 \wedge m > 0 \wedge g \geq 0 \wedge g < m^2 r$ as a precondition. This expression is invariant under the problem's dynamics and implies the postcondition $x < 0 \vee v < m$.

A.3 Benchmarks

This section lists all the benchmarks proposed. It also shows the solution of each benchmark, annotated with the meaning of the solution expressions. The algebraic formulas presented are synthesized by CESAR automatically. The annotations are added manually for the convenience of the reader. Table A.1 provides a one-sentence summary for each benchmark problem.

Table A.1: Benchmark listing.

Benchmark	**Description**
ETCS Train	Kernel of the European Train Control System case study [126].
Sled	Swerve to avoid a wall.
Intersection	Car must either cross an intersection before the light turns red, or stop before the intersection. It may never stop at the intersection.
Curvebot	A Dubin's car must avoid an obstacle.
Parachute	Use the irreversible choice to open a parachute with some air resistance to stay at a safe velocity. Drawn form [53].
Corridor	Navigate a corridor with a side passage and dead end.
Power Station	Choose between producing and distributing power while dealing with resistance loss and trying to meet a quota.
Coolant	A coolant system in an energy plant must maintain sufficient heat absorption while meeting coolant discharge limits.

A.3.1 ETCS Train

The European Train Control System has been systematically but manually modeled and proved safe in the literature [126]. We consider the central model of this study and apply CESAR to automate design. It is listed as the running example Model 7.

A.3.2 Sled

Model 12 Sled must swerve to avoid an obstacle.

$$\begin{array}{r|l}
\textsf{assum} & \boxed{I} \land T_x > 0 \land T_y > 0 \land V > 0 \land T > 0 \to [\{ \\
\textsf{ctrl} & \quad (?\boxed{G_1}\,;\, v_x := -V \;\cup\; ?\boxed{G_2}\,;\, v_x := V)\,; \\
\textsf{plant} & \quad (t := 0\,;\, \{x' = v_x, y' = V, t' = 1 \;\&\; t \leq T\}) \\
\textsf{safe} & \}^*](y < T_y \lor x < -T_x \lor T_x < x)
\end{array}$$

Where I, G_1 and G_2 are to be synthesized.

This benchmark displays CESAR's ability to reason about state-dependent fallbacks. The slope of a hill is pushing a sled forward along the y axis with constant speed v_y. However, there is a wall blocking the way. It starts at y axis position T_y and extends along the x axis from $-T_x$ to T_x. The sled must swerve to avoid the obstacle. It can either go left (with velocity $-V$) or right (with velocity V). Which action is best depends on where the sled already is. Swerving left is a safe strategy when the sled can pass from the $-x$ side, mathematically, $T_y > T_x + x + y$. Likewise, swerving right is a safe strategy when $T_y + x > T_x + y$. Neither action alone gives the optimal invariant, but CESAR's I^0 characterization correctly captures the disjunction to find it. In synthesizing the guards for the actions, CESAR also identifies when it is still safe to switch strategies.

CESAR finds the solution below. The algebraic formulas presented are synthesized by CESAR automatically. The annotations describing their meaning are added manually for the convenience of the reader.

$$I \equiv y < T_y \wedge (\overbrace{T_y > T_x + x + y}^{\text{can swerve left}} \vee \overbrace{T_y + x > T_x + y}^{\text{can swerve right}}) \vee \overbrace{T_x < x \vee T_x + x < 0}^{\text{already safe}}$$

The guard for going left is

$$\begin{aligned} G_1 \equiv\; & \overbrace{T_x + 2TV + y < T_y + x \wedge T_x + T_y > x + y \wedge x >= 0}^{\text{despite going left one time period, can still pass from right}} \\ & \vee \overbrace{T_x + x + y < T_y \wedge \neg T_x + x = 0}^{\text{can pass from left}} \vee \overbrace{0 = T_x + x}^{\text{at the left boundary}} \\ & \vee \overbrace{T_x + TV < x \wedge T_x + T_y \leq x + y}^{\text{far enough right to stay right after this cycle}} \vee \overbrace{T_x + x < 0}^{\text{already left}} \end{aligned}$$

The guard for going right is

$$\begin{aligned} G_2 \equiv\; & \overbrace{x < 0 \wedge T_x + 2TV + x + y < T_y \wedge T_x + T_y + x > y}^{\text{despite going right one time period, can still pass from left}} \vee \overbrace{T_x < x}^{\text{already right}} \\ & \vee \overbrace{T_x + y < T_y + x \wedge T_x > x}^{\text{can pass from right}} \vee \overbrace{T_x + TV + x < 0 \wedge T_x + T_y + x \leq y}^{\text{far enough left to stay right after this cycle}} \\ & \vee \qquad \overbrace{T_y > y \wedge T_x \leq x}^{\text{right at the right boundary but there is time still to swerve}} \end{aligned}$$

There are some redundancies in these expressions, but they are correct, comprehensible, and can be simplified further.

A.3.3 Parachute

The parachute benchmark presents the challenge of dynamics with a solution that departs from the decidable fragment of real arithmetic. A person is free falling. At most once, they are allowed to take the action of opening a parachute. Once they do, their air resistance changes to p. The objective is to land at a speed no greater than m. The benchmark is inspired by one that appears in the literature (running example in [53]). CESAR uses Pegasus's Darboux Polynomial generation to solve the problem.

Model 13 Parachute benchmark from [53].

assum | $(T > 0 \land p > 0 \land g > 0 \land r > 0 \land m > 0 \land v > 0 \land \boxed{I}) \to \{$

ctrl | $\quad (?\boxed{G_1}\,;\ \textsf{skip}) \cup (?\boxed{G_2}\,;\ r := p)\,;$

plant | $\quad (t := 0\,;\ \{x' = -v, v' = -r \cdot v^2 + g, t' = 1 \;\&\; v > 0 \land t \le T\})$

safe | $\}^*](x \ge 0 \to v < m)$

Where I, G_1 and G_2 are to be synthesized.

CESAR finds the solution below. The algebraic formulas presented are synthesized by CESAR automatically. The annotations describing their meaning are added manually for the convenience of the reader.

$$I \equiv \overbrace{g \ge rv^2 \land m > (gr^{-1})^{1/2}}^{\text{either start below terminal velocity and terminal velocity without parachute is already safe}}$$

$$\lor \quad \overbrace{m > (gp^{-1})^{1/2} \land pv^2 \le g}^{\text{or air terminal velocity with parachute is safe and start below terminal velocity}}$$

The guard for not opening the parachute is that already terminal velocity without the parachute is safe, and the person has not exceeded terminal velocity yet.

$$G_1 \equiv m > (gr^{-1})^{1/2} \land rv^2 \le g$$

Likewise, the guard for opening the parachute is that terminal velocity with the parachute is safe.

$$G_2 \equiv m > (gp^{-1})^{1/2} \land pv^2 \le g$$

A.3.4 Intersection

Model 14 Intersection benchmark.

assum | $(B > 0 \land T > 0 \land v \ge 0 \land \boxed{I}) \to \{$

ctrl | $\quad ((?\boxed{G_1}\,;\ a := 0) \cup (?\boxed{G_2}\,;\ a := -B))\,;$

plant | $\quad (t := 0\,;\ \{x' = v, v' = a, \mathit{timeToRed}' = -1, t' = 1 \;\&\; t \le T \land v \ge 0\})$

safe | $\}^*]((\mathit{timeToRed} > 0 \land v \ne 0) \lor \neg(x = 0))$

Where I, G_1 and G_2 are to be synthesized.

The intersection benchmark is a simple example of a system with free choice and state dependent fallback. A car sees a yellow light, and must decide whether to coast past the intersection, or to stop before it. It may never stop at the intersection, which is located at $x = 0$. Whether it would be safe to stop or to coast depends on the car's current position and velocity.

CESAR finds the solution below. The algebraic formulas presented are synthesized by CESAR automatically. The annotations describing their meaning are added manually for the convenience of the reader.

$$
\begin{aligned}
I \equiv\ & \overbrace{x > 0}^{\text{safe if already past the intersection}} \\
\vee\ & \overbrace{x < 0 \wedge (v \leq 0 \vee v(\textit{timeToRed} \cdot v + x) > 0)}^{\text{otherwise, before the intersection, safe if velocity is already 0 or could coast past intersection}} \\
\vee\ & v > 0 \wedge \big(timeToRed > 0 \wedge (\\
& \overbrace{B < \frac{v}{\textit{timeToRed}} \wedge B \cdot \textit{timeToRed}^2 < 2(\textit{timeToRed} \cdot v + x)}^{\text{signal flips before car stops. Even braking, car crosses intersection before signal flips}} \\
\vee\ & \overbrace{v^2 + 2 \cdot B \cdot x \neq 0 \wedge B \geq \frac{v}{\textit{timeToRed}}}^{\text{stop somewhere that's not the intersection, the signal flips after the car stops}})
\end{aligned}
$$

The guard for coasting has many repeated clauses, so we first explain them before presenting the expression. Assuming v positive and $x \leq 0$, $C \equiv v^3 + 2Bv(Tv + x) < 0$ means that after one time period of coasting, the car still stops before the intersection. $D_1 \equiv 0 = 3\textit{timeToRed} + 2v^{-1}x$ means that the signal flips when the car is $2/3$rds along the way of coasting to the intersection. $D_2 \equiv v(3\textit{timeToRed} \cdot v + 2x) > 0$ means that the signal flips after the car is $2/3$rds along the way of coasting to the intersection. $D_3 \equiv v(3\textit{timeToRed} \cdot v + 2x) < 0$ means that the signal flips before the car is $2/3$rds along the way of coasting to the intersection. $E \equiv B = \textit{timeToRed}^{-1}v$ means that the signal flips exactly when the car halts if it starts braking now. $F \equiv 0 = \textit{timeToRed} + v^{-1}x$ means that the signal flips exactly when the car reaches the intersection by coasting. $G \equiv 2B + v^2(\textit{timeToRed} \cdot v + x)^{-1} = 0$ means that the car will be at the intersection when the signal flips if it starts braking now. $H_1 \equiv v(\textit{timeToRed} \cdot v + x) < 0$ means that if the car coasts, it will be before the intersection when the signal flips. $H_2 \equiv v(\textit{timeToRed} \cdot v + x) > 0$ means that if the car coasts, it will be after the intersection when the signal flips.

$$
\begin{aligned}
G_1 \equiv\ & \overbrace{v = 0}^{\text{already still and safe}} \quad \vee (v > 0 \wedge (\\
& \overbrace{x > 0}^{\text{already past intersection}} \quad \vee x <= 0 \wedge (\\
& D_1 \wedge (\overbrace{(T < timeToRed \wedge E) \vee (C \wedge \neg E)}^{\text{can stop before intersection after cycle of coasting}}) \\
\vee\ & \overbrace{C \wedge (F \vee \neg G \wedge (D_3 \vee D_2 \wedge H_1))}^{\text{can stop before intersection after cycle of coasting (but } D_1 \text{ doesn't have to hold)}} \\
\vee\ & \overbrace{(D_2 \wedge H_1 \vee D_3) \wedge T < timeToRed \wedge G}^{\text{after one cycle of coasting and then braking, car passes intersection}} \\
\vee\ & \overbrace{H_2}^{\text{can coast past intersection before signal switches}} \quad)))
\end{aligned}
$$

Likewise, the guard for braking has many repeated clauses. $P \equiv 2B + v^2/x \leq 0$ means that the car won't stop before the intersection. $Q \equiv \textit{timeToRed} > \frac{BT^2+2x}{2(BT-v)}$ means that even after one braking cycle, the car can cross the intersection by coasting. $R \equiv x(v^2 + 2Bx) > 0$ means that the car will stop before the intersection. $S \equiv B \cdot \textit{timeToRed} + (v^2 + 2Bx)^{1/2} > v$ means that the car will stop before the intersection before the signal turns red. $U \equiv 2(Tv + x) < BT^2$ means that the car will stop before time T. $V \equiv BT^2 = 2(Tv + x)$ means that if it were to brake, the car will come to a stop at time T. $W \equiv BT^2 < 2(Tv + x)$ means that if it were to brake, the car will come to a stop after time T. Because of structural similarities with G_1, we do not provide a full annotation.

$$
\begin{aligned}
G_2 \equiv\, & x < 0 \wedge (v(Tv + x) < 0 \wedge (P \wedge Q \vee R) \\
& \vee v(Tv + x) > 0 \wedge T < -2v^{-1}x \wedge (W \wedge S \vee P \wedge Q \wedge U \\
& \quad \vee R \vee V \wedge \textit{timeToRed} > T) \\
& \vee 0 = T + v^{-1}x \wedge (P \wedge Q \wedge U \vee R) \\
& \vee T \geq -2v^{-1}x \wedge (R \vee S \wedge x(v^2 + 2Bx) < 0)) \vee x \geq 0
\end{aligned}
$$

A.3.5 Curvebot

Model 15 Curvebot benchmark.

```
 assum | $(T > 0 \wedge \boxed{I}) \rightarrow \{$
  ctrl |   $((\boxed{G_1}\,;\ om := 1) \cup (\boxed{G_2}\,;\ om := 0) \cup (\boxed{G_3}\,;\ om := -1))\,;$
 plant |   $(t := 0\,;\ \{x' = v, y' = w, v' = om \cdot w, w' = -om \cdot v, t' = 1 \,\&\, t \leq T\})$
  safe | $\}^*]\neg(x = 0 \wedge y = 0)$
```

Where I, G_1, G_2 and G_3 are to be synthesized.

Curvebot models a Dubin's car that must avoid an obstacle at (0,0). The dynamics result in a solution that is not in the decidable fragment of arithmetic, so CESAR again uses Pegasus to find a controllable invariant.

Our implementation generates the optimal invariant, consisting of everywhere except the origin. The algebraic formulas presented are synthesized by CESAR automatically. The annotations describing their meaning are added manually for the convenience of the reader.

$$I \equiv \neg x = 0 \vee \neg y = 0$$

The guard for setting om to 1 is simply that the origin does not lie on the resulting circular path.

$$G_1 \equiv y = 0 \wedge \neg 2w + x = 0 \vee \neg y(2wx + x^2 + y((-2)v + y)) = 0$$

The guard for setting om to 0 is that the origin does not lie in the straight line segment of length

$T(v^2 + w^2)^{(}1/2)$ ahead.

$$
\begin{aligned}
G_2 \equiv\, & \overbrace{v < 0 \land (\neg y = 0 \land (0 = w \lor 0 = x)}^{\text{going in the -}x\text{ direction}\cdots} \\
& \quad \lor \overbrace{x > 0 \land (\neg w = 0 \land \neg y = wx/v \lor v(Tv + x) < 0) \lor x < 0)}^{\cdots\text{ is safe}} \\
& \lor \overbrace{v = 0 \land (\neg x = 0 \lor y > 0 \land (w \geq 0 \lor w(Tw + y) < 0)}^{\text{purely y motion}\cdots} \\
& \quad \lor \overbrace{y < 0 \land (w(Tw + y) < 0 \lor w \leq 0))}^{\cdots\text{ is safe}} \\
& \lor \overbrace{v > 0 \land (\neg y = 0 \land (0 = w \lor 0 = x)}^{\text{going in the }x\text{ direction}\cdots} \\
& \quad \lor \overbrace{x > 0 \lor (\neg w = 0 \land \neg y = wx/v \lor v(Tv + x) < 0) \land x < 0)}^{\cdots\text{ is safe}}
\end{aligned}
$$

Like G_1, the guard for setting om to -1 is that the origin does not lie on the resulting circular path.

$$G_3 \equiv y = 0 \land \neg 2w = x \lor \neg y(x^2 + y(2v + y)) = 2wxy$$

This solution is almost optimal. It only misses the cases for G_1 and G_2 where despite the obstacle lying on the circular path, T is small enough that there is time to switch paths before collision.

A.3.6 Corridor

Corridor, shown in Model 8, is an example of a system requiring unrolling.

A.3.7 Power Station

Model 16 Power Station benchmark.

$$
\begin{array}{r|rl}
\text{assum} & 1 & (T = 1 \land i \geq 0 \land \mathit{chargeRate} = 7000000 \land J = 100 \\
 & 2 & \land V = 2000 \land R = 5 \land \mathit{quota} = 3000 \land \boxed{I}) \rightarrow [\{ \\
\hline
\text{ctrl} & 3 & \quad ((\boxed{G_1}\,;\, i := 0\,;\, \mathit{slope} := \mathit{chargeRate}) \\
 & 4 & \quad \cup (\boxed{G_2}\,;\, i := J\,;\, \mathit{slope} := -i \cdot V))\,; \\
\hline
\text{plant} & 5 & \quad (t := 0\,;\, \{\mathit{produced}' = i \cdot V - i^2 \cdot R, \\
 & 6 & \mathit{stored}' = \mathit{slope}, t' = 1, \mathit{gt}' = -1 \;\&\; t \leq T\}) \\
\hline
\text{safe} & 7 & \}^*](\mathit{stored} > 0 \land (\mathit{gt} > 0 \lor \mathit{produced} > \mathit{quota}))
\end{array}
$$

Where I, G_1 and G_2 are to be synthesized.

Power station is an example of a system that needs two steps of unrolling in order to reach the optimal invariant. A power station capable of producing 7000 kW can choose between charging (Line 3) and distributing out stored power at current 100 A and voltage of 2000 V (Line 4). Its objective is to meet an energy quota of 3000 J, excluding loss due to resistance of 5Ω by the time that timer gt counts down to 0. The station must never reach a state where it has no stored power left. The system is modeled in Model 16. The zero-shot invariant corresponds to the case where the station has already met its quota and is now charging. Discharge is not included in the invariant because regardless of how high stored energy is, in the unbounded time of the zero-shot invariant, a station that has chosen to distribute power will eventually run out of it, thus violating the condition $stored > 0$. The one shot invariant catches the case where the station first chooses to discharge till it meets its quota, and then flips to charging for infinite time. The two shot invariant, which is finally optimal, catches the case where the station first chooses to charge till it has enough energy stored to meet its quota, then discharges, and finally switches back to charging mode for infinite time.

CESAR finds the solution below. The algebraic formulas presented are synthesized by CESAR automatically. The annotations describing their meaning are added manually for the convenience of the reader.

$$
\begin{aligned}
I \equiv \mathit{produced} \leq 3000 \land (&\overbrace{3000(-51 + 50\mathit{gt}) + \mathit{produced} > 0}^{\text{enough time to charge} \cdots} \\
&\overbrace{\land 7000000\mathit{gt} + 48\mathit{produced} + \mathit{stored} > 7344000 \land 437500\mathit{gt} + 3\mathit{produced} < 459000}^{\cdots \text{ then distribute}} \\
&\lor \overbrace{150000\mathit{gt} + \mathit{produced} > 3000 \land 4\mathit{produced} + 3\mathit{stored} > 612000}^{\text{enough stored power to distribute and meet quota}}) \\
&\lor \overbrace{stored > 0 \land (437500\mathit{gt} + 3\mathit{produced} \geq 459000 \lor \mathit{produced} > 3000)}^{\text{without drawing stored energy, enough time to charge then distribute}}
\end{aligned}
$$

The guard for charging checks that the choice to charge for a cycle still leaves enough time for distributing the power.

$$
\begin{aligned}
G_1 \equiv &\overbrace{\mathit{gt} > 1 \land 3000(\mathit{gt} \cdot 50 - 51) + \mathit{produced} > 0}^{\text{enough time to charge then distribute}} \\
&\lor \overbrace{\mathit{gt} \cdot 35 < 36 \land \mathit{produced} > 3000}^{\text{quota already met}}
\end{aligned}
$$

The guard for distributing basically checks that the choice to discharge for a cycle still leaves enough stored energy.

$$
\begin{aligned}
G_2 \equiv &\overbrace{\mathit{produced} \leq 3000 \land 3000(50\mathit{gt} - 51) + \mathit{produced} \leq 0}^{\text{must distribute to stay safe}} \\
&\lor \overbrace{3000(50\mathit{gt} - 51) + \mathit{produced} > 0 \land stored > 200000 \land \mathit{gt} > 1}^{\text{enough stored energy to distribute for a cycle, can produce enough afterwards}} \\
&\lor \overbrace{\mathit{gt} \leq 1 \land stored > 200000 \land \mathit{produced} > 3000}^{\text{enough stored energy for a cycle, quota satisfied}}
\end{aligned}
$$

A.3.8 Coolant

The coolant benchmark is an example of a system that requires unrolling. A coolant system draws water at either a rate of F or 0 m^3/s from a reservoir. It runs the water through a heat exchanger,

Model 17 Coolant benchmark.

```
assum  $(T > 0 \land F > 0 \land \mathit{minAbsorbed} > 0 \land \mathit{maxDischarge} > 0$
       $\land\ \mathit{tempDiff} > 0 \land c > 0 \land \boxed{I}) \rightarrow \{$
ctrl     $((\boxed{G_1}\,;\ f := 0)$
         $\cup\ (\boxed{G_2}\,;\ f := F))\,;$
plant    $(t := 0\,;\ \{\mathit{absorbed}' = f \cdot c \cdot \mathit{tempDiff},$
       $\mathit{discharged}' = f, t' = 1, \mathit{gt}' = -1\ \&\ t \leq T\})$
safe   $\}^*](\mathit{discharged} < \mathit{maxDischarge} \land (\mathit{gt} > 0 \lor \mathit{absorbed} \geq \mathit{minAbsorbed}))$
```

Where I, G_1 and G_2 are to be synthesized.

drawing heat proportional to the specific heat of water c and the temperature difference between the water and the system. It then discharges the water into a river at the rate that it draws water. The cooling system must absorb at least $minAbsorbed$ heat by the time that timer gt counts down to 0. But it must also not discharge more that $maxDischarged$ water into the river to avoid environmental damage. The benchmark is modeled in Model 17.

CESAR finds the solution below. The algebraic formulas presented are synthesized by CESAR automatically. The annotations describing their meaning are added manually for the convenience of the reader.

$$\begin{aligned}
I \equiv\ & \overbrace{\mathit{absorbed} \geq \mathit{minAbsorbed} \land \mathit{discharged} < \mathit{maxDischarge}}^{\text{already absorbed enough heat}} \\
& \lor \overbrace{\mathit{gt} > 0 \land (\mathit{discharged} + F \cdot T < \mathit{maxDischarge}}^{\text{drawing water one cycle will not exceed discharge limit}} \\
& \quad \overbrace{\land \mathit{maxDischarge} \leq \mathit{discharged} + F \cdot (\mathit{gt} + T)}^{\text{but drawing water till timer } gt \text{ runs out will}} \\
& \quad \overbrace{\land c > (\mathit{absorbed} - \mathit{minAbsorbed}) \cdot \mathit{tempDiff}}^{\text{drawing as much water as discharge limit allows} \cdots} \\
& \qquad \overbrace{\cdot (\mathit{discharged} - \mathit{maxDischarge} + F \cdot T)^{-1}}^{\cdots \text{ absorbs enough heat}} \\
& \lor \overbrace{\mathit{discharged} + F \cdot (\mathit{gt} + T) < \mathit{maxDischarge}}^{\text{drawing water now} \cdots} \\
& \quad \overbrace{\land \mathit{gt}^{-1} \cdot (\mathit{absorbed} - \mathit{minAbsorbed} + c \cdot F \cdot \mathit{gt} \cdot \mathit{tempDiff}) \geq 0)}^{\cdots \text{ till the timer runs out is safe}}
\end{aligned}$$

The guard for drawing no water just needs to check that it still has enough time to absorb enough heat.

$$\begin{aligned}
G_1 \equiv\ & \overbracket{\mathit{absorbed} \geq \mathit{minAbsorbed}}^{\text{already absorbed enough heat}}\\
\lor\ & \overbracket{\mathit{gt} > T \land ((F = (\mathit{maxDischarge} - \mathit{discharged}) \cdot \mathit{gt}^{-1}}^{\text{drawing water will } \cdots}\\
& \overbracket{\lor \mathit{discharged} + T \cdot F \geq \mathit{maxDischarge}}^{\cdots \text{ exceed the } \cdots}\\
& \overbracket{\lor \mathit{gt} \cdot (\mathit{discharged} + \mathit{gt} \cdot F - \mathit{maxDischarge}) > 0)}^{\cdots \text{ discharge limit}}\\
\lor\ & \overbracket{\mathit{gt} \cdot (\mathit{discharged} + \mathit{gt} \cdot F - \mathit{maxDischarge}) < 0}^{\text{despite deferring one cycle } \cdots}\\
& \overbracket{\land (\mathit{absorbed} - \mathit{minAbsorbed}) \cdot (\mathit{gt} - T)^{-1} + F \cdot (c \cdot \mathit{tempDiff}) \geq 0)}^{\cdots \text{ can still absorb enough heat}}
\end{aligned}$$

The guard for drawing water checks that drawing water for one cycle will not exceed the discharge limit.

$$G_2 \equiv \mathit{discharged} + F \cdot T < \mathit{maxDischarge}$$

A.4 Simplification Impact Example

Reduction of even a loop free dGL formula, if done naïvely, can produce very large expressions. Not only are large expression hard to reason about, but they also slow down quantifier elimination. The effect on QE is so stark that the benchmark `Curvebot`, which completes in 26s with our simplification stack, times out after 20 minutes without simplification. Our simplification uses Mathematica's `Simplify` function. Additionally, to catch common patterns that CESAR produces but Mathematica does not handle, we implemented a custom simplifier with `Egg` and `z3`.

The impact of simplification can be dramatic. For example, the I_0 invariant produced by CESAR for `Curvebot` *without* simplification is the following expression:

$$\begin{aligned}
&((v \neq wx/y \land y \neq 0 \lor y = 0 \land (w = 0 \land (v \geq 0 \land x > 0 \lor v \leq 0 \land x < 0)\\
&\lor w \neq 0 \land x \neq 0) \lor w \geq 0 \land y > 0 \lor w \leq 0 \land y < 0)\\
&\lor x = 0 \land 2v \neq y \land y \neq 0 \lor x \neq 0 \land (2w + x \neq 0 \lor y \neq 0)\\
&\land (y = 0 \lor 2vx \cdot y \neq x \cdot (2wx + x^2 + y^2))) \lor (2w \neq x \lor y \neq 0)\\
&\land x \neq 0 \land (y = 0 \lor x^3 + x \cdot y \cdot (2v + y) \neq 2wx^2)\\
&\lor x = 0 \land 2v + y \neq 0 \land y \neq 0
\end{aligned}$$

Running this formula through our simplifier produces the expression $y \neq 0 \lor (y = 0 \land 0 \neq x)$. Thus, the expression is actually just $\neg(x = 0 \land y = 0)$.

Besides being much easier to read, simpler expressions are also much faster to eliminate quantifiers from. Since doubly exponential quantifier elimination is the bottleneck in scaling to increasingly complex problems, simplification is crucial to the scalability of CESAR. In the case of our example above, without simplification, `Curvebot` gets stuck on the following QE problem.

$$\begin{aligned}
&\forall v_1, w_1, x_1, y_1(((((v^2 + w^2 = v_1^2 + w_1^2 \land vw + vx - wy - xy\\
&= v_1w_1 + v_1x_1 - w_1y_1 - x_1y_1) \land w^2/2 + vy - y^2/2 = w_1^2/2 + v_1y_1 - y_1^2/2)\\
&\land v - y = v_1 - y_1) \land w^2/2 + wx + x^2/2 = w_1^2/2 + w_1x_1 + x_1^2/2)\\
&\land w + x = w_1 + x_1 \to (((v_1 \neq w_1x_1/y_1 \land y_1 \neq 0\\
&\lor y_1 = 0 \land (w_1 = 0 \land (v_1 \geq 0 \land x_1 > 0 \lor v_1 \leq 0 \land x_1 < 0)\\
&\lor w_1 \neq 0 \land x_1 \neq 0) \lor w_1 \geq 0 \land y_1 > 0 \lor w_1 \leq 0 \land y_1 < 0)\\
&\lor x_1 = 0 \land 2v_1 \neq y_1 \land y_1 \neq 0 \lor x_1 \neq 0 \land (2w_1 + x_1 \neq 0\\
&\lor y_1 \neq 0) \land (y_1 = 0 \lor 2v_1x_1 \cdot y_1 \neq x_1 \cdot (2w_1x_1 + x_1^2 + y_1^2)))\\
&\lor (2w_1 \neq x_1 \lor y_1 \neq 0) \land x_1 \neq 0 \land (y_1 = 0\\
&\lor x_1^3 + x_1 \cdot y_1 \cdot (2v_1 + y_1) \neq 2w_1x_1^2)\\
&\lor x_1 = 0 \land 2v_1 + y_1 \neq 0 \land y_1 \neq 0) \land \neg(x_1 = 0 \land y_1 = 0))
\end{aligned}$$

Table A.2: Simplification impact on solution size.

Benchmark	**% Solution Size Reduction**
ETCS Train	62%
Sled	41%
Intersection	38%
Curvebot	∞
Parachute	5%
Coolant	92%
Corridor	85%
Power Station	55%

But with simplification applied regularly, instead the corresponding call to QE is

$$\begin{aligned}
&\forall v_1, w_1, x_1, y_1(((((v^2 + w^2 = v_1^2 + w_1^2 \land vw + vx - wy - xy \\
&= v_1 w_1 + v_1 x_1 - w_1 y_1 - x_1 y_1) \land w^2/2 + vy - y^2/2 = w_1^2/2 + v_1 y_1 - y_1^2/2) \\
&\land v - y = v_1 - y_1) \land w^2/2 + wx + x^2/2 = w_1^2/2 + w_1 x_1 + x_1^2/2) \\
&\land w + x = w_1 + x_1 \rightarrow \neg(x_1 = 0 \land y_1 = 0))
\end{aligned}$$

which terminates in 70 milliseconds instead of timing out.

Finally, even on benchmark examples that would terminate without the use of *eager simplification*, enabling this optimization often results in shorter solutions. Table A.2 shows the percentage reduction in size of solutions due to simplification, where size is measured in terms of number of characters. A 0% reduction means no change and a 50% reduction means cutting formula-size in half.

Appendix B

Generalized Hybrid Game Synthesis (Appendix)

B.1 Naïve Ordering Counterexample

For game $a{:}\alpha$, let S' set the winning region for the overall game to be empty ($S'(a) = \bot$), meaning that Angel is never allowed to play at all. Now the winning subregions for later subgames are irrelevant because Angel cannot reach them, but S' nevertheless sets them to the theoretical optimal (for every subgame $b \neq a$, $S(b) \mapsto \langle a{:}\alpha_{b:}\rangle\phi$). For *any* other inductive subvalue map S, a good ordering should indicate that $S \sqsupseteq S'$ since S cannot possibly be less permissive, but the proposed naïve ordering does not. When S sets the winning subregion for any of the later subgames $b \neq a$ to anything less than the optimal solution, the check $\models S'(b) \to S(b)$ fails.

For a concrete visualization, consider the two inductive subvalue maps shown in Fig. B.1: S on the left and S' on the right. Because S' immediately blocks off *all* strategies at the root, a good ordering should indicate $S \sqsupseteq S'$, but the proposed naïve ordering does not because at the subgame α, S has a strictly stronger formula ($y = 1 \to \top$ is false). The problem in the naïve, point wise definition (Def. B.1) is it does not recognize that the $\top$ at subgame α is vacuous: S' permits no traces to reach subgame α in the first place, so the permissiveness of $\top$ is unimpressive.

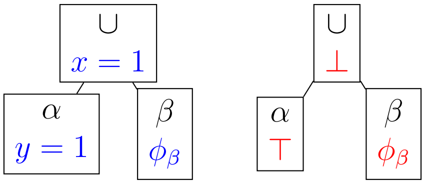


Figure B.1: Inductive subvalue maps S and S'. Locally at subgame α, S' seems more permissive, but global analysis reveals S is more permissive.

B.2 Inductive Subvalue Map Loop Condition Discussion

We discuss the details of the definition of an inductive subvalue map (Def. 6.8) for the case $a{:}(g{:}\gamma)^* \models S$, and correspondingly the synthesis framework $\lozenge\mathsf{map}$ (Algorithm 2) for the case of $\lozenge\mathsf{map}(a{:}(g{:}\gamma)^*, \phi)$.

B.2.1 Counterexample for Projection Condition Weakening

Def. 6.8 and Algorithm 2 for the case $(g{:}\gamma)^*$ use the check $\models S(a) \rightarrow \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\phi$, i.e., winning subregion $S(a)$ should ensure that there is a *finite* strategy to exit while staying within the control envelope. The subvalue map should ensure that Angel does not get stuck in a state where her only option is playing the loop forever. It might seem that the more liberal, simpler condition $\models S(a) \rightarrow \langle\alpha\rangle\phi$ also ensures Angel has a finite exit strategy, but this condition is subtly incorrect. Angel can get stuck in a state where there is theoretically a way to exit the loop but this requires going outside the control envelope. Eq. (B.1) shows a counterexample.

$$\begin{gathered} \langle a{:}(b{:}(c{:}x := x + 1 \cup d{:}x := x - 1))*\rangle x \geq 0 \\ S := \{a \mapsto \top, b \mapsto \top, c \mapsto \bot, d \mapsto \top\} \end{gathered} \tag{B.1}$$

For this game, if Def. 6.8 were modified such that $\langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\phi$ is replaced by $\langle\alpha\rangle\phi$, the subvalue map S would be inductive. However, it would also let Angel get stuck iterating the loop forever. Consider starting state $\{x \mapsto -1\}$. This state belongs to the winning subregion $S(a)$ of the overall game, so Angel should be able to win. And indeed she can win the game by running one iteration of the loop where she increments x. The new state would be $\{x \mapsto 0\}$, so Angel would exit the loop and win since winning condition $x \geq 0$ holds. However, if Angel is forced to actually respect the subvalue map (modeled by projection of S onto the game as shown in Eq. (B.2)), she cannot win.

$$\langle ?\top\,;\,(?\top\,;\,((?\bot\,;\,x := x + 1) \cup (?\top\,;\,x := x - 1)))^*\,;\,?x \geq 0\rangle x \geq 0 \tag{B.2}$$

Starting at $\{x \mapsto -1\}$, Angel has to repeat the loop since she has not reached the winning region $x \geq 0$. But inside the loop, the subvalue map says she can never choose to increment x (subgame c). So she is forced to decrement x, reaching new state $\{x \mapsto -2\}$. In principle, if she were allowed to ignore the subvalue map, she could still exit with finite iterations by incrementing x twice. But with subgame c blocked off, she is only ever allowed to decrement x and is stuck in the loop forever. To prevent such subvalue maps from counting as inductive, the check $\models S(a) \rightarrow \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\phi$ is necessary.

B.2.2 Counterexample for Recursive Condition Weakening

Def. 6.8 and Algorithm 2 for the case $(g{:}\gamma)^*$ also use the check $g{:}\gamma \models S$, i.e., the subvalue map for the loop body ensures that after playing the loop body, either there is still a finite exit strategy or Angel can already exit the loop. Eq. (B.3) shows a counterexample where using $g{:}\gamma \models S$ instead is not sound, since the inner subvalue map permit reaching a state without a finite exit strategy.

$$\begin{gathered} \langle a{:}(b{:}(c{:}x := x + 1 \cup d{:}x := x - 1))*\rangle x \geq 0 \\ S := \{a \mapsto x \geq -1, b \mapsto x \geq -1, c \mapsto x \geq -1, d \mapsto \top\} \end{gathered} \tag{B.3}$$

Starting at $\{x \mapsto -1\}$, it is possible for Angel to choose subgame d and decrement x. The new state is $\{x \mapsto -2\}$, from which there is now no finite exit strategy because to increment x, per the guard of subgame c, x must be at least -1. Angel is now stuck decrementing x forever.

B.3 Example of Subvalue Map Shortfall for Runtime Monitoring

We show how for a (non-inductive) subvalue map, the property $\models S(a) \rightarrow \langle a{:}\alpha \lightning S\phi \rangle \phi$ does not always hold. Consider the subvalue map of Fig. 6.1, but with ϕ_6 set to the empty subregion $\bot$ instead of $x > 0 \wedge (v \geq 0 \vee a > 0)$. As before, let the overall game be $a{:}\alpha$, and subgame labels be the numbers indexing the winning subregions. The new map is still a subvalue map since $\models \bot \rightarrow \langle a{:}\alpha_{5:} \rangle x > 0$. The untrusted controller monitoring correctness property is Eq. (6.3) with ϕ_6 replaced by $\bot$. The result has shape

$$x > 0 \wedge (v \geq 0 \vee a > 0) \rightarrow \big\langle \big(\big(?(\bot \vee (x > 0 \wedge v \geq 1)) \,;\, \cdots \cap \cdots \big) \,;\, \cdots \big)^{\times} \big\rangle\, x > 0.$$

This formula is not valid. Consider the starting state $\{x \mapsto 1, v \mapsto 0\}$. Demon first chooses to run the loop, then Angel must pass the test $x > 0 \wedge v \geq 1$, which fails. The *inductive* subvalue map prevents this situation by requiring that ϕ_8 implies $\phi_6 \vee \phi_5$, thus disqualifying this counterexample where Angel gets stuck.

B.4 Benchmarks

This appendix discusses the benchmarks, provides their full listings, and discusses the rewrite heuristics used to solve them.

B.4.1 New Examples Discussion

We discuss the examples representing diverse control challenges introduced in this chapter to demonstrate the flexibility of our dGL control envelope synthesis approach.

Event-triggered ETCS (Model 9) models a train from a case study modeling the European Train Control System (ETCS) [126], but with a key modification: the train is modeled event triggered instead of time triggered. The control envelope computed by our implementation finds the winning subregion for the overall loop to be $p < e \wedge (2Ae + v^2 < 2Ap \vee v \leq 0) \wedge (A \leq 0 \vee v > 0) \vee p < e \wedge (2(-B)e + v^2 < 2(-B)p \vee v \leq 0) \wedge (-B \leq 0 \vee v > 0)$. After simplification, the key term ends up being $e - p + v^2/2B > 0$, i.e., there is still enough time for the train to come to a stop before the end of motion authority e if it starts braking now. Hence, the control envelope has a strategy to keep the train safe whenever it is possible for the train to be safe by braking.

Surgical Robot (Model 19) is based on a case study from the literature [83] modeling force feedback in surgical robots. The control envelope computed by our implementation finds the winning subregion for the overall loop to be $(q_x - p_x)n_x + (q_y - p_y)n_y >= 0$, i.e., so long as the robot is not already outside the fixture, the subvalue map has a strategy for safe control.

Infinite Track (Model 20) is an example of infinite horizon switching. A vehicle in a looping track can choose to move in any of the four coordinate directions, but must remain moving at all times. To remain safe for arbitrarily long time, this vehicle must keep switching directions at the right moments, and make *infinitely* many switches. Our approach synthesizes a control envelope that indicates when it is safe to switch to any given direction. CESAR does not solve this problem as it does not handle infinite control strategies. The control envelope computed by our implementation finds the winning subregion for the overall loop to be $2R > x \wedge x > -2R \wedge 2R > y \wedge y > -2R \wedge (x > R \vee x < -R \vee y > R \vee y < -R)$, i.e., the subvalue map has a safe strategy for the vehicle at every point in the track.

Reach-avoid Robot (Model 21) demonstrates an example of envelope synthesis for reach-avoid problems, where an agent must *reach* an objective while *avoiding* unsafe situations. Such problems can model the simultaneous requirement of safety and liveness. In this example, the robot is safe in the square $[-2R, 2R] \times [-2R, 2R]$ (enforced in domain constraint of Line 3) and must reach the target region $[R, 2R] \times [R, 2R]$ (on Line 4). The robot can either travel upward ($v_x = 0, v_y = V$) or leftward ($v_x = -V, v_y = 0$). The synthesized subvalue map identifies that it is safely possible to reach the target region starting in $[R, 2R] \times [-2R, 2R]$.

Highway Driving (Model 22) is a time-triggered problem based on the core control challenge from a case study on highway driving [94]. Two cars, are driving on a highway. The lead car, with velocity v_l and acceleration a_l is driven by some external agent with the physical limitation that the car's accelerates is bounded above by A and its braking is bounded below by $-B$ (Line 2). The following car, with velocity v_f and acceleration a_f, is driven by a controller that must follow the lead car while avoiding collisions (Line 3). The controllers revise decisions in a time triggered fashion, with maximum latency T. Regardless of the lead car's behavior, the following car must always be able to stop in time to avoid collision (Line 5). This problem does not fit CESAR's template, which does not support an adversarial agent like the lead car. In the computed control envelope, the winning region for the overall game is $p_f < p_l \wedge v_f \leq v_l$, i.e. the subvalue map has a strategy to avoid the collisions whenever that velocity of the controlled car is less than that of the lead car and hasn't already collided with it.

The descriptions of the CESAR benchmarks can be found in Appendix A.3 [70]. The envelopes that we synthesize for these benchmarks are the same as those computed by CESAR.

B.4.2 Quadcopter

The procedurally generated Quadcopter benchmark suite is based on the following template with the addition of randomly generated values for velocity and obstacles.

The Quadcopter chooses whether to go up or down and runs continuous dynamics. The catch is that the quadcopter has a slow processor and cannot make a new choice in less than 1 second (Line 4). However, it can choose to run the dynamics for any time of its choice that is greater than 1 second (Line 3 has an Angel ODE). It can repeat this process of choosing and then sticking with a choice any number of times (Angel loop in Line 4). While it flies, it must avoid obstacles at all times. It must also not crash into the floor at $y > 0$. It must eventually reach the target region $x > 20$. CESAR cannot solve this problem since it lies outside the template. NYCS solves all instances.

An example of the formula that can be generated for `<avoid obstacles>` obstacles is

Model 18 Quadcopter Suite Template

```
assume     $x > 0 \land V = \texttt{<velocity>} \rightarrow \Big\langle$
direction  $\Big( (v_y := 1) \cup (v_y := -1) \Big)$ ;
plant        $t := 0; \{x' = V, y' = v_y, t' = 1 \;\&\; \texttt{<avoid obstacles>} \land y \geq 0\}$ ;
safe         $?t \geq 1 \Big)^{*} \Big\rangle\, x > 20$
```

$(x < 2 \lor 2 < x \lor y < 0 \lor 3 < y) \land (x < 5 \lor 6 < x \lor y < 0 \lor 3 < y) \land (x < 5 \lor 8 < x \lor y < 9 \lor 9 < y) \land (x < 8 \lor 10 < x \lor y < 7 \lor 7 < y)$. This formula has four obstacles. The first one is a line segment from $(2, 0)$ to $(2, 3)$. The second is a rectangle from $(5, 0)$ to $(6, 3)$. The third is a rectangle from $(5, 9)$ to $(8, 9)$. The fourth is a rectangle from $(8, 7)$ to $(10, 7)$. An example of a generated number that fills `<velocity>` is 1. The procedural generation parameters allow up to 5 obstacles with widths and heights of at most 3 distributed in the region $[0, 10] \times [0, 10]$. Velocity is an integer between 1 and 5.

As expected, the implementation solves problems with fewer obstacles quickly, with the fastest solutions for one obstacle completing in a couple of seconds. However, the time taken increases with the number of obstacles, and more importantly, when obstacles are placed in such a way that they interact to make the solution space more complex. Table B.1 summarizes the generated problems and their solving outcomes.

Table B.1: Quadcopter benchmark suite generated parameters and solving outcomes. In the result column, ✓ indicates that a nonempty control envelope was computed, while ✗ indicates that algorithm timed out after 20 minutes, failing to compute an envelope.

No.	Velocity	Obstacles	Result
1	5	$x < 9 \lor 11 < x \lor y < 1 \lor 3 < y$	✓
2	4	$x < 9 \lor 9 < x \lor y < 3 \lor 3 < y$	✓
3	1	$(x < 2 \lor 2 < x \lor y < 0 \lor 3 < y) \land (x < 5 \lor 6 < x \lor y < 0 \lor 3 < y) \land (x < 5 \lor 8 < x \lor y < 9 \lor 9 < y) \land (x < 8 \lor 10 < x \lor y < 7 \lor 7 < y)$	✗
4	1	$x < 3 \lor 6 < x \lor y < 9 \lor 10 < y$	✓
5	2	$x < 0 \lor 1 < x \lor y < 2 \lor 4 < y$	✓
6	2	$(x < 7 \lor 10 < x \lor y < 8 \lor 8 < y) \land (x < 3 \lor 5 < x \lor y < 8 \lor 9 < y)$	✓
7	4	$(x < 7 \lor 8 < x \lor y < 1 \lor 3 < y) \land (x < 5 \lor 5 < x \lor y < 6 \lor 8 < y)$	✗
8	1	$(x < 3 \lor 3 < x \lor y < 8 \lor 11 < y) \land (x < 0 \lor 0 < x \lor y < 1 \lor 1 < y)$	✓
9	5	$(x < 1 \lor 3 < x \lor y < 7 \lor 9 < y) \land (x < 4 \lor 4 < x \lor y < 0 \lor 3 < y)$	✗
10	1	$\top$	✓
11	2	$x < 3 \lor 4 < x \lor y < 6 \lor 8 < y$	✓

No.	**Velocity**	**Obstacles**	**Result**
12	2	$(x < 10 \lor 13 < x \lor y < 10 \lor 11 < y) \land (x < 5 \lor 5 < x \lor y < 7 \lor 8 < y) \land (x < 7 \lor 10 < x \lor y < 10 \lor 12 < y) \land (x < 3 \lor 4 < x \lor y < 7 \lor 8 < y) \land (x < 2 \lor 2 < x \lor y < 2 \lor 4 < y)$	✗
13	3	$(x < 2 \lor 2 < x \lor y < 7 \lor 7 < y) \land (x < 8 \lor 11 < x \lor y < 7 \lor 8 < y) \land (x < 1 \lor 2 < x \lor y < 9 \lor 12 < y)$	✗
14	4	$x < 9 \lor 11 < x \lor y < 4 \lor 7 < y \land x < 3 \lor 3 < x \lor y < 8 \lor 8 < y \land x < 9 \lor 11 < x \lor y < 8 \lor 11 < y \land x < 8 \lor 11 < x \lor y < 4 \lor 7 < y \land x < 7 \lor 8 < x \lor y < 9 \lor 10 < y$	✓
15	3	$(x < 1 \lor 4 < x \lor y < 9 \lor 10 < y) \land (x < 7 \lor 10 < x \lor y < 0 \lor 3 < y) \land (x < 8 \lor 10 < x \lor y < 8 \lor 11 < y) \land (x < 3 \lor 6 < x \lor y < 3 \lor 4 < y)$	✗
16	4	$(x < 1 \lor 1 < x \lor y < 8 \lor 11 < y) \land (x < 8 \lor 10 < x \lor y < 6 \lor 9 < y) \land (x < 5 \lor 8 < x \lor y < 8 \lor 11 < y) \land (x < 6 \lor 6 < x \lor y < 3 \lor 4 < y) \land (x < 10 \lor 13 < x \lor y < 10 \lor 13 < y)$	✗
17	4	$\top$	✓
18	3	$(x < 5 \lor 5 < x \lor y < 4 \lor 4 < y) \land (x < 1 \lor 2 < x \lor y < 5 \lor 8 < y) \land (x < 8 \lor 11 < x \lor y < 9 \lor 12 < y) \land (x < 2 \lor 2 < x \lor y < 1 \lor 3 < y) \land (x < 6 \lor 7 < x \lor y < 1 \lor 2 < y)$	✗
19	3	$(x < 4 \lor 6 < x \lor y < 0 \lor 0 < y) \land (x < 8 \lor 8 < x \lor y < 9 \lor 10 < y) \land (x < 2 \lor 2 < x \lor y < 2 \lor 2 < y) \land (x < 3 \lor 3 < x \lor y < 0 \lor 2 < y) \land (x < 6 \lor 9 < x \lor y < 8 \lor 11 < y)$	✓
20	2	$(x < 2 \lor 3 < x \lor y < 9 \lor 11 < y) \land (x < 10 \lor 13 < x \lor y < 4 \lor 5 < y) \land (x < 9 \lor 11 < x \lor y < 8 \lor 11 < y)$	✓
21	3	$(x < 4 \lor 4 < x \lor y < 5 \lor 5 < y) \land (x < 5 \lor 6 < x \lor y < 7 \lor 9 < y) \land (x < 10 \lor 13 < x \lor y < 3 \lor 4 < y)$	✓
22	4	$(x < 6 \lor 8 < x \lor y < 3 \lor 4 < y)$	✓
23	3	$(x < 5 \lor 8 < x \lor y < 8 \lor 9 < y) \land (x < 0 \lor 3 < x \lor y < 3 \lor 6 < y) \land (x < 6 \lor 8 < x \lor y < 6 \lor 9 < y)$	✓
24	4	$(x < 3 \lor 3 < x \lor y < 7 \lor 10 < y) \land (x < 9 \lor 9 < x \lor y < 1 \lor 4 < y) \land (x < 3 \lor 5 < x \lor y < 10 \lor 12 < y) \land (x < 4 \lor 4 < x \lor y < 6 \lor 9 < y)$	✓
25	1	$(x < 6 \lor 6 < x \lor y < 9 \lor 10 < y)$	✓

As a demonstrative example, we discuss the control envelope computed for problem 23 in the benchmark suite. Fig. B.2 shows a Mathematica plot of the winning subregion for the overall loop of this example. The problem has three obstacles, two of which overlap. It is naturally unsafe to start at an obstacle. But it is further unsafe to start right before an obstacle because collision with the obstacle becomes unavoidable. Notice a triangle of unsafe space right after the lowest obstacle. This unsafe triangle is an example of how obstacles can interact to create unsafe regions. In this triangle, it is neither safe to go upwards because of collision with the upper obstacle, nor is it safe to go downwards because of collision with the floor. This envelope is computed in 68 seconds.

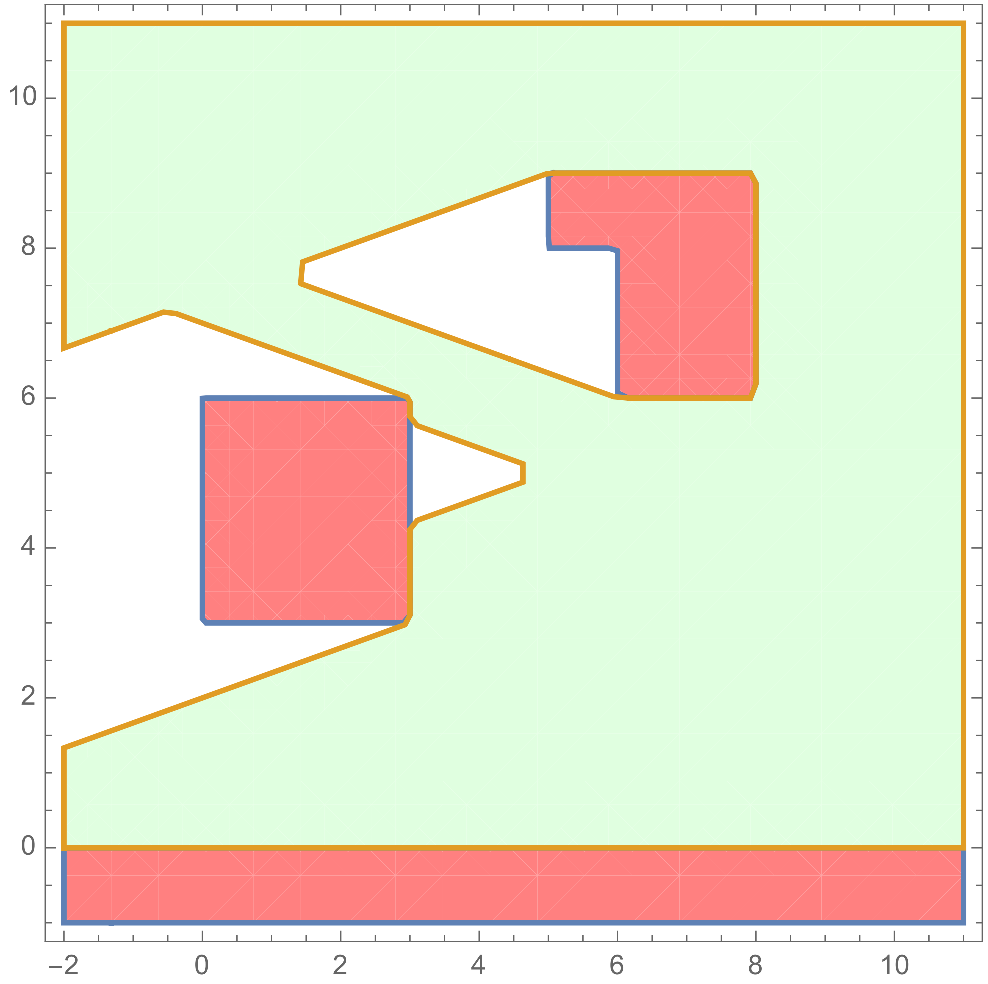


Figure B.2: Computed control envelope for quadcopter problem 23. Red regions are unsafe (obstacles or floor). Green regions are safe starting points per the computed control envelope. The x and y coordinates represent the quadcopter's position.

B.4.3 Full Benchmark Listings

We list the dGL formulas for all the new examples in Fig. 6.3a.

Model 19 Surgical Robot example based on [83]

```
assume      | $T > 0 \land K > 0 \land n_x^2 + n_y^2 = 1 \rightarrow \big\langle$
input       | $\big(f_{xp} := \otimes\,;\ f_{yp} := \otimes\,;$
damping     | $g := *; ?g \geq 0\,;$
plant       | $t := 0; \big\{q_x' = K f_x g, q_y' = K f_y g, f_x' = f_{xp}, f_y' = f_{yp}, t' = 1 \,\&\, t \leq T\big\}^d$
safe        | $\big)^\times\big\rangle\, ((q_x - p_x)n_x + (q_y - p_y)n_y \geq 0)$
```

Model 20 Infinite Track

```
assume      | $T > 0 \land V > 0 \land R > 0 \land 2VT < R \rightarrow \big\langle$
east/west   | $\Big(\big((v_x := V; v_y := 0) \cup (v_x := -V; v_y := 0) \cup$
north/south |     $(v_x := 0; v_y := -V) \cup (v_x := 0; v_y := V)\big)\,;$
plant       |     $t := 0; \{x' = v_x, y' = v_y, t' = 1 \,\&\, t \leq T\}^d\Big)^\times$
safe        | $\big\rangle\, (2R > x \land x > -2R \land 2R > y \land y > -2R \land$
            |     $(x > R \lor x < -R \lor y > R \lor y < -R))$
```

Fig. 6.3b compares the performance of our approach to CESAR on the CESAR benchmark suite. The performance is similar at most benchmarks with only a few seconds of overhead. An exception is Intersection, where CESAR is significantly more expensive. The observed differences are likely a consequence of different simplification heuristics rather than any fundamental algorithmic reasons. Simplification is itself an expensive operation, but if done at the right juncture, can make the even more expensive quantifier eliminations that occur later in symbolic execution cheaper. Thus simplification has a large influence on performance. Further code tuning and heuristics optimizations can likely reduce the overheads. LLM assistance was used to generate some of the boilerplate code of the implementation.

B.4.4 Rewrite Heuristics

Our implementation uses rewriting heuristics to simplify problems into shapes where they can either be symbolically executed or solved using other invariant generation heuristics. Table B.2 lists the rewrite heuristics and which evaluation problem each is used in.

We discuss how the rewrite heuristics work.

Model 21 Reach-avoid Robot

assume | $V > 0 \land R > 0 \rightarrow \Big\langle$
direction | $\Big(\big((v_x := -V; v_y := 0) \cup (v_x := 0; v_y := V)\big)\,;$
plant | $\{x' = v_x, y' = v_y \& \,(2R \geq x \land x \geq -2R \land 2R \geq y \land y \geq -2R)\}\Big)^* \Big\rangle$
safe | $R \leq x \land x \leq 2R \land R \leq y \land y \leq 2R$

Model 22 Highway Driving

assume | $A > 0 \land B > 0 \land T > 0 \land v_f > 0 \land v_l > 0 \rightarrow \Big\langle$
lead car | $\Big(a_l := \otimes; !{-B} < a_l \land a_l < A\,;$
controlled car | $a_f := *; ? - B < a_f \land a_f < A\,;$
plant | $t := 0; \big\{p_f' = v_f, v_f' = a_f, p_l' = v_l, v_l' = a_l, t' = 1 \;\&\; t \leq T\big\}^d \Big)^\times \Big\rangle$
safe | $p_f < p_l$

- *Extremal assignment rewrite.* This rewrite replaces a guarded assignment (e.g., $x := \otimes; ?x > 0$) with an assignment to an extremal value identified from the guard (e.g., $x := 0$). The rewritten program is now much easier to reason about, since a free assignment otherwise requiring a quantifier to reason about has been replaced by a constant expression.
- *Adversarial one-shot rewrite.* This rewrite is similar in spirit to the *one-shot unrolling* of CESAR [70]. It takes an Angel loop ending with an Angel ODE (e.g., the inner loop of Model 9) and replaces it with a single iteration of the loop but with the ODE changed to a Demon ODE guarded at the end by an Angel test for the domain constraint of the original ODE. The intuition is that the rewritten program emulates running any number of iterations of the original loop with the ODE run for any amount of time, so a precondition for the harder rewritten program is likely to also be a precondition for the original program.
- *Loop unrolling.* This heuristic unrolls loops a fixed number of times. For example, $\langle\alpha^*\rangle\phi$ is rewritten as $\langle\alpha\rangle\phi$ or $\langle\alpha\rangle\langle\alpha\rangle\phi$ or $\langle\alpha\rangle\langle\alpha\rangle\langle\alpha\rangle\phi$ and so on.

Table B.2: Rewrite heuristics used in the evaluation problems.

Heuristic	Problems
Extremal assignment rewrite	Surgical robot, Highway driving
Adversarial one-shot rewrite	Event-triggered ETCS
Loop unrolling	Reach-avoid robot, Quadcopter
CESAR rewrite	Highway driving, CESAR benchmarks

- *CESAR rewrite.* This is the one-shot refinement and multi-shot unrolling from CESAR (Chapter 5 [70]) where a time-triggered control loop is replaced by running the ODE for an unbounded amount of time. Because of our framework's retrospective checks, the requirement that the original program should have an idempotence property ("action permanence") which was essential to the soundness of CESAR is relaxed.

B.4.5 Refinements

While synthesizing subvalue maps for loops, Algorithm 2 uses checks to test invariant candidates before accepting them. It is possible to use *refinements* to perform these checks. A game β is an *Angelic refinement* of game α when for any formula ψ, $\models \langle\beta\rangle\psi \rightarrow \langle\alpha\rangle\psi$. Some rewrites result in refinements, making the invariant candidate generated correct by construction. For instance, for the Angelic loop in Reach-avoid robot (Model 21), let α be the loop body (Line 2 and Line 3), and ϕ the postcondition (Line 4). The inductiveness check is $\mathsf{Inv} \rightarrow \langle(a:\alpha^*)\overset{\exists}{\downarrow}S\rangle\phi$. If Inv is computed by rewriting loop $\langle\alpha^*\rangle\phi$ as $\langle\alpha\rangle\phi$ (loop unrolling rewrite) and then setting $\mathsf{Inv} = \mathsf{simpl}(\langle\alpha\rangle\phi)$ while writing all the intermediate symbolic execution steps of simpl into the subvalue map S, then Inv will pass the check by construction. The reason is that $\langle a{:}\alpha\overset{\exists}{\downarrow}S\rangle\phi$ is a refinement of $\langle(a:\alpha)^*\overset{\exists}{\downarrow}S\rangle\phi$. Since $\models \mathsf{Inv} \rightarrow \langle a{:}\alpha\overset{\exists}{\downarrow}S\rangle\phi$ by construction, it follows that $\models \mathsf{Inv} \rightarrow \langle(a:\alpha)^*\overset{\exists}{\downarrow}S\rangle\phi$. We exploit such refinement properties in our implementation to soundly reduce the number of expensive checks performed.

B.5 Proofs

We prove the correctness of the auxiliary theorems and lemmas in the chapter. The proofs for the theorems and lemmas that appear in the main text are in Appendix D. Notation $S|_U$ is the restriction of the subvalue map S to the set of subgames $U \cup \{\mathsf{end}\}$.

Definition B.1 (Weak ordering)**.** We define a weaker ordering relation $\gtrsim$ on inductive subvalue maps that compares formulas at each subgame separately. For two inductive subvalue maps S_1 and S_2 for the game $a{:}\alpha$, S_1 is at least as good as S_2, written $S_1 \gtrsim S_2$, iff for each subgame b in $\mathsf{subgames}(a{:}\alpha)$, $\models S_2(b) \rightarrow S_1(b)$ and $\models S_2(\mathsf{end}) \rightarrow S_1(\mathsf{end})$.

Lemma B.1 (Ordering subsumption)**.** *If $S \gtrsim S'$, then $S \supseteq S'$.*

Proof. By definition of $\gtrsim$, for each subgame $b \in \mathsf{subgames}(\alpha) \cup \{\mathsf{end}\}$, $\models S'(b) \rightarrow S(b)$. Using the dGL Gödel generalization rule G, we infer $\models \langle(a{:}\alpha_{:b})\overset{\forall}{\downarrow}S'\rangle(S(b) \rightarrow S'(b))$ showing that $S \supseteq S'$ when interpreted as an Angelic strategy. When interpreted as a Demonic strategy, the same argument holds: using the dGL Gödel generalization rule G we can infer that $\models [(a{:}\alpha_{:b})[]\overset{\forall}{\downarrow}S'](S'(b) \rightarrow S(b))$. $\square$

Lemma B.2 (Subvalue Map Projection is a Refinement)**.** *The projection of an Angelic subvalue map onto a game α is an Angelic refinement of the game, i.e.,*

$$\models \langle a{:}\alpha\overset{\exists}{\downarrow}S\rangle\psi \rightarrow \langle\alpha\rangle\psi.$$

Dually, the projection of a Demonic subvalue map is a Demonic refinement.

Proof. We show this for Angelic subvalue maps, the proof for Demonic subvalue maps is symmetric. Intuitively this lemma is true because the projection of an Angelic subvalue map only adds Angelic tests, which make the game monotonically harder. The proof uses structural induction based on the shape of $a{:}\alpha$. When $a{:}\alpha$ is:

- Atomic and not controlled by Angel, i.e., $\alpha \in \{x := e, x := *, ?Q, !Q, \{x' = f(x) \,\&\, Q\}^d\}$, the result is immediate since $a{:}\alpha \underset{\downarrow}{\exists} S = \alpha$.
- Atomic and controlled by Angel, i.e., $\alpha \in \{x := \otimes, \{x' = f(x) \,\&\, Q\}\}$, then $a{:}\alpha \underset{\downarrow}{\exists} S = \alpha\,;\, ?S(\mathsf{end})$. By the dGL axioms $\langle ? \rangle$ and $\langle ; \rangle$, $\models \langle \alpha\,;\, ?S(\mathsf{end})\rangle\psi \leftrightarrow \langle\alpha\rangle(\psi \land S(\mathsf{end}))$. By the dGL axiom M, $\models \langle\alpha\rangle(\psi \land S(\mathsf{end})) \rightarrow \langle\alpha\rangle\psi$. By transitivity of implication, $\models \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\psi \rightarrow \langle\alpha\rangle\psi$.
- $g{:}\gamma \cup d{:}\delta$, then $a{:}\alpha \underset{\downarrow}{\exists} S = (?S(g)\,;\, g{:}\gamma \underset{\downarrow}{\exists} S) \cup (?S(d)\,;\, d{:}\delta \underset{\downarrow}{\exists} S)$.
  By the dGL axiom $\langle \cup \rangle$,
  $\models \langle (?S(g)\,;\, g{:}\gamma \underset{\downarrow}{\exists} S) \cup (?S(d)\,;\, d{:}\delta \underset{\downarrow}{\exists} S)\rangle\psi \leftrightarrow \langle ?S(g)\,;\, g{:}\gamma \underset{\downarrow}{\exists} S\rangle\psi \lor \langle ?S(d)\,;\, d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi$. By the dGL axioms $\langle ; \rangle$ and $\langle \cup \rangle$,
  $\models \langle ?S(g)\,;\, g{:}\gamma \underset{\downarrow}{\exists} S\rangle\psi \lor \langle ?S(d)\,;\, d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi \leftrightarrow (S(g) \land \langle g{:}\gamma \underset{\downarrow}{\exists} S\rangle\psi) \lor (S(d) \land \langle d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi)$.
  By conjunction elimination,
  $\models (S(g) \land \langle g{:}\gamma \underset{\downarrow}{\exists} S\rangle\psi) \lor (S(d) \land \langle d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi) \rightarrow \langle a{:}\gamma \underset{\downarrow}{\exists} S\rangle\psi \lor \langle d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi$.
  By the inductive hypothesis,
  $\models \langle g{:}\gamma \underset{\downarrow}{\exists} S\rangle\psi \rightarrow \langle\gamma\rangle\psi$ and $\models \langle d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi \rightarrow \langle\delta\rangle\psi$.
  By the transitivity of implication, $\models \langle g{:}\gamma \underset{\downarrow}{\exists} S\rangle\psi \lor \langle d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi \rightarrow \langle\gamma\rangle\psi \lor \langle\delta\rangle\psi$.
  By the dGL axiom $\langle \cup \rangle$,
  $\models \langle\gamma\rangle\psi \lor \langle\delta\rangle\psi \leftrightarrow \langle\gamma \cup \delta\rangle\psi$.
- $g{:}\gamma \cap d{:}\delta$, then $a{:}\alpha \underset{\downarrow}{\exists} S = g{:}\gamma \underset{\downarrow}{\exists} S \cap d{:}\delta \underset{\downarrow}{\exists} S$. By the dGL axiom $[\cup]$, $\models \langle g{:}\gamma \underset{\downarrow}{\exists} S \cap d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi \leftrightarrow \langle g{:}\gamma \underset{\downarrow}{\exists} S\rangle\psi \land \langle d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi$. By the inductive hypothesis, $\models \langle g{:}\gamma \underset{\downarrow}{\exists} S\rangle\psi \rightarrow \langle\gamma\rangle\psi$ and $\models \langle d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi \rightarrow \langle\delta\rangle\psi$. Thus, $\models \langle g{:}\gamma \underset{\downarrow}{\exists} S\rangle\psi \land \langle d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi \rightarrow \langle\gamma\rangle\psi \land \langle\delta\rangle\psi$. By the dGL axiom $[\cup]$, $\models \langle\gamma\rangle\psi \land \langle\delta\rangle\psi \leftrightarrow \langle\gamma \cap \delta\rangle\psi$.
- $g{:}\gamma; d{:}\delta$, then $a{:}\alpha \underset{\downarrow}{\exists} S = g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(d))\,;\, d{:}\delta \underset{\downarrow}{\exists} S$.
  Using the dGL axiom $\langle ; \rangle$,
  $\models \langle g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(d))\,;\, d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi \leftrightarrow \langle g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(d))\rangle\langle d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi$. By the inductive hypothesis, $\models \langle d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi \rightarrow \langle\delta\rangle\psi$.
  Thus, $\models \langle g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(d))\rangle\langle d{:}\delta \underset{\downarrow}{\exists} S\rangle\psi \rightarrow \langle g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(d))\rangle\,\langle\delta\rangle\psi$. Further, by the inductive hypothesis, $\models \langle g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(d))\rangle\langle\delta\rangle\psi \rightarrow \langle\gamma\rangle\langle\delta\rangle\psi$. By the dGL axiom $\langle ; \rangle$, $\models \langle\gamma\rangle\langle\delta\rangle\psi \leftrightarrow \langle\gamma; \delta\rangle\psi$.
- $a{:}(g{:}\gamma)^*$, then $a{:}\alpha \underset{\downarrow}{\exists} S = (?S(g)\,;\, g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(s)))^*; ?S(\mathsf{end})$. By dGL axioms $\langle ; \rangle$, $\langle ? \rangle$, and M, $\models \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\psi \rightarrow \langle (?S(g)\,;\, g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(a)))^*\rangle\,\psi$. By the dGL proof rule FP,
  $$\frac{\psi \lor \langle (?S(g)\,;\, g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(a))\rangle\,\langle\gamma^*\rangle\psi \rightarrow \langle\gamma^*\rangle\psi}{\langle (?S(g)\,;\, g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(a)))^*\rangle\,\psi \rightarrow \langle\gamma^*\rangle\psi}\,.$$
  Now, $\langle (?S(g)\,;\, g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(a)))\rangle\,\langle\gamma^*\rangle\psi$, per the inductive hypothesis, implies $\langle\gamma\rangle\langle\gamma^*\rangle\psi$. Per dGL axiom $\langle * \rangle$, $\langle\gamma\rangle\langle\gamma^*\rangle\psi \leftrightarrow \langle\gamma^*\rangle\psi$. This allows us to show the premise, since $\models \psi \lor \langle\gamma^*\rangle\psi \rightarrow \langle\gamma^*\rangle\psi$.
- $a{:}(g{:}\gamma)^\times$, then $a{:}\alpha \underset{\downarrow}{\exists} S = \left(g{:}\gamma \underset{\downarrow}{\exists} S\right)^\times$. By the dGL proof rule loop,

$$\dfrac{\langle (g{:}\gamma \underset{\downarrow}{\exists} S)^{\times} \rangle\, \psi \vdash J \qquad J \vdash \langle \gamma \rangle\, J \qquad J \vdash \psi}{\langle (g{:}\gamma \underset{\downarrow}{\exists} S)^{\times} \rangle\, \psi \vdash \langle \gamma^{\times} \rangle\, \psi} \ .$$

Setting invariant J to $\langle (g{:}\gamma \underset{\downarrow}{\exists} S)^{\times} \rangle\, \psi$, the first premise is immediate.
The second premise is now
$\langle (g{:}\gamma \underset{\downarrow}{\exists} S)^{\times} \rangle\, \psi \vdash \langle \gamma \rangle\, \langle (g{:}\gamma \underset{\downarrow}{\exists} S)^{\times} \rangle\, \psi$.
By $\langle {}^{\times} \rangle$, $\langle (g{:}\gamma \underset{\downarrow}{\exists} S)^{\times} \rangle\, \psi \rightarrow \langle g{:}\gamma \underset{\downarrow}{\exists} S \rangle\, \langle (g{:}\gamma \underset{\downarrow}{\exists} S)^{\times} \rangle\, \psi$.
By the inductive hypothesis, $\vDash \langle g{:}\gamma \underset{\downarrow}{\exists} S \rangle\, \langle (g{:}\gamma \underset{\downarrow}{\exists} S)^{\times} \rangle\, \psi \rightarrow \langle \gamma \rangle\, \langle (g{:}\gamma \underset{\downarrow}{\exists} S)^{\times} \rangle\, \psi$, thus proving the second premise.
The third premise is proved as follows.

$$\langle{}^{\times}\rangle\,\dfrac{\mathrm{WL}\,\dfrac{\mathrm{id}\,\dfrac{*}{\psi \vdash \psi}}{\psi \wedge \langle g{:}\gamma \underset{\downarrow}{\exists} S \rangle\, \langle (g{:}\gamma \underset{\downarrow}{\exists} S)^{\times} \rangle\, \psi \vdash \psi}}{\langle (g{:}\gamma \underset{\downarrow}{\exists} S)^{\times} \rangle\, \psi \vdash \psi}$$

□

Lemma B.3 (MPC Subvalue Map Projection Equivalence)**.** *The projection game* $\langle a{:}\alpha \underset{\downarrow}{\exists} S \rangle \phi$ *of the MPC Angelic subvalue map (Def. 6.9) is equivalent to the original game* $\langle \alpha \rangle \phi$*. Dually, the projection game* $[a\ {:}\alpha[] \underset{\downarrow}{\exists} S] \phi$ *of the MPC Demonic subvalue map (Def. 6.9) is equivalent to the original game* $[\alpha] \phi$*.*

Proof. We show this for Angelic subvalue maps, the proof for Demonic subvalue maps is symmetric. Intuitively this lemma is true because the projection of an Angelic subvalue map only adds Angelic tests, which make the game monotonically harder. The proof uses structural induction based on the shape of $a{:}\alpha$. Note that for the MPC solution, end is ϕ. When $a{:}\alpha$ is:

- Atomic and not controlled by Angel, i.e., $\alpha \in \{x := e, x := *, ?Q, !Q, \{x' = f(x)\ \&\ Q\}^{d}\}$, the result is immediate since $a{:}\alpha \underset{\downarrow}{\exists} S = \alpha$.
- Atomic and controlled by Angel, i.e., $\alpha \in \{x := \otimes, \{x' = f(x)\ \&\ Q\}\}$, then $a{:}\alpha \underset{\downarrow}{\exists} S = \alpha\, ;\, ?\phi$. By the dGL axioms $\langle ? \rangle$ and $\langle ; \rangle$, $\vDash \langle \alpha\, ;\, ?\phi \rangle \phi \leftrightarrow \langle \alpha \rangle (\phi \wedge \phi) \leftrightarrow \langle \alpha \rangle \phi$.
- $g{:}\gamma \cup d{:}\delta$, then $a{:}\alpha \underset{\downarrow}{\exists} S = (?S(g)\, ;\, g{:}\gamma \underset{\downarrow}{\exists} S) \cup (?S(d)\, ;\, d{:}\delta \underset{\downarrow}{\exists} S)$.
  $\langle (?S(g)\, ;\, g{:}\gamma \underset{\downarrow}{\exists} S) \cup (?S(d)\, ;\, d{:}\delta \underset{\downarrow}{\exists} S) \rangle \phi$
  $\leftrightarrow \langle ?S(g)\, ;\, g{:}\gamma \underset{\downarrow}{\exists} S \rangle \phi \vee \langle ?S(d)\, ;\, d{:}\delta \underset{\downarrow}{\exists} S \rangle \phi$ (dGL axiom $\langle \cup \rangle$)
  $\leftrightarrow (S(g) \wedge \langle g{:}\gamma \underset{\downarrow}{\exists} S \rangle \phi) \vee (S(d) \wedge \langle d{:}\delta \underset{\downarrow}{\exists} S \rangle \phi)$ (dGL axioms $\langle ; \rangle$, $\langle \cup \rangle$)
  $\leftrightarrow (S(g) \wedge \langle \gamma \rangle \phi) \vee (S(d) \wedge \langle \delta \rangle \phi)$ (inductive hypothesis)
  $\leftrightarrow (\langle \gamma \rangle \phi \wedge \langle \gamma \rangle \phi) \vee (\langle \delta \rangle \phi \wedge \langle \delta \rangle \phi)$ (MPC definition)
  $\leftrightarrow \langle \gamma \rangle \phi \vee \langle \delta \rangle \phi$.
- $g{:}\gamma \cap d{:}\delta$, then $a{:}\alpha \underset{\downarrow}{\exists} S = g{:}\gamma \underset{\downarrow}{\exists} S \cap d{:}\delta \underset{\downarrow}{\exists} S$.
  $\langle g{:}\gamma \underset{\downarrow}{\exists} S \cap d{:}\delta \underset{\downarrow}{\exists} S \rangle \phi$
  $\leftrightarrow \langle g{:}\gamma \underset{\downarrow}{\exists} S \rangle \phi \wedge \langle d{:}\delta \underset{\downarrow}{\exists} S \rangle \phi$ (dGL axiom $[\cup]$)
  $\leftrightarrow \langle \gamma \rangle \phi \wedge \langle \delta \rangle \phi$ (inductive hypothesis)
  $\leftrightarrow \langle \gamma \cap \delta \rangle \phi$ (dGL axiom $[\cup]$).

- $g{:}\gamma; d{:}\delta$, then $a{:}\alpha \underset{\downarrow}{\fatsemi} S = g{:}\gamma \underset{\downarrow}{\fatsemi} S(\mathsf{end} \mapsto S(d))\ ;\ d{:}\delta \underset{\downarrow}{\fatsemi} S$.
  $\langle g{:}\gamma \underset{\downarrow}{\fatsemi} S(\mathsf{end} \mapsto S(d))\ ;\ d{:}\delta \underset{\downarrow}{\fatsemi} S\rangle\phi$
  $\leftrightarrow \langle g{:}\gamma \underset{\downarrow}{\fatsemi} S(\mathsf{end} \mapsto S(d))\rangle\langle d{:}\delta \underset{\downarrow}{\fatsemi} S\rangle\phi$ (dGL axiom [;])
  $\leftrightarrow \langle g{:}\gamma \underset{\downarrow}{\fatsemi} S(\mathsf{end} \mapsto S(d))\rangle\langle\delta\rangle\phi$ (inductive hypothesis)
  $\leftrightarrow \langle\gamma\rangle\langle\delta\rangle\phi$ (inductive hypothesis)
  $\leftrightarrow \langle\gamma;\delta\rangle\phi$ (dGL axiom [;]).
- $a{:}(g{:}\gamma)^*$
  By Lemma B.2, $\models \langle a{:}\alpha \underset{\downarrow}{\fatsemi} S\rangle\,\phi \rightarrow \langle\gamma^*\rangle\,\phi$.
  We now show the opposite direction, i.e., $\models \langle\gamma^*\rangle\,\phi \rightarrow \langle a{:}\alpha \underset{\downarrow}{\fatsemi} S\rangle\,\phi$.
  Since $S(a) = \langle\gamma^*\rangle\,\phi$, we have $\models \langle\gamma^*\rangle\,\phi \rightarrow S(a)$. From $a{:}\alpha \models S$, we get $\models S(a) \rightarrow \langle a{:}\alpha \underset{\downarrow}{\fatsemi} S\rangle\,\phi$.
  Thus, by transitivity of implication, $\models \langle\gamma^*\rangle\,\phi \rightarrow \langle a{:}\alpha \underset{\downarrow}{\fatsemi} S\rangle\,\phi$.
- $a{:}(g{:}\gamma)^\times$
  By Lemma B.2, $\models \langle a{:}\alpha \underset{\downarrow}{\fatsemi} S\rangle\,\phi \rightarrow \langle\gamma^\times\rangle\,\phi$. We now show the opposite direction, i.e., $\models \langle\gamma^\times\rangle\,\phi \rightarrow \langle a{:}\alpha \underset{\downarrow}{\fatsemi} S\rangle\,\phi$.
  Since $S(a) = \langle\gamma^\times\rangle\,\phi$, we have $\models \langle\gamma^\times\rangle\,\phi \rightarrow S(a)$. We now show that $\models S(a) \rightarrow \langle a{:}\alpha \underset{\downarrow}{\fatsemi} S\rangle\,\phi$, so that by transitivity of implication, $\models \langle\gamma^\times\rangle\,\phi \rightarrow \langle a{:}\alpha \underset{\downarrow}{\fatsemi} S\rangle\,\phi$.
  By the dGL proof rule loop,
  $$\frac{S(a) \vdash J \quad J \vdash \langle g{:}\gamma \underset{\downarrow}{\fatsemi} S(\mathsf{end} \mapsto S(a))\rangle\, J \quad J \vdash \phi}{S(a) \vdash \langle\big(g{:}\gamma \underset{\downarrow}{\fatsemi} S(\mathsf{end} \mapsto S(a))\big)^\times\rangle\,\phi}\ .$$
  Setting invariant J to $S(a)$, the first premise is immediate. The last premise is $S(a) \vdash \phi$ which follows from $S(a) \leftrightarrow \phi \wedge \langle\gamma\rangle\,\langle\gamma^\times\rangle\,\phi$ per dGL axiom $\langle^\times\rangle$. Only the middle premise remains. It is $S(a) \vdash \langle g{:}\gamma \underset{\downarrow}{\fatsemi} S(\mathsf{end} \mapsto S(a))\rangle\, S(a)$.
  Per the inductive hypothesis, $\models \langle g{:}\gamma \underset{\downarrow}{\fatsemi} S(\mathsf{end} \mapsto S(a))\rangle\, S(a) \leftrightarrow \langle\gamma\rangle S(a)$. Thus, the middle premise can be rewritten as $S(a) \vdash \langle\gamma\rangle S(a)$. Recall that $S(a) = \langle\gamma^\times\rangle\,\phi$, so that the middle premise is $\langle\gamma^\times\rangle\,\phi \vdash \langle\gamma\rangle\langle\gamma^\times\rangle\,\phi$. By the dGL axiom $\langle^\times\rangle$, $\langle\gamma^\times\rangle\,\phi \rightarrow \langle\gamma\rangle\langle\gamma^\times\rangle\,\phi$, thus proving the middle premise.

□

Lemma B.4 (MPC Valid). *The MPC solution (Def. 6.9) S for a game $a{:}\alpha$ for Angel winning condition ϕ is an inductive subvalue map compatible with ϕ: $a{:}\alpha \models S$. Dually, the MPC solution S for a game $a{:}\alpha$ for Demon winning condition ϕ is an inductive subvalue map: $a{:}\alpha[]\!\models S$.*

Proof. Proved by induction on the structure of the game $a{:}\alpha$. We show this for the Angel MPC solution; the proof for the Demonic MPC solution is analogous. Firstly, for the MPC solution, $S(\mathsf{end}) = \phi$, so ϕ is a compatible winning condition. The base case is when $a{:}\alpha$ is atomic, i.e., $\alpha \in \{x := e, x := *, ?f, !f, \{x' = f(x)\ \&\ Q\}, \{x' = f(x)\ \&\ Q\}^d\}$. In this case the MPC subvalue map consists of the mapping from a to $\langle\alpha\rangle\phi$ (along with $\mathsf{end} \mapsto \phi$), which matches the corresponding condition for $a{:}\alpha \models S$ in Def. 6.8. In the recursive case, if $a{:}\alpha$ has the structure:

1. $g{:}\gamma \cup d{:}\delta$: by the inductive hypothesis, as $S|_{\mathsf{subgames}(g:\gamma)}$ is the MPC solution for game γ and $S|_{\mathsf{subgames}(d:\delta)}$ is the MPC solution for game δ, $g{:}\gamma \models S|_{\mathsf{subgames}(g:\gamma)}$ and $d{:}\delta \models S|_{\mathsf{subgames}(d:\delta)}$. So, $g{:}\gamma \models S$ and $d{:}\delta \models S$. It remains to show that $S(a) \rightarrow S(g) \vee S(d)$. By

the definition of the MPC solution, $S(a) = \langle g{:}\gamma \cup d{:}\delta\rangle\phi$, $S(g) = \langle g{:}\gamma\rangle\phi$ and $S(d) = \langle d{:}\delta\rangle\phi$. By the dGL rule $\langle\cup\rangle$, $\langle g{:}\gamma \cup d{:}\delta\rangle\phi = \langle g{:}\gamma\rangle\phi \vee \langle d{:}\delta\rangle\phi$. Thus, $S(a) \to S(g) \vee S(d)$.

2. $g{:}\gamma \cap d{:}\delta$: by the inductive hypothesis, as $S|_{\mathsf{subgames}(g:\gamma)}$ is the MPC solution for game γ and $S|_{\mathsf{subgames}(d:\delta)}$ is the MPC solution for game δ, $g{:}\gamma \models S|_{\mathsf{subgames}(g:\gamma)}$ and $d{:}\delta \models S|_{\mathsf{subgames}(d:\delta)}$. So, $g{:}\gamma \models S$ and $d{:}\delta \models S$. It remains to show that $S(a) \to S(g) \wedge S(d)$. By the definition of the MPC solution, $S(a) = \langle g{:}\gamma \cap d{:}\delta\rangle\phi$, $S(g) = \langle g{:}\gamma\rangle\phi$ and $S(d) = \langle d{:}\delta\rangle\phi$. By the dGL rule $\langle\cap\rangle$, $\langle g{:}\gamma \cap d{:}\delta\rangle\phi = \langle g{:}\gamma\rangle\phi \wedge \langle d{:}\delta\rangle\phi$. Thus, $S(a) \to S(g) \wedge S(d)$.
3. $g{:}\gamma\,;\, d{:}\delta$: by the inductive hypothesis, as $S|_{\mathsf{subgames}(d:\delta)}$ is the MPC solution for game δ and Angel winning condition ϕ, $d{:}\delta \models S|_{\mathsf{subgames}(d:\delta)}$. So $d{:}\delta \models S$. Further, as $S|_{\mathsf{subgames}(g:\gamma)}$ is the MPC solution for game γ and Angel winning condition $\langle d{:}\delta\rangle\phi = S(\delta)$, by the inductive hypothesis, $g{:}\gamma \models S(\mathsf{end} \mapsto S(\delta))|_{\mathsf{subgames}(g:\gamma)}$. So $g{:}\gamma \models S(\mathsf{end} \mapsto S(\delta))$. It remains to show that $S(a) \to S(g)$ which is immediate from the definition of the MPC solution since $a{:}\alpha_{a:} = a{:}\alpha_{g:}$ in this case.
4. $g{:}\gamma^*$: observe that $S|_{\mathsf{subgames}(g:\gamma)}$ is the MPC solution for game γ and Angel winning condition $\langle\gamma^*\rangle\phi$. $S(a) = \langle\gamma^*\rangle\phi$ and $S(a) \to \phi$, so $\langle\gamma^*\rangle\phi = S(a) \vee \phi$, so $S|_{\mathsf{subgames}(g:\gamma)}$ is the MPC solution for game γ and Angel winning condition $S(a) \vee \phi$. Then, by the inductive hypothesis, $g{:}\gamma \models S(\mathsf{end} \mapsto S(a))|_{\mathsf{subgames}(g:\gamma)}$. So $g{:}\gamma \models S(\mathsf{end} \mapsto S(a))$. The remaining condition is $S(a) \to \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\phi$. For MPC solution S, $\langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\,\phi = \langle\gamma^*\rangle\phi$ (Lemma B.3). Since $S(a) = \langle\gamma^*\rangle\phi$, the condition is satisfied.
5. $g{:}\gamma^\times$: observe that $S|_{\mathsf{subgames}(g:\gamma)}$ is the MPC solution for game γ and Angel winning condition $\langle\gamma^\times\rangle\phi$.
   By definition, $S(a) = \langle\gamma^\times\rangle\phi$.
   So, by the inductive hypothesis, $g{:}\gamma \models S(\mathsf{end} \mapsto S(a))|_{\mathsf{subgames}(g:\gamma)}$.
   Thus $g{:}\gamma \models S(\mathsf{end} \mapsto S(a))$.
   Next we show that $\models S(a) \to S(g) \wedge \phi$. Using the dGL axiom $\langle{}^\times\rangle$, $\langle\gamma^\times\rangle\phi = \langle\gamma\rangle\langle\gamma^\times\rangle\phi \wedge \phi$. Substituting back the MPC definitions of $S(a) = \langle\gamma^\times\rangle\phi$ and $S(g) = \langle\gamma\rangle S(a)$, this means $S(a) = S(g) \wedge \phi$. So, $\models S(a) \to S(g) \wedge \phi$.

□

Lemma B.5 (Subvalue maps win)**.** *For any game $a{:}\alpha$, Angelic subvalue map S, and compatible Angel winning condition ϕ, $\langle\alpha\rangle\,\phi \models \langle a{:}\alpha \underset{\downarrow}{\forall} S\rangle\,\phi$.*

Proof. The proof follows from structural induction along with the application of the usual dGL axioms and proof rules.

- If $a{:}\alpha$ is atomic and not controlled by Angel, i.e., $\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x)\ \&\ Q\}^d\}$, then $\langle\alpha\rangle\,\phi = \langle a{:}\alpha \underset{\downarrow}{\forall} S\rangle\,\phi$ and the result is immediate.
- If $a{:}\alpha = x := *$, then $a{:}\alpha \underset{\downarrow}{\forall} S$ is $x := \otimes\,;\, !S(\mathsf{end})$. Thus we must show that

$$\langle\alpha\rangle\,\phi \models \langle x := \otimes\,;\, !S(\mathsf{end})\rangle\,\phi.$$

By the dGL axioms $\langle;\rangle$ and $[?]$, this is

$$\langle\alpha\rangle\,\phi \models \langle x := \otimes\rangle\,(S(\mathsf{end}) \to \phi).$$

Since ϕ is a compatible Angel winning condition, $S(\mathsf{end}) \to \phi$ holds. Thus, we need to show

$$\langle \alpha \rangle \, \phi \models \langle x := \otimes \rangle \, \top.$$

By the semantics of nondeterministic assignments in dGL, the succedent is $\top$, completing the proof.

- If $a{:}\alpha$ is $\{x' = f(x) \mathbin{\&} Q\}$, then the argument is similar to the previous case. $a{:}\alpha \mathbin{\forall\!\!\downarrow} S$ is $\{x' = f(x) \mathbin{\&} Q\}^d \, ; \, !S(\mathsf{end})$. Thus we must show that

  $$\langle \alpha \rangle \, \phi \models \langle \{x' = f(x) \mathbin{\&} Q\}^d \, ; \, !S(\mathsf{end}) \rangle \, \phi.$$

  Applying the dGL axioms $\langle ; \rangle$, $\langle ^d \rangle$, and $[?]$, this is

  $$\langle \alpha \rangle \, \phi \models \langle \{x' = f(x) \mathbin{\&} Q\}^d \rangle \, (S(\mathsf{end}) \to \phi).$$

  Since ϕ is a compatible Angel winning condition, $S(\mathsf{end}) \to \phi$ holds. Thus, we need to show

  $$\langle \alpha \rangle \, \phi \models \langle \{x' = f(x) \mathbin{\&} Q\}^d \rangle \, \top.$$

  By the semantics of differential games in dGL, the succedent is $\top$, completing the proof.
- If $a{:}\alpha = g{:}\gamma \cup d{:}\delta$, then $a{:}\alpha \mathbin{\forall\!\!\downarrow} S$ is $(!S(g) \, ; \, g{:}\gamma \mathbin{\forall\!\!\downarrow} S) \cap (!S(d) \, ; \, d{:}\delta \mathbin{\forall\!\!\downarrow} S)$. Thus we must show that

  $$\langle \alpha \rangle \, \phi \models \langle (!S(g) \, ; \, g{:}\gamma \mathbin{\forall\!\!\downarrow} S) \cap (!S(d) \, ; \, d{:}\delta \mathbin{\forall\!\!\downarrow} S) \rangle \, \phi.$$

  By the dGL axioms $\langle ; \rangle$ and $[\cup]$, this is

  $$\langle \alpha \rangle \, \phi \models (S(g) \to \langle g{:}\gamma \mathbin{\forall\!\!\downarrow} S \rangle \, \phi) \wedge (S(d) \to \langle d{:}\delta \mathbin{\forall\!\!\downarrow} S \rangle \, \phi).$$

  But by the inductive hypothesis, we have $\langle \gamma \rangle \, \phi \models \langle g{:}\gamma \mathbin{\forall\!\!\downarrow} S \rangle \, \phi$ and $\langle \delta \rangle \, \phi \models \langle d{:}\delta \mathbin{\forall\!\!\downarrow} S \rangle \, \phi$. $S(g) \models \langle \gamma \rangle \, \phi$ and $S(d) \models \langle \delta \rangle \, \phi$ by the definition of a subvalue map. By the proof rules $\to$R and WL, the following two formulas hold: $S(g) \to \langle g{:}\gamma \mathbin{\forall\!\!\downarrow} S \rangle \, \phi$ and $S(d) \to \langle d{:}\delta \mathbin{\forall\!\!\downarrow} S \rangle \, \phi$. Thus, we have $\langle \alpha \rangle \, \phi \models (S(g) \to \langle g{:}\gamma \mathbin{\forall\!\!\downarrow} S \rangle \, \phi) \wedge (S(d) \to \langle d{:}\delta \mathbin{\forall\!\!\downarrow} S \rangle \, \phi)$, completing the proof for this case.
- If $a{:}\alpha = g{:}\gamma^*$, then $\langle a{:}\alpha \mathbin{\forall\!\!\downarrow} S \rangle \, \phi = \langle (!S(g) \, ; \, g{:}\gamma \mathbin{\forall\!\!\downarrow} S(\mathsf{end} \mapsto S(a)))^\times \, ; \, !S(\mathsf{end}) \rangle \, \phi$.
  Applying axioms $\langle ; \rangle$ and $[?]$, this is

  $$\langle (!S(g) \, ; \, g{:}\gamma \mathbin{\forall\!\!\downarrow} S(\mathsf{end} \mapsto S(a)))^\times \rangle \, (S(\mathsf{end}) \to \phi)$$

  .
  Since ϕ is a compatible Angel winning condition, $\models S(\mathsf{end}) \to \phi$.
  Thus, we need to show

  $$\langle \alpha \rangle \, \phi \models \langle (!S(g) \, ; \, g{:}\gamma \mathbin{\forall\!\!\downarrow} S(\mathsf{end} \mapsto S(a)))^\times \rangle \, (\top)$$

  This we do using the loop rule with invariant $\langle \alpha \rangle \, \phi$. The inductive stop is shown using the inductive hypothesis, i.e., $\langle \gamma \rangle \, \phi \models \langle g{:}\gamma \mathbin{\forall\!\!\downarrow} S(\mathsf{end} \mapsto S(a)) \rangle \, \langle \alpha \rangle \, \phi$. Here, $\langle \alpha \rangle \, \phi$ is a compatible winning condition because $S(a) \to \langle \alpha \rangle \, \phi$ holds by the definition of a subvalue map.

- If $a{:}\alpha = g{:}\gamma \cap d{:}\delta$, then $a{:}\alpha \overset{\forall}{\downarrow} S$ is $(g{:}\gamma \overset{\forall}{\downarrow} S) \cap (d{:}\delta \overset{\forall}{\downarrow} S)$. We must show that

  $$\langle \gamma \cap \delta \rangle \, \phi \models \langle (g{:}\gamma \overset{\forall}{\downarrow} S) \cap (d{:}\delta \overset{\forall}{\downarrow} S) \rangle \, \phi.$$

  By the dGL axiom $\langle \cup \rangle$, this is

  $$\langle \gamma \rangle \, \phi \wedge \langle \delta \rangle \, \phi \models \langle g{:}\gamma \overset{\forall}{\downarrow} S \rangle \, \phi \wedge \langle d{:}\delta \overset{\forall}{\downarrow} S \rangle \, \phi.$$

  Applying the proof rules $\wedge$R and WL, the goals to show are

  $$\langle \gamma \rangle \, \phi \models \langle g{:}\gamma \overset{\forall}{\downarrow} S \rangle \, \phi \text{ and } \langle \delta \rangle \, \phi \models \langle d{:}\delta \overset{\forall}{\downarrow} S \rangle \, \phi.$$

  Both of these follow from the inductive hypotheses.
- If $a{:}\alpha = g{:}\gamma \,;\, d{:}\delta$, then $a{:}\alpha \overset{\forall}{\downarrow} S$ is $(g{:}\gamma \overset{\forall}{\downarrow} S(\mathsf{end} \mapsto S(d))) \,;\, (d{:}\delta \overset{\forall}{\downarrow} S)$. We must show that

  $$\langle \gamma \,;\, \delta \rangle \, \phi \models \langle (g{:}\gamma \overset{\forall}{\downarrow} S(\mathsf{end} \mapsto S(d))) \,;\, (d{:}\delta \overset{\forall}{\downarrow} S) \rangle \, \phi.$$

  By the dGL axiom [;], this is

  $$\langle \gamma \rangle \, \langle \delta \rangle \, \phi \models \langle g{:}\gamma \overset{\forall}{\downarrow} S(\mathsf{end} \mapsto S(d)) \rangle \, \langle d{:}\delta \overset{\forall}{\downarrow} S \rangle \, \phi.$$

  By the inductive hypothesis, we know that $\langle \gamma \rangle \, \langle \delta \rangle \, \phi \rightarrow \langle g{:}\gamma \overset{\forall}{\downarrow} S(\mathsf{end} \mapsto S(d)) \rangle \, S(d)$ holds because $\langle \delta \rangle \, \phi$ is compatible with $S(\mathsf{end} \mapsto S(d))$, since by the definition of a subvalue map, we have $S(d) \rightarrow \langle \delta \rangle \, \phi$. Per inductive hypothesis, we also know that $\langle \delta \rangle \, \phi \models \langle d{:}\delta \overset{\forall}{\downarrow} S \rangle \, \phi$. Thus by the transitivity of implication along with the monotonicity rule, we complete the proof.
- If $a{:}\alpha = g{:}\gamma^{\times}$, then $a{:}\alpha \overset{\forall}{\downarrow} S$ is $(g{:}\gamma \overset{\forall}{\downarrow} S(\mathsf{end} \mapsto S(a))^{\times})$. We must show that

  $$\langle \gamma^{\times} \rangle \, \phi \models \langle (g{:}\gamma \overset{\forall}{\downarrow} S(\mathsf{end} \mapsto S(a)))^{\times} \rangle \, \phi.$$

  We show this by using the dGL loop rule loop with invariant $\langle \gamma^{\times} \rangle \, \phi$. The inductive stop is shown using the inductive hypothesis, i.e.,

  $$\langle \gamma \rangle \, \langle \gamma^{\times} \rangle \, \phi \models \langle g{:}\gamma \overset{\forall}{\downarrow} S(\mathsf{end} \mapsto S(a)) \rangle \, \langle \gamma^{\times} \rangle \, \phi,$$

  where $\langle \gamma^{\times} \rangle \, \phi$ is a compatible winning condition because $S(a) \rightarrow \langle \gamma^{\times} \rangle \, \phi$ holds by the definition of a subvalue map. The invariant holds initially by id and implies the postcondition by the dGL axiom $\langle ^{\times} \rangle$.

$\square$

Lemma B.6 (Inductive subvalue maps have some strategy)**.** *For any game $a{:}\alpha$, Angelic inductive subvalue map S, and compatible Angel winning condition ϕ, $S(a) \models \langle a{:}\alpha \overset{\exists}{\downarrow} S \rangle \, \phi$.*

Proof. This follows from structural induction, unwrapping the inductive subvalue map definition, and applying the usual dGL axioms and proof rules.

- If $a{:}\alpha$ is atomic and not controlled by Angel, i.e., $\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x) \mathrel{\&} Q\}^d\}$, then $a{:}\alpha \underset{\downarrow}{\fatsemi} S = a{:}\alpha$. In these cases, by the definition of an inductive subvalue map, $\models S(a) \rightarrow \langle a{:}\alpha\rangle\, \mathsf{end}$. Further since ϕ is a compatible winning condition, $\models S(\mathsf{end}) \rightarrow \phi$. By the monotonicity rule, $S(a) \models \langle a{:}\alpha\rangle\, \phi$. This allows us to conclude that $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\fatsemi} S\rangle\, \phi$.
- If $a{:}\alpha = a{:}x := *$, then $a{:}\alpha \underset{\downarrow}{\fatsemi} S = x := *\,;\, ?S(\mathsf{end})$. Thus we must show that

  $$S(a) \models \langle x := *\,;\, ?S(\mathsf{end})\rangle\, \phi.$$

  By the dGL axioms $\langle ; \rangle$ and $\langle ? \rangle$, this is

  $$S(a) \models \langle x := *\rangle\, S(\mathsf{end}) \wedge \phi.$$

  Since ϕ is a compatible Angel winning condition, $S(\mathsf{end}) \rightarrow \phi$ holds. Applying the monotonicity rule, we must show that

  $$S(a) \models \langle x := *\rangle\, S(\mathsf{end}).$$

  By the definition of an inductive subvalue map for Angelic free assignments along with the $\rightarrow$R rule, we conclude that $S(a) \models \langle x := *\rangle\, S(\mathsf{end})$.
- If $a{:}\alpha = a{:}\{x' = f(x) \mathrel{\&} Q\}$, the proof follows exactly the same steps as the previous case.
- If $a{:}\alpha = g{:}\gamma \cup d{:}\delta$, then $a{:}\alpha \underset{\downarrow}{\fatsemi} S = (?S(g)\,;\, g{:}\gamma \underset{\downarrow}{\fatsemi} S) \cup (?S(d)\,;\, d{:}\delta \underset{\downarrow}{\fatsemi} S)$. Thus we must show that

  $$S(a) \models \langle (?S(g)\,;\, g{:}\gamma \underset{\downarrow}{\fatsemi} S) \cup (?S(d)\,;\, d{:}\delta \underset{\downarrow}{\fatsemi} S)\rangle\, \phi.$$

  By the dGL axioms $\langle ; \rangle$ and $\langle \cup \rangle$, this is

  $$S(a) \models (S(g) \rightarrow \langle g{:}\gamma \underset{\downarrow}{\fatsemi} S\rangle\, \phi) \vee (S(d) \rightarrow \langle d{:}\delta \underset{\downarrow}{\fatsemi} S\rangle\, \phi).$$

  By the definition of an inductive subvalue map, we have $\models S(a) \rightarrow S(g) \vee S(d)$. Thus by proof rules cut, $\vee$L, $\vee$R, and WR it suffices to show the following two goals:

  $$S(g) \models S(g) \rightarrow \langle g{:}\gamma \underset{\downarrow}{\fatsemi} S\rangle\, \phi \text{ and } S(d) \models S(d) \rightarrow \langle d{:}\delta \underset{\downarrow}{\fatsemi} S\rangle\, \phi.$$

  Applying the rule $\rightarrow$R, the remaining goals hold per the inductive hypotheses, $S(g) \models \langle g{:}\gamma \underset{\downarrow}{\fatsemi} S\rangle\, \phi$ and $S(d) \models \langle d{:}\delta \underset{\downarrow}{\fatsemi} S\rangle\, \phi$.
- We show the case of Angel loop, which follows directly from the definition of an inductive subvalue map. If $a{:}\alpha = g{:}\gamma^*$, then by the definition of an inductive subvalue map, $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\fatsemi} S\rangle\, S(\mathsf{end})$. Since ϕ a compatible winning condition, we have $S(\mathsf{end}) \models \phi$. Using dGL rule M, we can conclude that $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\fatsemi} S\rangle\, \phi$.
- If $a{:}\alpha = g{:}\gamma\,;\, d{:}\delta$, then $a{:}\alpha \underset{\downarrow}{\fatsemi} S = g{:}\gamma \underset{\downarrow}{\fatsemi} S(\mathsf{end} \mapsto S(d))\,;\, d{:}\delta \underset{\downarrow}{\fatsemi} S$. Thus we must show that

  $$S(a) \models \langle g{:}\gamma \underset{\downarrow}{\fatsemi} S(\mathsf{end} \mapsto S(d))\,;\, d{:}\delta \underset{\downarrow}{\fatsemi} S\rangle\, \phi.$$

  By the dGL axioms $\langle ; \rangle$, this is

  $$S(a) \models \langle g{:}\gamma \underset{\downarrow}{\fatsemi} S(\mathsf{end} \mapsto S(d))\rangle\, \langle d{:}\delta \underset{\downarrow}{\fatsemi} S\rangle\, \phi.$$

By the inductive hypothesis along with the fact that $\models S(a) \rightarrow S(g)$ from the definition of inductive subvalue maps,

$$S(a) \models \langle {\scriptstyle g:}\gamma \overset{\exists}{\downarrow} S(\mathsf{end} \mapsto S(d)) \rangle \, S(d).$$

Additionally, by the inductive hypothesis,

$$S(d) \models \langle {\scriptstyle d:}\delta \overset{\exists}{\downarrow} S \rangle \, \phi.$$

Applying the M rule, we can conclude that

$$S(a) \models \langle {\scriptstyle g:}\gamma \overset{\exists}{\downarrow} S(\mathsf{end} \mapsto S(d)) \rangle \, \langle {\scriptstyle d:}\delta \overset{\exists}{\downarrow} S \rangle \, \phi,$$

completing the proof for this case.

- If ${\scriptstyle a:}\alpha = {\scriptstyle g:}\gamma \cap {\scriptstyle d:}\delta$, then ${\scriptstyle a:}\alpha \overset{\exists}{\downarrow} S = {\scriptstyle g:}\gamma \overset{\exists}{\downarrow} S \cap {\scriptstyle d:}\delta \overset{\exists}{\downarrow} S$. Thus we must show that

  $$S(a) \models \langle {\scriptstyle g:}\gamma \overset{\exists}{\downarrow} S \cap {\scriptstyle d:}\delta \overset{\exists}{\downarrow} S \rangle \, \phi.$$

  By the dGL axiom $\langle \cap \rangle$, this is

  $$S(a) \models \langle {\scriptstyle g:}\gamma \overset{\exists}{\downarrow} S \rangle \, \phi \wedge \langle {\scriptstyle d:}\delta \overset{\exists}{\downarrow} S \rangle \, \phi.$$

  By the definition of an inductive subvalue map, we have $S(a) \models S(g) \wedge S(d)$. Applying the proof rules $\wedge$R, cut, $\wedge$L and WL, the goals to show are

  $$S(g) \models \langle {\scriptstyle g:}\gamma \overset{\exists}{\downarrow} S \rangle \, \phi \text{ and } S(d) \models \langle {\scriptstyle d:}\delta \overset{\exists}{\downarrow} S \rangle \, \phi.$$

  Both of these follow from the inductive hypotheses, completing the proof for this case.
- If ${\scriptstyle a:}\alpha = {\scriptstyle g:}\gamma^{\times}$, then ${\scriptstyle a:}\alpha \overset{\exists}{\downarrow} S = ({\scriptstyle g:}\gamma \overset{\exists}{\downarrow} (S(\mathsf{end} \mapsto S(a))))^{\times}$. Thus we must show that

  $$S(a) \models \langle ({\scriptstyle g:}\gamma \overset{\exists}{\downarrow} (S(\mathsf{end} \mapsto S(a))))^{\times} \rangle \, S(\mathsf{end}).$$

  We use the loop rule with invariant $S(a)$. $S(a)$ holds initially by assumption. We show that is holds inductively, i.e., that

  $$S(a) \models \langle {\scriptstyle g:}\gamma \overset{\exists}{\downarrow} (S(\mathsf{end} \mapsto S(a))) \rangle \, S(a).$$

  By the definition of an inductive subvalue map, $\models S(a) \rightarrow S(g)$. Thus, after applying cut, we must show

  $$S(g) \models \langle {\scriptstyle g:}\gamma \overset{\exists}{\downarrow} (S(\mathsf{end} \mapsto S(a))) \rangle \, S(a).$$

  This holds by the inductive hypothesis. Finally, the invariant implies the postcondition by the definition of an inductive subvalue map, which requires that $\models S(a) \rightarrow S(\mathsf{end})$.

□

Lemma B.7 (Inductive subvalue maps win)**.** *For any game ${\scriptstyle a:}\alpha$, Angelic inductive subvalue map S, and compatible Angel winning condition ϕ, $S(a) \models \langle {\scriptstyle a:}\alpha \overset{\forall}{\downarrow} S \rangle \, \phi$.*

Proof. Follows from Lemma B.5 because an inductive subvalue map is a subvalue map (Theorem 6.3) and $S(a) \rightarrow \langle \alpha \rangle \, \phi$ holds per the definition of a subvalue map. □

B.6 Additional Definitions

This appendix provides additional definitions and constructions including the Demonic version for definitions whose Angelic subvalue map version is already defined in the main text, and dGL operational semantics concepts.

B.6.1 dGL Action Semantics

dGL actions can be interpreted as transformations from old state to new state. The definition of the functions is a part of dGL operational semantics [121][Appendix C], but we recall the definitions in Eq. (B.4), extended to labels and the extra action for sequence that this chapter uses in its presentation.

$$\begin{aligned}
&\lfloor (x := e) \rfloor_\sigma = \lfloor (x := e)^d \rfloor_\sigma = \sigma(x \mapsto e) \quad \lfloor \mathsf{act} \rfloor_\sigma = \sigma \text{ for } \mathsf{act} \in \{\mathfrak{g}, \mathfrak{s}, \mathfrak{l}, \mathfrak{r}, \mathfrak{c}, \mathfrak{g}^d, \mathfrak{s}^d, \mathfrak{l}^d, \mathfrak{r}^d, \text{label}\} \\
&\lfloor (?Q) \rfloor_\sigma = \begin{cases} \sigma & \text{if } \sigma \in [\![Q]\!] \\ \text{not defined} & \text{otherwise} \end{cases} \qquad \lfloor (!Q) \rfloor_\sigma = \begin{cases} \sigma & \text{if } \sigma \in [\![Q]\!] \\ \text{not defined} & \text{otherwise} \end{cases} \\
&\lfloor (\{x' = f(x) \& Q @ t\}) \rfloor_\sigma = \lfloor (\{x' = f(x) \& Q @ t\}^d) \rfloor_\sigma = \\
&\quad \varphi(t) \text{ for the unique (differentiable) } \varphi : [0, t] \to \text{States}, \varphi(0) = \sigma, \\
&\quad \text{for all } s \in [0, t] \left(\frac{\mathsf{d}\varphi(r)(x)}{\mathsf{d}r}(s) = f(\varphi(s)(x)) \land \varphi(s) \models Q \right). \\
&\quad \text{Not defined if no such } \varphi \text{ exists.}
\end{aligned} \tag{B.4}$$

B.6.2 Labeled Game Trees

We construct the labeled game tree, extending the operational game tree of dGL [121][Appendix C] with labels. The operator ˆ denotes concatenation under prefix closure. That is:

1. For two actions/labels act_1 and act_2, $\{\mathsf{act}_1 \hat{} \mathsf{act}_2\}$ is the set $\{\mathsf{act}_1, \mathsf{act}_1 \cdot \mathsf{act}_2\}$.
2. For an action act and a sequence of actions b, $\{\mathsf{act} \hat{} b\}$ is the set $\{\mathsf{act}, \mathsf{act} \cdot b\}$.
3. For two actions/labels act_1 and act_2, and sequence b, $\{\mathsf{act}_1 \hat{} \mathsf{act}_2 \hat{} b\}$ is the set $\{\mathsf{act}_1, \mathsf{act}_1 \cdot \mathsf{act}_2, \mathsf{act}_1 \cdot \mathsf{act}_2 \cdot b\}$.
4. For an action act and *a set of sequences* $\{a, b\}$, $\{\mathsf{act} \hat{} \{a, b\}\}$ is the set $\{\mathsf{act}, \mathsf{act} \cdot a, \mathsf{act} \cdot b\}$.

Thus, ˆ is overloaded to denote the concatenation of actions, sequences, and sets of sequences under prefix closure.

Definition B.2 (Labeled Semantics)**.** For any game $a{:}\alpha$ and state σ, the labeled semantics $g(a{:}\alpha)(\sigma)$ is defined as follows, where ˆ denotes concatenation under prefix closure (Appendix B.6.2).

$$\begin{aligned}
\mathcal{T}(x := \theta, \sigma) &\equiv \{a \hat{} x := \theta\} \\
\mathcal{T}(x := \otimes, \sigma) &\equiv \{a \hat{} x := \theta^d \ : \ \theta \in \mathbb{R}\}
\end{aligned}$$

$$
\begin{aligned}
\mathcal{T}(x := *, \sigma) &\equiv \{a\hat{}x := \theta \;:\; \theta \in \mathbb{R}\}\\
\mathcal{T}(\{x' = f(x)\ \&\ Q\}, \sigma) &\equiv \{a\hat{}(x' = f(x)\&Q@r) \;:\; r \in \mathbb{R}, r \geq 0, \varphi(0) = \sigma \text{ for some}\\
&\qquad \text{(differentiable) } \varphi : [0, r] \to \mathcal{S} \text{ such that } \frac{d\varphi(t)(x)}{dt}(\zeta) = [\![f(x)]\!]_{\varphi(\zeta)}\\
&\qquad \text{and } \varphi(\zeta) \in [\![Q]\!]^I \text{ for all } \zeta \leq r\}\\
\mathcal{T}(?Q, \sigma) &\equiv \{a\hat{}?Q\}\\
\mathcal{T}(!Q, \sigma) &\equiv \{a\hat{}!Q\}\\
\mathcal{T}(g{:}\gamma \cup d{:}\delta, s) &\equiv \{a\hat{}\mathfrak{l}\hat{}b \;:\; b \in g(g{:}\gamma)(\sigma)\} \cup\\
&\qquad \{a\hat{}\mathfrak{r}\hat{}b \;:\; b \in g(d{:}\delta)(\sigma)\}\\
\mathcal{T}(g{:}\gamma \cap d{:}\delta, \sigma) &\equiv \{a\hat{}\mathfrak{l}^d\hat{}b \;:\; b \in g(g{:}\gamma)(\sigma)\} \cup\\
&\qquad \{a\hat{}\mathfrak{r}^d\hat{}b \;:\; b \in g(d{:}\delta)(\sigma)\}\\
\mathcal{T}(g{:}\gamma\,;\, d{:}\delta, \sigma) &\equiv a\hat{}\mathfrak{c}\hat{}\mathcal{T}(g{:}\gamma, \sigma) \cup \bigcup_{\substack{t \in \mathsf{leaf}(a\hat{}\mathfrak{c}\hat{}\mathcal{T}(g:\gamma,\sigma))\\ \lfloor t \rfloor_\sigma \neq \mathsf{undefined}}} t\hat{}\mathcal{T}(d{:}\delta, \lfloor t \rfloor_\sigma)\\
\mathcal{T}((g{:}\gamma)^*, \sigma) &\equiv \bigcup_{n<\omega} f^n(\{(a\hat{}\mathfrak{s}), (a\hat{}\mathfrak{g})\})\\
&\qquad \text{where } f^n \text{ is the } n\text{-fold composition of the function}\\
&\qquad f(Z) \overset{\text{def}}{=} Z \cup \bigcup_{t\hat{}\mathfrak{g} \in \mathsf{leaf}(Z), v_u \in \mathcal{T}(\alpha, \lfloor t\hat{}\mathfrak{g} \rfloor_\sigma)} t\hat{}\mathfrak{g}\hat{}\\
&\qquad \left\{ \begin{array}{ll} v_u & \text{if } \lfloor t\hat{}\mathfrak{g}\hat{}v_u \rfloor_\sigma = \mathsf{undefined}\\ v_u\hat{}\{(a\hat{}\mathfrak{s}), (a\hat{}\mathfrak{g})\} & \text{otherwise} \end{array} \right\}\\
\mathcal{T}((g{:}\gamma)^\times, \sigma) &\equiv \bigcup_{n<\omega} f^n(\{(a\hat{}\mathfrak{s}^d), (a\hat{}\mathfrak{g}^d)\})\\
&\qquad \text{where } f^n \text{ is the } n\text{-fold composition of the function}\\
&\qquad f(Z) \overset{\text{def}}{=} Z \cup \bigcup_{t\hat{}\mathfrak{g}^d \in \mathsf{leaf}(Z), v_u \in \mathcal{T}(\alpha, \lfloor t\hat{}\mathfrak{g}^d \rfloor_\sigma)} t\hat{}\mathfrak{g}^d\hat{}\\
&\qquad \left\{ \begin{array}{ll} v_u & \text{if } \lfloor t\hat{}\mathfrak{g}^d\hat{}v_u \rfloor_\sigma = \mathsf{undefined}\\ v_u\hat{}\{(a\hat{}\mathfrak{s}^d), (a\hat{}\mathfrak{g}^d)\} & \text{otherwise} \end{array} \right\}
\end{aligned}
$$

Leaves are the nodes of a tree with no children, i.e., any sequence l such that there is no sequence $l \cdot s$ where s is a single action/label in the tree.

B.6.3 Label-free Game Trees

To get back unlabeled dGL games operational semantics, we erase labels from the labeled semantics.

Definition B.3 (Erasing Labels)**.** The function π_{act} takes in a node or game tree and returns it with all subgame labels removed. First we define erasing labels for a single node. Here, π_{act}

accepts a node s (which is a sequence) and returns a label-free node (the sequence with labels dropped).

$$\pi_{\mathsf{act}}(s) \;\equiv\; (a_i \;:\; a_i \in s,\, a_i \notin \text{subgame labels})$$

Erasing labels for a game tree t proceeds by erasing labels for every node, producing a label-free tree.

$$\pi_{\mathsf{act}}(t) \;\equiv\; \{\pi_{\mathsf{act}}(s) \;:\; s \in t \text{ and } \pi_{\mathsf{act}}(s) \neq ()\}$$

If erasing labels results in an empty sequence (), it is discarded.

Definition B.4 (Label-free Semantics)**.** The label-free game tree of a dGL game $a{:}\alpha$ results from erasing labels in the labeled semantics, i.e., $\pi_{\mathsf{act}}(\mathcal{T}(a{:}a, \sigma))$.

B.6.4 Demonic Subvalue Projection

Definition B.5 (Demonic subvalue projection)**.** The *projection* of Demonic subvalue map S onto dGL game $a{:}\alpha$ with Demon winning condition ϕ, written $a\,{:}\alpha[]\underset{\downarrow}{\exists}S$, is generated recursively per the structure of $a{:}\alpha$ as follows. If $a{:}\alpha$ has structure:

$$a{:}x := \times \text{ then } x := \times\,;\, !\phi.$$

$$a{:}\{x' = f(x)\ \&\ Q\}^d \text{ then } \{x' = f(x)\ \&\ Q\}^d\,;\, !\phi$$

$$a{:}(g{:}\gamma \cap d{:}\delta) \text{ then } (!S(g)\,;\, g{:}\gamma\underset{\downarrow}{\exists}S) \cap (!S(d)\,;\, d{:}\delta\underset{\downarrow}{\exists}S)$$

$$a{:}(g{:}\gamma)^{\times} \text{ then } (!S(g)\,;\, g\,{:}\gamma[]\underset{\downarrow}{\exists}S)^{\times};\, !\phi$$

$$a{:}(g{:}\gamma;\, d{:}\delta) \text{ then } g\,{:}\gamma[]\underset{\downarrow}{\exists}S\,;\, d\,{:}\delta[]\underset{\downarrow}{\exists}S$$

$$a{:}(g{:}\gamma \cup d{:}\delta) \text{ then } g\,{:}\gamma[]\underset{\downarrow}{\exists}S \cup d\,{:}\delta[]\underset{\downarrow}{\exists}S \qquad a{:}(g{:}\gamma)^{*} \text{ then } g\,{:}\gamma[]\underset{\downarrow}{\exists}S^{*}$$

atomic and not controlled by Demon, i.e.,

$$\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x)\ \&\ Q\}^d\},\ \text{then } \alpha$$

B.6.5 Universal Projection of Demonic Subvalue Maps

Definition B.6 (Universal Projection)**.** The *universal projection* of Demonic subvalue map S onto dGL game $a{:}\alpha$ with Demon winning condition ϕ, written $a{:}\alpha[]\underset{\downarrow}{\forall}\,S$, is generated recursively per the structure of $a{:}\alpha$ as follows. If $a{:}\alpha$ has structure:

$$a{:}x := \otimes \text{ then } !(\exists x\, \phi)\,;\, x := *\,;\, ?\phi.$$

$$a{:}\{x' = f(x)\ \&\ Q\}^d, \;\text{ then } !\langle\alpha\rangle\phi\,;\, \{x' = f(x)\ \&\ Q\}\,;\, ?\phi.$$

$$a{:}(g{:}\gamma \cap d{:}\delta),\ \text{then } !(S(g) \vee S(d))\,;\, ((?S(g)\,;\, g{:}\gamma[]\underset{\downarrow}{\forall}\,S) \cup (?S(d)\,;\, d{:}\delta[]\underset{\downarrow}{\forall}\,S))\,.$$

$$a{:}(g{:}\gamma)^{\times},\ \text{then } !(S(g) \vee \phi)\,;\, (?S(g)\,;\, []{:}\gamma\underset{\downarrow}{\forall}\,S\,;\, ?S(a) \vee \phi)^{*}\;;\, ?\phi.$$

$$a{:}(g{:}\gamma;\, d{:}\delta),\ \text{then } g{:}\gamma[]\underset{\downarrow}{\forall}\,S\,;\, d{:}\delta[]\underset{\downarrow}{\forall}\,S.$$

$$a{:}(g{:}\gamma \cap d{:}\delta),\ \text{then } g{:}\gamma[]\underset{\downarrow}{\forall}\,S \cap d{:}\delta[]\underset{\downarrow}{\forall}\,S.$$

$$a{:}(g{:}\gamma)^{\times},\ \text{then } g{:}\gamma[]\underset{\downarrow}{\forall}\,S^{\times}.$$

atomic and not controlled by Demon, i.e.,

$$\alpha \in \{x := e, x := *, ?Q, !Q, \{x' = f(x)\ \&\ Q\}\},\ \text{then } \alpha.$$

B.6.6 Strategy Set Generation

Def. 6.4 shows how an agent's Subvalue map lets it decide what actions to take at a given state. This section shows how to generate a set of strategies (as defined in dGL operational semantics) given a subvalue map by tracing through the game while keeping track of which strategy leads to what state, and using the policy to predict what actions to take at decision points.

Definition B.7 (Transformation to Strategy Set)**.** The function $\mathcal{S}_{a:\alpha}(S)(b,\sigma)$ produces the set of Angel strategies to play game $a{:}\alpha$ following subvalue map S starting at subgame b, where σ represents the current state, where $\hat{}$ is prefix-closed concatenation as shown in Appendix B.6.2. If the structure of b is:

$$b{:}x := e \text{ then } \{\{(x := e)\}\}$$

$$b{:}x := * \text{ then } \{p \ : \ p \in \mathcal{P}_{a:\alpha}(S)(b,\sigma)\}$$

$$b{:}x := \otimes \text{ then } \{\{(x := e^d) \ : \ e \in \mathbb{R}\}\}$$

$$b{:}\{x' = f(x) \ \&\ Q\} \text{ then } \{\{p\} \ : \ \mathcal{P}_{a:\alpha}(S)(b,\sigma)\}$$

$$b{:}\{x' = f(x) \ \&\ Q\}^d \text{ then }$$
$$\{\{(x' = f(x) \& Q^d \mathbin{@} t) \ : \ t \in \mathbb{R}, t \geq 0 \text{ where}$$
$$\varphi(0) = \sigma, \varphi \text{ follows } x' = f(x), \text{ and } \forall s \in [0,t]\, \varphi(s) \models Q\}\}$$

$$b{:}?Q \text{ then } \{\{(?Q)\}\}$$

$$b{:}!Q \text{ then } \{\{(!Q)\}\}$$

$$b{:}(g{:}\gamma \cup d{:}\delta) \text{ then}$$
$$\begin{cases} \{\} & \text{if } \mathcal{P}_{a:\alpha}(S)(b) = \{\} \\ \{\{\mathfrak{l}\,\hat{}\,t\} \ : \ t \in \mathcal{S}_{g:\gamma}(S)(g,\sigma)\} & \text{if } \mathcal{P}_{a:\alpha}(S)(b) = \{\mathfrak{l}\} \\ \{\{\mathfrak{r}\,\hat{}\,t\} \ : \ t \in \mathcal{S}_{d:\delta}(S)(d,\sigma)\} & \text{if } \mathcal{P}_{a:\alpha}(S)(b) = \{\mathfrak{r}\} \\ \{\{\mathfrak{l}\,\hat{}\,t\} \ : \ t \in \mathcal{S}_{g:\gamma}(S)(g,\sigma)\} \cup & \text{otherwise if } \mathcal{P}_{a:\alpha}(S)(b) = \{\mathfrak{l},\mathfrak{r}\} \\ \quad \{\{\mathfrak{r}\,\hat{}\,t\} \ : \ t \in \mathcal{S}_{d,\sigma:\delta}(S)(d)\} & \end{cases}$$

$$b{:}(g{:}\gamma \cap d{:}\delta) \text{ then } \{\mathfrak{l}^d\,\hat{}\,t_l \cup \mathfrak{r}^d\,\hat{}\,t_r \ : \ t_l \in \mathcal{S}_{g:\gamma}(S)(g,\sigma) \text{ and } t_r \in \mathcal{S}_{d:\delta}(S)(d,\sigma)\}$$

$$b{:}(g{:}\gamma \,;\, d{:}\delta) \text{ then } \{t \cup \bigcup_{v \in \mathsf{leaf}(t)} (v\,\hat{}\,u) \ : \ t \in \mathcal{S}_{g:\gamma}(S(\mathsf{end} \mapsto S(d)))(g,\sigma) \text{ and}$$
$$u \in \mathcal{S}_{d:\delta}(S)(d)(\lfloor v \rfloor_\sigma)\}$$

where if $\lfloor v \rfloor_\sigma$ is undefined, then u is empty.

$$b{:}(g{:}\gamma)^* \text{ then } \bigcup_{n<\omega} \{t \ : \ t \in f^n(\mathcal{P}_{a:\alpha}(S)(b,\sigma)) \text{ and } \nexists u(u\,\hat{}\,\mathfrak{g} \in \mathsf{leaf}(t))\}$$

where f^n is n-fold composition of the function :

$$f(Z) = \bigcup_{t \in Z} \{t \cup$$
$$\bigcup_{u\,\hat{}\,\mathfrak{g} \in \mathsf{leaf}(t)} \left\{ \begin{array}{ll} \{u\,\hat{}\,\mathfrak{g}\,\hat{}\,v_u\} & \text{if } \lfloor u\,\hat{}\,\mathfrak{g}\,\hat{}\,v_u \rfloor_\sigma = \mathsf{undefined} \\ \{u\,\hat{}\,\mathfrak{g}\,\hat{}\,v_u\,\hat{}\,o_{uv}\} & \text{otherwise} \end{array} \right\} :$$

$$\begin{aligned}
&v_u \in \mathcal{S}_{g:\gamma}(S(\mathsf{end} \mapsto S(a)))(g)(\lfloor u \rfloor_\sigma) \text{ and}\\
&o_{uv} \in \mathcal{P}_{a:\alpha}(S)(b, \lfloor u\,\hat{}\,\mathfrak{g}\,\hat{}\,v_u \rfloor_\sigma)\}
\end{aligned}$$

$$b{:}(g{:}\gamma)^\times \text{ then } \max_{\subseteq}\{\bigcup C \;:\; C \subseteq \bigcup_{n<\omega} f^n(\{\{\mathfrak{s}^d, \mathfrak{g}^d\}\}) \text{ is a nonempty } \subseteq\text{-chain}\}$$

where f^n is n-fold composition of the function :

$$f(Z) = \bigcup_{t \in Z}\{t \cup \bigcup_{\substack{u\,\hat{}\,\mathfrak{g}^d \in \mathsf{leaf}(t),\\ \lfloor u \rfloor_\sigma \text{ defined}}}$$

$$\left\{\begin{array}{ll}\{u\,\hat{}\,\mathfrak{g}^d\,\hat{}\,v\,\hat{}\,\{\mathfrak{s}^d, \mathfrak{g}^d\} \;:\; v \in \mathcal{S}_{g:\gamma}(S_a)(g)(\lfloor u \rfloor_\sigma)\} & \lfloor u\,\hat{}\,\mathfrak{g}^d\,\hat{}\,v \rfloor_\sigma \text{ defined}\\ \{u\,\hat{}\,\mathfrak{g}^d\,\hat{}\,v \;:\; v \in \mathcal{S}_{g:\gamma}(S_a)(g)(\lfloor u \rfloor_\sigma)\} & \lfloor u\,\hat{}\,\mathfrak{g}^d\,\hat{}\,v \rfloor_\sigma \text{ undefined}\end{array}\right.\}$$

where $S_a = S(\mathsf{end} \mapsto S(a))$

with implicit closure under prefixes.

To reason only about the states reachable by playing the game trees, we should introduce a version of the game trees free of actions that do nothing.

Definition B.8 (Erasing skip Actions)**.** Let the set of skip actions $\mathfrak{S}$ be

$$\mathfrak{S} = \{\mathfrak{g}, \mathfrak{s}, \mathfrak{l}, \mathfrak{r}, \mathfrak{c}, \mathfrak{g}^d, \mathfrak{s}^d, \mathfrak{l}^d, \mathfrak{r}^d\}$$

. The function $\pi_{\neg\mathfrak{S}}$ takes in a node or game tree and returns it with all labels and skip actions removed. First we define $\pi_{\neg\mathfrak{S}}$ for a single node. Here, $\pi_{\neg\mathfrak{S}}$ accepts a node s (which is a sequence) and returns a node (the sequence with labels dropped).

$$\pi_{\neg\mathfrak{S}}(s) \;\equiv\; (a_i \;:\; a_i \in s,\, a_i \notin \text{subgame labels and } a_i \notin \mathfrak{S})$$

Erasing labels and skip actions for a game tree t proceeds by erasing these for every node, producing a new tree.

$$\pi_{\neg\mathfrak{S}}(t) \;\equiv\; \{\pi_{\neg\mathfrak{S}}(s) \;:\; s \in t \text{ and } \pi_{\neg\mathfrak{S}}(s) \neq ()\}.$$

B.6.7 Inductive Demonic Subvalue Maps

Definition B.9 (Inductive Demonic subvalue maps)**.** Let S be a map from the subgames of $a{:}\alpha$ to winning subregions. S is an *inductive Demonic subvalue map* for game $a{:}\alpha$, written $a{:}\alpha[]\models S$, when the following holds. If $a{:}\alpha$ has structure:

$$\begin{aligned}
&\text{atomic, i.e., } \alpha \in \{x := e, x := *, x := \otimes, ?Q, !Q, \{x' = f(x) \,\&\, Q\},\\
&\qquad\qquad \{x' = f(x) \,\&\, Q\}^d\} \text{ then } \models S(a) \to [\alpha]S(\mathsf{end}).\\
&a{:}(g{:}\gamma \cup d{:}\delta) \text{ then } \models S(a) \to S(g) \wedge S(d) \text{ and } g{:}\gamma[]\models S \text{ and } d{:}\delta[]\models S.\\
&a{:}(g{:}\gamma \cap d{:}\delta) \text{ then } \models S(a) \to S(g) \vee S(d) \text{ and } g{:}\gamma[]\models S \text{ and } d{:}\delta[]\models S.\\
&a{:}(g{:}\gamma\,;\, d{:}\delta) \text{ then } \models S(a) \to S(g) \text{ and } g{:}\gamma[]\models S(\mathsf{end} \mapsto S(d)) \text{ and } d{:}\delta[]\models S.\\
&a{:}(g{:}\gamma)^* \text{ then } \models S(a) \to S(\mathsf{end}) \wedge S(g) \text{ and } g{:}\gamma[]\models S(\mathsf{end} \mapsto S(a)).\\
&a{:}(g{:}\gamma)^\times \text{ then } \models S(a) \to [a\,{:}\alpha[]\underset{\downarrow}{\exists} S]S(\mathsf{end}) \text{ and } g{:}\gamma[]\models S(\mathsf{end} \mapsto S(a)).
\end{aligned}$$

B.6.8 Game Prefix

This definition uses *the empty game* skip which can be treated like the game $?true$ in terms of effect on plays, except with no corresponding action in the operational semantics.

Definition B.10 (Game Prefix)**.** The game suffix $a{:}\alpha_{:b}$ of subgame $b{:}\beta$ in $\mathsf{subgames}(a{:}\alpha)$ is constructed as below. If $a{:}\alpha$ is $b{:}\beta$ and α is not a loop, then $a{:}\alpha_{:b}$ is the empty game, skip. If $b = a$ and $a{:}\alpha$ is a loop, then $a{:}\alpha_{:b}$ is $a{:}\alpha$. If b is end then $a{:}\alpha_{:b}$ is the entire game $a{:}\alpha$. Otherwise, if $a{:}\alpha$ has structure:

$$a{:}(g{:}\gamma)^* \text{ or } a{:}(g{:}\gamma)^\times \text{ then } a{:}\alpha_{:b} = a{:}\alpha\,;\,(g{:}\gamma_{:b})$$

$$a{:}(g{:}\gamma \cup d{:}\delta) \text{ or } a{:}(g{:}\gamma \cap d{:}\delta) \text{ then } \begin{cases} a{:}\alpha_{:b} = g{:}\gamma_{:b} & b \in \mathsf{subgames}(g{:}\gamma) \\ a{:}\alpha_{:b} = d{:}\delta_{:b} & \text{otherwise} \end{cases}$$

$$a{:}(g{:}\gamma\,;\,d{:}\delta) \text{ then } \begin{cases} a{:}\alpha_{:b} = g{:}\gamma_{:b} & b \in \mathsf{subgames}(g{:}\gamma) \\ a{:}\alpha_{:b} = g{:}\gamma\,;\,d{:}\delta_{:b} & \text{otherwise} \end{cases}$$

B.6.9 Solving Function for Inductive Demonic Subvalue Maps

$[]\mathsf{map}(a{:}\alpha, \phi)$, defined below, computes an inductive Demonic subvalue map for game $a{:}\alpha$ and Demon winning condition ϕ. Operator $\uplus$ denotes the disjoint union of two subvalue maps.

$$[]\mathsf{map}(a{:}\alpha, \phi) := \{a \mapsto \mathsf{simpl}([\alpha]\phi), \mathsf{end} \mapsto \phi\} \text{ where } \alpha \in \{x := e, x := *, x := \otimes,$$
$$?Q, !Q, \{x' = f(x) \& Q\}, \{x' = f(x) \& Q\}^d\}$$

$$[]\mathsf{map}(a{:}(g{:}\gamma \cap d{:}\delta), \phi) := S_1 \uplus S_2 \uplus \{a \mapsto S_1(g) \vee S_2(d)\}$$
$$\text{where } S_1 := []\mathsf{map}(g{:}\gamma, \phi),\ S_2 := []\mathsf{map}(d{:}\delta, \phi) \setminus \mathsf{end}$$

$$[]\mathsf{map}(a{:}(g{:}\gamma \cup d{:}\delta), \phi) := S_1 \uplus S_2 \uplus \{a \mapsto S_1(g) \wedge S_2(d)\}$$
$$\text{where } S_1 := []\mathsf{map}(g{:}\gamma, \phi),\ S_2 := []\mathsf{map}(d{:}\delta, \phi) \setminus \mathsf{end}$$

$$[]\mathsf{map}(a{:}(g{:}\gamma; d{:}\delta), \phi) := S_1 \uplus S_2 \uplus \{a \mapsto S_2(g)\}$$
$$\text{where } S_1 := []\mathsf{map}(d{:}\delta, \phi),\ S_2 := []\mathsf{map}(g{:}\gamma, S_1(d)) \setminus \mathsf{end}$$

$$[]\mathsf{map}(a{:}(g{:}\gamma)^\times, \phi) := S \uplus \{a \mapsto \mathsf{Inv}, \mathsf{end} \mapsto \phi\} \text{ if } \langle a{:}\alpha \mathbin{\underset{\downarrow}{⨟}} S\rangle\phi$$
$$\text{where } S := []\mathsf{map}(g{:}\gamma, \mathsf{Inv} \vee \phi) \setminus \mathsf{end}$$

$$[]\mathsf{map}(a{:}(g{:}\gamma)^*, \phi) := S \uplus \{a \mapsto \mathsf{Inv}, \mathsf{end} \mapsto \phi\} \text{ if } \models \mathsf{Inv} \to S(g) \wedge \phi$$
$$\text{where } S := []\mathsf{map}(g{:}\gamma, \mathsf{Inv}) \setminus \mathsf{end}$$

Appendix C

Automatic Verification and Synthesis (Appendix)

C.1 Tactic for Model 6

The tactic that the LLM discovers to prove Model 6 is as follows:

```
unfold;
loop("x <= g/d & y <= a/b & g/d >= xmin &
    a/b >= ymin & x > 0 & y > 0 & a > 0 &
    b > 0 & d > 0 & g > 0 & x >= xmin &
    y >= ymin", 1);
<(
  QE("Z3"),
  unfold;
  existsR("g/d-x", 1);
  unfold;
  <(
    QE("Z3"),
    existsR("a/b-y", 1);
    unfold;
    <(
      QE("Z3"),
      unfold;
      print("State before ODE cut after
              choosing xadd and yadd");
      dC("((d*x-g)^2+(b*y-a)^2)=0", 1);
      <(
        dW(1); QE("Z3"),
        dbx("2*(d*x-g)*(b*y-a)*(b*y-d*x)/
              ((d*x-g)^2+(b*y-a)^2)", 1)
      )
```

```
        )
    ),
    QE("Z3")
)
```

Here, the print statements help the LLM to track progress and get information about the proof state in case of failure. The use of `QE("Z3")` indicates that the Z3 SMT solver is used to discharge the corresponding proof obligation. In the prompt, the LLM is encouraged to break down the proof rather than using auto so that the proof remains stable despite changes to the automation and quicker to check.

C.2 Other Case Studies

This sections lists the case studies used in the LLM-powered verification chapter (Chapter 4) and synthesis chapter (Chapter 7).

Case Study: Train.

A train must stop before its *end of motion authority* e, beyond which may lie other trains. The train can brake using two mechanisms: a normal brake that decelerates at most at rate b_0, and an air brake that ramps up deceleration gradually, as pressure leaks from brake pipes, at rate p until it reaches maximum deceleration A_{pb}. This case study is inspired by a previous case study on verification of trains with air brakes [101], but with the addition of rolling resistance and adversarial game dynamics.

Model 23 Explanation. Model 23 shows the dGL formula modeling the train control problem via a game between Angel (the train controller) and Demon (environmental uncertainties). Angel chooses between the control modes of normal operation and air brake application (Line 8). In normal operation, she may choose any acceleration between $-b_0$ and a_0. Then, Demon chooses how long the train will run before Angel gets to re-evaluate her decision, bounded by a maximum time latency T (Line 13 and 14). The physical dynamics operate in two modes, depending on whether the air brakes are ramping up (Line 13) or have stabilized at their maximum (Line 14). This is modeled by ODEs for each mode within an inner loop that allows switching between them. Demon controls the number of iterations of the outer control loop (Line 6 to 14), forcing Angel to maintain safety for arbitrarily long. The overall modal formula of form $\langle\alpha\rangle\phi$ is true in states where Angel has a strategy to play so that regardless of how Demon plays, she will win, that is, the train controller has a way to maintain safety (Line 15). To reason correctly about this problem we must again reason over the nonlinear dynamics of the train, and ensure safety over unbounded time horizons. The mode switching inner-loop structure adds a further layer of complexity.

Model 23 Case study: Train Control. Gray parts are not included in the synthesis variant of the problem, and must be computed. Brown text is not a part of the verification problem, but added to the synthesis problem.

assume

$$\begin{array}{l} e - \mathtt{trainPos} - \dfrac{\mathtt{vel}^2}{2 \cdot b_0} > 0 \land \\ \quad a_0 > 0 \land b_0 > 0 \land a_1 < 0 \land \\ \quad a_2 < 0 \land T > 0 \land \\ \quad A_{pb} < 0 \land p < 0 \land \mathtt{vel} \geq 0 \\ \quad \land m_b = 0 \land \mathtt{airBrake} = 0 \rightarrow \end{array}$$

ctrl

$$\begin{array}{l} \langle (\, t := 0\,; \\ \quad (\,(\mathtt{trainAcc} := *; \\ \qquad ?-b_0 \leq \mathtt{trainAcc} < a_0\,; \\ \qquad m_b := 0\,; \\ \qquad \mathtt{airBrake} := 0) \cup \\ \qquad (\mathtt{trainAcc} := -b_0\,; \\ \qquad m_b := p)\,)^d\,; \end{array}$$

plant

$$\left(\left\{ \begin{array}{l} \mathtt{trainPos}' = \mathtt{vel}, \\ \mathtt{vel}' = \mathtt{trainAcc} + \\ \quad a_1 \cdot \mathtt{vel} + a_2 \cdot \mathtt{vel}^2 \\ \quad + \mathtt{airBrake}, \\ \mathtt{airBrake}' = m_b, \\ t' = 1 \,\&\, t \leq T \\ \quad \land \mathtt{vel} \geq 0 \\ \quad \land \mathtt{airBrake} \geq A_{pb} \end{array} \right\}^d \cap \left\{ \begin{array}{l} \mathtt{trainPos}' = \mathtt{vel}, \\ \mathtt{vel}' = \mathtt{trainAcc} + \\ \quad a_1 \cdot \mathtt{vel} + a_2 \cdot \mathtt{vel}^2 \\ \quad + \mathtt{airBrake}, \\ \mathtt{airBrake}' = 0, \\ t' = 1 \,\&\, t \leq T \\ \quad \land \mathtt{vel} \geq 0 \\ \quad \land \mathtt{airBrake} \leq A_{pb} \end{array} \right\}^d \right)^\times$$

safe

$$)^\times \rangle (e - \mathtt{trainPos} > 0)$$

Model 24 Case study: Chemical Reaction Control. Gray parts are not included in the synthesis variant of the problem, and must be computed.

```
assume  1  $k_A > 0 \land k_B > 0 \land k_C > 0 \land k_T \geq 0 \land$
        2      $T > 0 \land \mathtt{A} \geq 0 \land \mathtt{B} \geq 0 \land \mathtt{C} \geq 0$
        3      $\land \mathtt{Temp} \geq 0 \land (k_B \cdot \mathtt{A} \leq k_A \cdot \mathtt{B} \land$
        4      $k_A \cdot \mathtt{Temp} + k_T \cdot \mathtt{A} \leq k_A \cdot T_{max}$
        5      $\lor k_A \cdot \mathtt{B} \leq k_B \cdot \mathtt{A} \land$
        6      $k_B \cdot \mathtt{Temp} + k_T \cdot \mathtt{B} \leq k_B \cdot T_{max}) \rightarrow$
ctrl    7  $\langle ( ( \mathtt{isOn} := 0 \cap \mathtt{isOn} := 1 ) ; ø := 0 ;$
plant   8      $\left\{ \begin{array}{l} \mathtt{A}' = -\mathtt{A} \cdot \mathtt{B} \cdot \mathtt{Temp} \cdot \mathtt{isOn} \cdot k_A, \\ \mathtt{B}' = -\mathtt{A} \cdot \mathtt{B} \cdot \mathtt{Temp} \cdot \mathtt{isOn} \cdot k_B, \\ \mathtt{C}' = \mathtt{A} \cdot \mathtt{B} \cdot \mathtt{Temp} \cdot \mathtt{isOn} \cdot k_C, \\ \mathtt{Temp}' = \mathtt{A} \cdot \mathtt{B} \cdot \mathtt{Temp} \cdot \mathtt{isOn} \cdot k_T, \\ ø' = 1 \,\&\, ø \leq T \land \mathtt{A} \geq 0 \land \mathtt{B} \geq 0 \\ \land \mathtt{C} \geq 0 \land \mathtt{Temp} \geq 0 \end{array} \right\}^d$
safe    9    $)^\times \rangle (\mathtt{Temp} \leq T_{max})$
```

Case Study: Chemical Reaction.

An exothermic chemical reaction $A + B \rightarrow C$ is taking place. The reaction rates depend on the temperature of the reactor `Temp`, which increases in proportion to the reaction rate. The reaction rate also depends on the concentrations of the reactants A and B. The safety requirement is that the temperature never exceeds a maximum safe temperature T_{max}. A time-triggered controller can turn the reaction on and off, getting to revise its decision in at most every T time units. However, we do not have direct access to the controller, and instead conservatively safeguard the system by not putting in too much reactant to begin with. The verification challenge is to check whether regardless of what the controller does, safety is always ensured. The physical reactor setup in this case study is similar to one from the literature [19], but with an important change in setup: the controller is not directly accessible, and we want to verify safety for all possible controller actions.

Model 24 shows the full control formula. We make some initial assumptions about the quantities of reactant and signs of the rate constants at Line 1 and Line 2. Then Demon chooses whether to turn the reaction on or off at Line 7 in a time-triggered control loop, also controlled by Demon. In the end, Angel wins if the temperature never exceeds T_{max} (Line 9). This system again requires reasoning about complex dynamics over different, arbitrarily alternating control modes over unbounded iterations. The solution to the dynamics lies outside the decidable (polynomial) fragment of real arithmetic, requiring non-trivial deductive reasoning about differential equations via invariants.

Model 25 Case study: Coolant System. ϕ_{init} is a long formula listed in Fig. C.1. It is not provided in the synthesis variant of the problem, and must be computed.

```
assume  1  $\phi_{\text{init}} \land T > 0 \land F > 0 \land a_{min} > 0 \land$
        2      $d_{max} > 0 \land c_1 > 0 \land c_2 > 0 \rightarrow$
ctrl    3  $\langle((((?\mathtt{absbd} < a^d_{min} \,;\, \mathtt{flow} := F) \cap$
        4      $(\mathtt{absbd} \geq a^d_{min} \,;\, \mathtt{flow} := 0))\,;$
plant   5    $\emptyset := 0\,;$
        6    $\left\{\begin{array}{l} \mathtt{absbd}' = \mathtt{flow} \cdot c_1 \cdot \\ \quad \mathtt{temp\_diff}, \\ \mathtt{disch}' = \mathtt{flow}, \\ \mathtt{temp\_diff}' = -\mathtt{flow} \cdot \\ \quad c_2 \cdot \mathtt{temp\_diff} + g, \\ \emptyset' = 1, \\ \mathtt{deadline}' = -1 \,\&\, \emptyset \leq T \end{array}\right\}^d$
safe    7    $)^\times\rangle(\mathtt{disch} \leq d_{max} \land$
        8       $(\mathtt{deadline} \geq 0 \lor \mathtt{absbd} \geq a_{min}))$
```

Case Study: Coolant

Consider a coolant system in a nuclear power plant that must achieve the control objective of absorbing *at least* a_{min} units of heat while expending *at most* d_{max} units of coolant in a given time frame. The coolant system controller chooses how much coolant to send through the system, and then goes to sleep for some amount of time bounded by maximum latency T, repeatedly polling in a control loop. Model 25 shows the formula modeling this system, that we must prove valid. We must reason directly in terms of symbolic parameters such as required absorption a_{min} and maximum flow rate F. The system has different control modes, can run for arbitrarily long (because of loop iterations), and dynamical behavior qualitatively changes depending on the initial temperature, all of which our reasoning will precisely account for.

Model 25 Explanation. We provide an explanation of the dGL model for this case study. Lines 1 and 2 show initial assumptions expressing physical constraints, e.g., that the maximum flow rate F is positive. Formula ϕ_{init} is a complex formula listed in Fig. C.1 representing initial conditions that make safe control possible: the discharge limit should not already have been exceeded, and if the heat absorption quota is not already met, then there is enough time and discharge capacity left to still meet it. At lines 3 and 4, the control loop begins.

The controller assigns a coolant flow rate. The choice $\mathtt{flow} := F$ is chosen in states where the absorption quota has not yet been met ($\mathtt{absbd} < a_{min}$), and the choice $\mathtt{flow} := 0$ is chosen otherwise. At Line 6, the controller goes to sleep and the state evolves per the physical dynamics, where the heat absorbed is proportional to temperature difference between the coolant and the

$$
\begin{array}{l}
\phi_{\mathsf{inv}} \equiv \overbrace{\texttt{disch} \le d_{max}}^{\text{discharge within limit}} \wedge \overbrace{F > 0 \wedge c_1 > 0 \wedge c_2 > 0 \wedge T > 0}^{\text{physics constraints}} \wedge (\\
\quad \overbrace{\texttt{absbd} \ge a_{min}}^{\text{already absorbed enough}} \vee (\overbrace{\texttt{deadline} \ge 0}^{\text{there's time left}} \wedge (\\
\begin{array}{l|l}
\begin{array}{l}\text{Current}\\ \texttt{temp\_diff}\\ \text{lower}\\ \text{bounds}\\ \text{all future}\\ \texttt{temp\_diff}\end{array} &
\begin{array}{l}
(\overbrace{\texttt{temp\_diff} \le g/(F \cdot c_2)}^{\texttt{temp\_diff}\text{ only increases}} \wedge \overbrace{c_1 \cdot \texttt{temp\_diff} \cdot (d_{max} -}^{\text{can absorb enough}\cdots}\\
\overbrace{\texttt{disch} - F \cdot T) \ge a_{min} - \texttt{absbd}}^{\cdots\text{ without exceeding discharge budget}} \wedge\\
\overbrace{\texttt{deadline} \cdot c_1 \cdot \texttt{temp\_diff} \cdot F \ge a_{min} - \texttt{absbd}}^{\text{can absorb enough before deadline}}) \vee
\end{array}\\
\\
\begin{array}{l}\dfrac{g}{F \cdot c_2}\\ \text{lower}\\ \text{bounds}\\ \texttt{temp\_diff}\end{array} &
\begin{array}{l}
(\overbrace{\texttt{temp\_diff} > g/(F \cdot c_2)}^{\texttt{temp\_diff}\text{ bound}} \wedge \overbrace{c_1 \cdot (g/(F \cdot c_2))\cdot}^{\text{can absorb enough}\cdots}\\
\overbrace{(d_{max} - \texttt{disch} - F \cdot T) \ge a_{min} - \texttt{absbd}}^{\cdots\text{ without exceeding discharge budget}} \wedge\\
\overbrace{\texttt{deadline} \cdot c_1 \cdot g/c_2 \ge a_{min} - \texttt{absbd}}^{\text{can absorb enough before deadline}}))))
\end{array}
\end{array}
\end{array}
\tag{C.1}
$$

Figure C.1: Formula ϕ_{init} for the Coolant case study.

system, total coolant discharged is tracked by variable `disch`, the system generates heat at rate g, and the auxiliary `deadline` variable counts down to when the system must have met its absorption obligations. The controller stays asleep for at most T time units, though it may wake up earlier, with variable ø tracking time elapsed. Finally, at lines 7 and 8, the control loop closes, with the syntax $(\alpha)^{\times}$. The control loop repeats arbitrarily many times, and we must verify that all possible numbers of iterations are safe. The modal formula $\langle\alpha\rangle\phi$ is true when *all* possible runs result in a state satisfying postcondition ϕ. Therefore, the overall formula of Model 25 is valid when, given the assumptions of lines 1 and 2, all executions lead to a state where the control objective is satisfied, that is, $\texttt{disch} \le d_{max} \wedge (\texttt{deadline} \le 0 \to \texttt{absbd} \ge a_{min})$ holds. This case study is an example of a hybrid *system*, since there is no adversarial game play.

Case Study: Van der Pol.

A pair of coupled Van der Pol oscillators must be set so that the sum of their positions `x1` and `x2` never exceeds a given bound x_{max}. The system dynamics are governed by parameters m and b. The oscillators can run arbitrarily long, and safety must be ensured throughout. This case study requires reasoning about nonlinear dynamics over unbounded time horizons. Model 26 shows the full control formula. Angel sets initial conditions at Line 2 and Line 3, then Demon chooses how long the oscillators run at Line 4, after which safety is checked at Line 5. Then Demon chooses how long to run the oscillators at Line 4, and whether to repeatedly further run them in the loop at Line 5. Angel wins in the end if the sum of the positions never exceeds x_{max} (Line 5). This

Model 26 Case study: Coupled Van der Pol Oscillators. Gray parts are not included in the synthesis variant of the problem, and must be computed.

```
assume  $x_{max} > 0 \rightarrow$
ctrl    $\langle \mathtt{x1} := *;\ \mathtt{x2} := *;\ \mathtt{y1} := *;\ \mathtt{y2} := *;$
          $?m > 0 \wedge b > 0^d;\ ($
plant       $\left\{ \begin{array}{l} \mathtt{x1}' = \mathtt{y1}, \\ \mathtt{y1}' = m \cdot (1 - \mathtt{x1}^2) \cdot \mathtt{y1} \\ \qquad + b \cdot (\mathtt{x2} - \mathtt{x1}) - \mathtt{x1}, \\ \mathtt{x2}' = \mathtt{y2}, \\ \mathtt{y2}' = m \cdot (1 - \mathtt{x2}^2) \cdot \mathtt{y2} \\ \qquad - b \cdot (\mathtt{x2} - \mathtt{x1}) - \mathtt{x2} \end{array} \right\}^d$
safe      $)^\times \rangle (\mathtt{x1} + \mathtt{x2} \leq x_{max})$
```

case study is inspired by one appearing in ARCH-COMP [55], but is modified to add adversarial game dynamics and a loop structure that makes formal reasoning more challenging.

Note In the synthesis variants of the problems, assumptions are written not in the form

$$\text{assumptions} \rightarrow \langle \alpha \rangle \text{postcondition},$$

but rather as

$$\langle ?\text{assumptions (excluding gray parts)}^d\ ;\ \alpha \rangle \text{postcondition}.$$

This does not make a semantic difference, but allows subvalue map computation to account for these assumptions as it passes over the assumptions subgame, and produce a subvalue for the overall game that does not redundantly include the assumptions.

C.2.1 Synthesis Guidelines

In the *synthesis* variants of the problems, the following guidelines are used:

- For *Lotka-Volterra*: Set the system to its equilibrium point as soon as possible, and assume what you must so that the equilibrium point is good enough to maintain the post condition.
- For *train*: Find the control solution for when the train can stop by braking continuously, even conservatively ignoring rolling resistance and air brakes.
- For *chemical reaction*: Find the control solution for the situation where the reactant runs out before temperature can become too large.
- For *coolant*: Find the solution where initially enough heat has not been absorbed, but over time, enough is absorbed before the timer requirement or discharge capacity can be exceeded.

- For *Van der Pol*: Keep the two oscillators in antiphase, maintaining the postcondition this way.

C.3 Verification Prompts

This section lists all the prompts used for the verification case studies. The prompts use the OpenAI api format with system and user queries, and sometimes include few-shot prompts. Here we have shown the queries for the chemical reaction case study as an example. Some Unicode characters have been replaced with LaTeX commands and line breaks have been added for compilation. Prompts are listed in the order in which they are called in the verification progress, and have a one-to-one correspondence with the LLM queries depicted in Fig. 4.2.
1. AnalyzeGame is the first query in the pipeline that produces an analysis of the game used as context in later prompts. One example is shown as a few-shot example.

```
role: system
content: "Your task is to analyze a game and identify which
actions are taken by the Angel (controllable player) versus
the Demon (adversarial player).

## Background Information

In differential game logic (dGL), there are two players:
- **Angel**: The controllable player who makes choices we can
  control. Angel constructs include:
  - Angel choices: `a ++ b` (we choose between a and b)
  - Angel loops: `{a}*` (we choose how many iterations)
  - Angel tests: `?P` (we can proceed if P holds, otherwise we
    immediately lose)
  - Angel ODEs: `{x'=f(x) & Q}` (we control the duration)
  - Angel nondeterministic assignments: `x := *` (we choose the
    value)

There is the dual operator `^@` that switches Angel and Demon
roles in a game construct.
Duals can be nested arbitrarily, so carefully check nesting
levels to determine which player controls each action.
Being within an odd number of `^@` operators indicates Demon
control, while an even number (including zero) indicates Angel
control.
For example, in the game `{(a ++ b)^@; (c++d)}^@`, the choice
between `a` and `b` is controlled by Angel, being within two
`^@` operators, while the choice between `c` and `d` is
controlled by Demon.
Using this notation, we go over the Demon constructs below,
```

```
indicating that they are Demon-controlled by wrapping them in a
single `^@` operator.

- **Demon**: The adversarial player who makes choices we cannot
  control. Demon has all the same constructs as Angel.
  - Demon choices: `{a ++ b}^@` (adversary chooses between a and
    b)
  - Demon loops: `{a*}^@`
     (adversary chooses how many iterations)
  - Demon tests: `{?P}^@` (adversary can proceed if P holds, and
    immediately loses otherwise. So Angel can *assume* in
    subsequent game play that P held at this point; otherwise
    she would already have won.)
  - Demon ODEs: `{x’=f(x) & Q}^@` (adversary controls the
    duration)
  - Demon nondeterministic assignments: `{x := *}^@` (adversary
    chooses the value)

Additionally, the non-controlled operations are:
- Assignment: `x := e` (deterministic assignment like in regular
  programs)
- Sequential composition: `a; b` (first play a, then b)

## Notes
- ODEs can sometimes introduce unmatched single quotes, so
  account for that when parsing.
- In syntax such as { a++b; x:=* }*^@,
  the `^@` applies to the entire loop including its body.
  In this example, Demon controls the choice within the loop
  between a and b, as well as the assignment to x,
  because both are inside one `^@`.

Some subgames are labeled with unique identifiers with the
following syntax:
    <subgame_id>: <subgame>

These subgame ids will help make your answers more specific.

## Input

You will be provided a dGL game.

## Output
```

```
Your analysis should identify:
1. What actions or decisions are under Angel’s control in the
   game.
2. What actions or decisions are under Demon’s control in the
   game.
3. The control modes in the game, that is, regions of different
   qualitative behavior (e.g. growth vs. oscillation vs. decay
   vs. equilibrium).
The modes are typically defined by two things: (1) initial
conditions inducing different qualitative behavior in the
dynamics, and (2) control actions leading to different possible
values of variables. Comment about the asymptotes or polynomial
invariants of each mode, when relevant.
4. The overall control pattern in the game.

## Further Instructions

Provide your analysis as clear, readable text describing the
Angel and Demon actions, and control modes. Be specific about
what choices each player makes and how they might interact.
Structure it into the sections corresponding to the points
above.
Be as concise as possible while still being clear, with no
superfluous or filler text or hedging.

- Carefully consider the differential equation. There can be
  coupling between variables that affects your reasoning.
- Based on the dynamics, you may some times need to come up with
  upper/lower bounds of quantities and use that to derive sound
  preconditions. It’s best to use simplistic, polynomial bounds
  so that they verify more easily.

You will now be provided with 0 or more examples of games and
their analysis, followed by a new game that you must analyze."
- role: user
  content: |-
    Game to analyze:
    {? x >= 0 & g > 0 & air > 0 & parachute > air &
      T > 0 & m > 0;^@};
    {{subgame_c: {{{{{r := air;} ++ {r := parachute;}}^@};
    {t := 0;}}}; {subgame_j:
      {x’ = v, v’ = r * v ^ 2 - g, t’ = 1 &
       x >= 0 & v < 0 & t <= T}
    }}*}^@}
```

```
  Please analyze this game and identify the requested
  items (Angel vs. Demon control actions, control modes,
  overall control pattern).
- role: assistant
 answer:
  mode: null
  content: |
   1) Angel-controlled actions
   - Choice {{{r := air;} ++ {r := parachute;}}^@
    | Angel chooses drag coefficient r each iteration,
    either r := air (low
    drag) or r := parachute (high drag).

   2) Demon-controlled actions
   - Initial test
    {? x >= 0 & g > 0 & air > 0 & parachute > air &
     T > 0 & m > 0}^@ | Demon test. Angel can assume these
    preconditions hold initially; otherwise, play won’t
    proceed.
   - Looping subgame_c
    | Demon chooses how many iterations of subgame_c
    to perform (including possibly zero; finite but
    unbounded).
    He chooses whether to repeat for one more cycle
    or stop at the end of each iteration.
   - ODE duration: subgame_j
    - Demon controls how long to flow
    along the ODE x’ = v, v’ = r v^2 - g, t’ = 1
    within the evolution domain x >= 0, v < 0, t <= T.
    He may stop at any time before the domain boundary
    is reached, including zero time; he cannot continue
    past any boundary (for example x cannot go below 0).
    Since t is reset to 0 at the start of each iteration,
    the duration is always capped by T
    (though the other boundary conditions may force
     an earlier stop).

   3) Control modes (qualitative behaviors)
   - Mode AIR HIGH VEL
    (Angel chose r = air, low drag,
     v<-(g/air)^(1/2)):
    - air * v^2 > g, so v’ > 0.
      Velocity increases (becomes less negative)
```

```
    over time, approaching terminal velocity
    -((g / air)^(1/2)) from below. Will not cross it.
- Mode PARACHUTE HIGH VEL
  (Angel chose r = parachute, high drag,
   v<-(g/parachute)^(1/2)):
  - parachute * v^2 > g, so v' > 0.
    Velocity increases (becomes less negative)
    over time, approaching terminal velocity
    -((g / parachute)^(1/2)) from below,
    which is a higher (less negative) terminal
    velocity than in AIR mode.
- Mode AIR EQU VEL
  (Angel chose r = air, low drag,
   and v is terminal velocity -((g / air)^(1/2))):
  - v' = air * v^2 - g = 0,
    so velocity remains constant at terminal velocity.
- Mode PARACHUTE EQU VEL
  (Angel chose r = parachute, high drag,
   and v is terminal velocity -((g / parachute)^(1/2))):
  - v' = parachute * v^2 - g = 0,
    so velocity remains constant at terminal velocity,
    slower than in AIR mode.
- Mode AIR LOW VEL
  (Angel chose r = air, low drag,
   and 0 > v > -((g / air)^(1/2))):
  - air * v^2 < g, so v' < 0.
    Velocity decreases (becomes more negative)
    over time. It approaches terminal velocity
    -((g / air)^(1/2)) from above but will not cross it.
- Mode PARACHUTE LOW VEL
  (Angel chose r = parachute, high drag,
   and 0 > v > -((g / parachute)^(1/2))):
  - parachute * v^2 < g, so v' < 0.
    Velocity decreases (becomes more negative)
    over time, approaching terminal velocity
    -((g / parachute)^(1/2)) from above,
    but will not cross it.

4) Overall control pattern
- Some global preconditions hold initially (x >= 0, g > 0,
  air > 0, parachute > air, T > 0, m > 0).
- Per iteration:
  1) Demon decides whether to continue for another
     iteration or stop (Demon loop).
```

```
        2) Angel picks drag coefficient r \in {air, parachute}
           (Angel choice).
        3) Reset timer t := 0 (deterministic assignment).
        4) Demon chooses ODE flow duration
           within domain (Demon ODE).
      - Across iterations: Demon decides how many iterations to
        perform and length of evolution per iteration.
        Angel's control lever is her binary choice of drag
        coefficient.
- role: user
  content: |-
    Game to analyze:
    (kA > 0 & kB > 0 & kC > 0 & kT >= 0 & T > 0 &
    A >= 0 & B >= 0 & C >= 0 & Temp >= 0 &
    (kB*A <= kA*B & kA*Temp + kT*A <= kA*Tmax |
     kA*B <= kB*A & kB*Temp + kT*B <= kB*Tmax)) ->
    < { { { isOn:=0; ++ isOn:=1; }; t:=0;
        { A' = - A*B*Temp*isOn*kA,
          B' = - A*B*Temp*isOn*kB,
          C' = A*B*Temp*isOn*kC,
          Temp' = A*B*Temp*isOn*kT,
          t' = 1 & t<=T & A>=0 & B>=0 & C>=0 & Temp>=0
        };
      }* }^@ > Temp<=Tmax

    Please analyze this game and identify the requested
    items
    (Angel vs. Demon control actions, control modes, overall
    control pattern).
```

2. GetTactic is the second prompt in the pipeline (first in the proof loop). It gets a Bellerophon proof tactic from the LLM.

```
- role: system
  content: |
    You are trying to prove a differential game logic (dGL)
    theorem in KeYmaera X.

    You must provide a Bellerophon tactic to complete the
    proof.

    ## Background Information

    Here is some information about dGL.
    In differential game logic (dGL), there are two players:
```

```
- **Angel**: The controllable player who makes choices we
  can control. Angel constructs include:
  - Angel choices: `a ++ b` (we choose between a and b)
  - Angel loops: `{a}*` (we choose how many iterations)
  - Angel tests: `?P` (we can proceed if P holds,
    otherwise we immediately lose)
  - Angel ODEs: `{x'=f(x) & Q}` (we control the duration)
  - Angel nondeterministic assignments: `x := *` (we
    choose the value)

There is the dual operator `^@` that switches Angel and
Demon roles in a game construct.
Duals can be nested arbitrarily, so carefully check
nesting levels to determine which player controls each
action.
Being within an odd number of `^@` operators indicates
Demon control, while an even number (including zero)
indicates Angel control.
For example, in the game `{(a ++ b)^@; (c++d)}^@`, the
choice between `a` and `b` is controlled by Angel, being
within two `^@` operators, while the choice between `c`
and `d` is controlled by Demon.
Using this notation, we go over the Demon constructs
below, indicating that they are Demon-controlled by
wrapping them in a single `^@` operator.

- **Demon**: The adversarial player who makes choices we
  cannot control. Demon has all the same constructs as
  Angel.
  - Demon choices: `{a ++ b}^@` (adversary chooses
    between a and b)
  - Demon loops: `{a*}^@`
    (adversary chooses how many iterations)
  - Demon tests: `{?P}^@` (adversary can proceed if P
    holds, and immediately loses otherwise. So Angel can
    *assume* in subsequent game play that P held at this
    point; otherwise she would already have won.)
  - Demon ODEs: `{x'=f(x) & Q}^@` (adversary controls the
    duration)
  - Demon nondeterministic assignments: `{x := *}^@`
    (adversary chooses the value)

Additionally, the non-controlled operations are:
- Assignment: `x := e` (deterministic assignment like in
```

~~~
  regular programs)
- Sequential composition: `a; b` (first play a, then b)

## Notes
- ODEs can sometimes introduce unmatched single quotes,
  so account for that when parsing.
- In syntax such as { a++b; x:=* }*^@, the `^@` applies
  to the entire loop including its body. In this
  example, Demon controls the choice within the loop
  between a and b, as well as the assignment to x,
  because both are inside one `^@`.

Here is some information about Bellerophon, the tactic
language in which you must output.
# Bellerophon Tactic Guide (KeYmaera X 5.1)

## 1. Overview

**KeYmaera X** proves properties of hybrid systems using
differential dynamic logic (dL) or hybrid games using
differential game logic (dGL).
A **tactic** is a small program that automatically applies
proof rules.
**Bellerophon** is the language for writing these tactics.
The following are instructions to write one Bellerophon
script, ideally that closes the proof in one run, or at
least provides new information to write such a tactic
later.

---

## 2. Proof State Basics

To write good tactics, you should internally keep track
of proof state. A tactic maps one proof state to another.
A proof state consists of zero or more goals (subgoals).
Each goal is a sequent consisting of an antecedent
(assumptions) and a succedent (formulas to prove).

### 2.1 Sequents and Antecedents in the KeYmaera X CLI

A **sequent** in the CLI is written as:

```
~~~

```
-1:  x>=0 GreaterEqual
-2:  y=0 Equal
==> 1:  x>=0 GreaterEqual
```

which corresponds to the following sequent in mathematical notation:

```
x>=0, y=0  |-  x>=0
```

Formulas are indexed.
Negative indices indicate antecedent formulas, and positive indices indicate succedent formulas.

The antecedent lists assumptions.
The succedent lists formulas to prove.

We can assume all antecedent formulas are true and must use them to prove at least one succedent formula.
You just need to prove one succedent formula to close the goal.

2.2 Goals and Subgoals

Tactics transform the current goal into zero or more subgoals.
You apply tactics until no subgoals remain, which would indicate the theorem is proved.

2.3 Formula Indexing

Many tactics require a formula index.
A common failure mode is using the wrong index.
Use printing (Section 3.2) to inspect the state and indices while debugging.

---

3. Proving Interactions

3.1 Branches (Multiple Goals)

When a tactic creates multiple goals (e.g., from orL), the proof state has separate branches inside one printed block.

Example pattern:

```
===== State of Hard Cases ==== ElidingProvable(Provable{
        -1:  <formula for case 1>  Or
        -2:  <assumption>          Greater
        ...
      ==> 1:  <modal or post-condition>  Box
        from
        -1:  <formula for first disjunct>  And
        ...
      ==> 1:  <some goal 1>  Box
        with
        -1:  <formula for remaining disjuncts>  Or
        ...
      ==> 1:  <some goal 2>  Box}) =====
print done (no progress, 1s)
Done .
```

Here:

- The from part shows the first branch (e.g., first disjunct, such as a>=0 & ...).
- The with part shows the other branch.
- Each branch has its own sequent (==> line) with indexed formulas.

Each branch is a separate goal that must be closed.

You should check if a subgoal is closed by printing it. If the subgoal is proved, you should see something like:

```
===== subgoal state ==== ElidingProvable(Provable{
    -1:  x>0 Greater
    ==> 1:  x>=0 GreaterEqual proved}) =====
    print done (proved, 1s)
```

3.2 Running and Rerunning Tactics; Printing State

If you know of a tactic that you want to slightly modify, simply reuse the same text, and edit it as needed.

To inspect the current proof state at any point, use the tactic:

```
print("State of Hard Cases")
```

This prints the current provable state (all goals and their indexed formulas) at the point where the tactic is invoked.
Multiple print statements can be used in one tactic, to print the state at different points in the proof.
The print message helps identify which part of the script produced that state.

3.3 Printing for Debugging

For debugging, when the proof state has been reduced to one or more purely arithmetic subgoals, you can:

1. Use print to display the current arithmetic constraints.
2. Copy these constraints into an external tool such as Z3 to search for counterexamples by checking if the formula (AND of all antecedent constraints) -> (OR of all succedent constraints) is valid (i.e., its negation is unsatisfiable).

---

4. Bellerophon Essentials

- A script is a sequence of tactic expressions.
- t1 ; t2 runs t1 then t2.
- <(t1, t2) applies t1 to the first branch and t2 to the second branch. Note the absence of a closing angle bracket.
- A tactic either:
  - succeeds and may produce subgoals,
  - fails, or
  - closes the current goal.

---

5. Primitive Tactics

5.1 Logical Tactics

Common examples (indices refer to formulas in the sequent):

```
- Implication: implyR(i)
      This transforms a goal of form |- A -> B into
      A |- B.
- Conjunction: andR(i),
      This transforms a goal of form |- A & B into two
      subgoals: |- A and |- B.
  - andL(i)
      This transforms a goal of form
        A & B, \Gamma |- \Delta into
      A, B, \Gamma |- \Delta,
      where the i-th antecedent formula in the original
      subgoal was A & B.
- Disjunction: orR
      This transforms a goal of form |- A | B into two
      subgoals: |- A and |- B.
  - orL(i)
      This transforms a goal of form
        A | B, \Gamma |- \Delta into
      two subgoals:
        A, \Gamma |- \Delta and B, \Gamma |- \Delta,
      where the i-th antecedent formula in the original
      subgoal was A | B.
- Universal: allR
      This transforms a goal of form
      |- \forallx. P(x)
      into |- P(y) for a fresh variable y.
  - allL(i):
      This transforms a goal of form
        \forallx. P(x), \Gamma |- \Delta
      into P(t),
        \Gamma |- \Delta for a term t, where the i-th
      antecedent formula in the original subgoal was
      \forallx. P(x).
  - allL2R(i)
      This transforms a goal of form
        x=e, \Gamma |- \Delta, \forallx.
      P(x) into \Gamma |- \Delta, P(e),
      where the i-th antecedent formula in the
      original subgoal was x=e.
  - cut(formula)
      This introduces formula as an intermediate lemma
      to prove.
      It transforms a goal of form \Gamma |- \Delta
      into two subgoals:
```

    (1) Use subgoal: formula, \Gamma |- \Delta
    (2) Show subgoal: \Gamma |- formula.

5.2 Program / Modal Tactics

For modal formulas like [\alpha]\varphi and ⟨\alpha⟩\varphi, tactics handle:
- assignments,
- tests,
- nondeterministic choice,
- sequential composition,
- duals.

These reduce program structure to simpler logical or arithmetic goals. You will almost never need to use these low-level tactics directly, as they are all handled by the high-level `unfold` tactic.

A structure not handled by unfold, that you will need to handle directly is loops. Usually this needs the loop rule, `loop("invariant", i)`, which transforms a goal of form \Gamma |- [\alpha*]\varphi, \Delta where [\alpha*]\varphi is at index i into three subgoals:
    1. init subgoal: \Gamma |- invariant, \Delta
    2. step subgoal: invariant |- [\alpha]invariant
    3. post subgoal: invariant |- \varphi

Here, invariant is the (literal) formula you provide as a loop invariant.

There is a second class of modality tactics used to simplify not the program inside the modality, but the postcondition after the modality or otherwise modify the modality. The most common are:
- boxAnd(i)
    Transforms a goal of form [a](P&Q) to [a]P & [a]Q. Usually followed by andR(i) to split into two subgoals.
- GV(i)
    Transforms a goal of form \Gamma |- [a]P, \Delta to \Gammaconst |- P, \Deltaconst, where \Gammaconst and \Deltaconst are the subsets of \Gamma and \Delta that contain only formulas with constants (no variables).

~~~~
- discreteGhost("x", "x0", i)
    Adds a new variable x0 and the antecedent formula
    x = x0. The position, i, should be that of the
    modality formula (usually 1).

### 5.3 Differential Equation Tactics

For goals involving differential equations of the form:

```
[{x’ = f(x) & Q}] P
```

you can use the following high-level ODE tactics:
- dIRule(i) (Differential Invariant)
    Applies a differential invariant rule to the ODE
    formula at index i.
    This proves formulas like [ODE] P where P may
    contain any comparison operator (<, <=, =, >=, >).
    The rule generates two proof obligations:
  1. The postcondition P holds initially, and
  2. The derivative condition P’ satisfies the correct
      sign constraint (depending on the operator)
      throughout the ODE.
      For example, if in `[{x’ = f(x) & Q}] P`,  P is
      p <= 0, the check is
      `Q |= [x’:=f(x)] p’ <= 0`;
        if P is p > 0, the check is
        `Q |= [x’:=f(x)] p’ > 0`;
        and similarly for the other operators.
        Along with Q, any assumptions from the
        antecedents that had only constants (no
        variables) are also kept.
        Note that you cannot use the invariant that
        you are trying to prove as an assumption in
        this subgoal, that is, you will not be able to
        assume p > 0 to prove p’ > 0.
    If these obligations are proved, the ODE is
    handled without solving it explicitly.
    Note how to be able to prove subgoal 2, we may
    want to strengthen the domain constraint Q using
    dC first before applying dIRule.

- dC(\\varphi", i) (Differential Cut)
~~~~

Adds an additional domain constraint \varphi to the ODE at index i.
This transforms a goal with `[{x'=f(x) & Q}] P` at position i into two subgoals:

1. Use:
   The formula at position i becomes
   `[{x'=f(x) & Q & \varphi}] P`
2. Show:
   The formula at position i becomes
   `[{x'=f(x) & Q}] \varphi`

- dW(i) (Differential Weakening)

Applies weakening to an ODE goal at index i.
If the formula at index i was `[{x'=f(x) & Q}] P`, the new goal becomes `Q |- P`, which removes the ODE entirely. Along with Q, any assumptions from the antecedents that had only constants (no variables) are also kept.
It makes sense to apply dW only after sufficiently strengthening the domain constraint Q using dC first.

These tactics reduce ODE proof obligations to simpler goals.
A common pattern is to cut in *solutions* or *bounds* of the variables involved in the ODE and postcondition using dC.
In the branch where we need to show that the bound holds, it is common to use dIRule.
In the other, "use" branch, we typically apply further dC cuts to accumulate more bounds.
Then after accumulating sufficiently many such cuts, we apply dW followed by QE to finish the proof.
Sometimes the order of cuts really matters, as earlier cuts may become important to closing the dIRule obligations of later cuts.

Often, bounds are best expressed in terms of the initial values of variables before the ODE starts, e.g., x <= x0 + v*t for a variable x with initial value x0 and constant velocity v over time t.
In this case, if x0 does not already exist, a useful tactic is `discreteGhost("x", "x0", i)`, which adds a new variable x0 and the antecedent formula x = x0. The position, i, should be that of the ODE formula (usually

1).

Besides dIRule, another useful tactic that is especially useful for ODEs with exponential solutions, is the Darboux tactic:
`dbx("g", 1)`. Given a goal of form
`A |- [x’=f(x) & Q] p>=0`, this tactic generates two subgoals: `A |- p>=0` and `Q |- p’ >= g*p`, where p’ is the Lie derivative of p along the ODE x’=f(x). Here, >= can be replaced by any comparison operator. dbx is usually the way to go when we want to prove something about a variable whose derivative is dependent on itself.
In cases where you are trying to prove, e.g., p’>=0 and it is not immediate that p’>0, but it is true that p’ >= g*p, you should use Darboux and not dIRule (here >= and > can be replaced by the other comparison operators).
Unlike dIRule, dbx does not create further subgoals, and is itself a terminal tactic.

5.4 Arithmetic Tactics

When there are no modalities left, the goal is purely arithmetic.
Then, it is possible to use the `QE` tactic to close the goal.
It is recommended to use `QE("Z3")` which runs QE with the Z3 SMT solver backend, as it is generally most effective.

If QE times out, try hiding irrelevant formulas using `hide(i)` to remove formula with index i from the subgoal permanently.
It can also sometimes help to use the `fullSimplify` tactic to make expressions simpler.

If QE fails, there is likely an error in the proof or the theorem is not valid. It can help to look for a counterexample as described in Section 3.3.

---

6. Working With Indices

6.1 Index Semantics

Indices refer to the current goal only:
- 1 is the first succedent formula.
- 2 is the second succedent formula, etc.
- -1 is the first antecedent formula.
  - -2 is the second antecedent formula, etc.

After some tactics, the sequent may change, so indices may point to different formulas.

6.2 Common Failure: Incorrect Positions

A common reason for tactic failure is incorrect indices, e.g.:
- the index no longer refers to the formula you expect, or
- the selected formula does not match the rule (e.g., andL on a non-& formula).

Debugging pattern:
1. Insert print before the failing step.
2. Check the exact index and formula.
3. Update the index or tactic accordingly.

---

7. Combinators

Useful control constructs:
- Sequential composition: ‘t1 ; t2‘
  Run t1 then t2 on the resulting goals.
- Branching ‘<(t1, t2)‘
  Apply t1 to the first branch and t2 to the second, after a rule that splits the goal.

Combining these is generally enough to express proofs.

---

8. High-Level Tactics

High-level tactics:

- perform several logical steps at once,
- handle common patterns in hybrid programs,
- push the proof toward arithmetic goals.

The two most common ones are:

- unfold
  Unfolds simpler constructs (assignments, tests, choices, sequences, antecedent conjunctions, dual, implication). Often changes sequent structure and might create multiple branches.
- auto
  Applies a library of rules to try to automatically prove the goal. Generally succeeds on simpler goals.

You can use these to make the tactic shorter, but the tradeoff is less control and harder debugging.

---

9. Debugging

Common debugging workflows:

1. Insert print between suspicious steps to see the goal after each one. This can help identify where the proof diverges from expectations.
2. Remove steps until the error disappears. Once behavior is as expected, add more steps.

---

10. Additional Tips:

10.1 Syntactic:

- Comments in Bellerophon are written using the syntax `/* This is a comment */`.
- Do not use a concluding semicolon after the last tactic in your script. The semicolon is only a separator, not a terminator. Write `t1 ; t2`, not `t1 ; t2 ;`.
- The branching combinator `<( ... )` does not have a closing angle bracket. Write `<(t1, t2)`, not

```
  ‘<(t1, t2)>‘.
- When using branching tactics like dC or cut, further
  development should be nested inside the branches using
  ‘<( ..., ... )‘ syntax. Write ‘dC("formula", pos);
  <(t1; dC("formula", pos); <(t2, t3), t4)‘, not
  ‘dC("formula", pos); <(t1, t4) ; dC("formula", pos);
  <(t2 ; t3)‘. Here, the first dC splits the goal into
  two branches. On the first branch we apply t1 followed
  by dC that further splits into two branches where we
  apply t2 and t3 respectively. On the second branch of
  the first dC, we apply t4.
- Output along the lines of "Exception in thread "main"
  6:2 Lexer 6:2 Lexer does not recognize input at ..."
  indicates that there is a syntax error in the
  Bellerophon tactic. There can still be syntax errors
  that do not get caught by the lexer, though.
- The Bellerophon parser supports only ASCII characters.
- You can apply tactics like andL and orL only to
  formulas that have the corresponding logical
  connective at the top level. For example, andL would
  not apply to a | b & c because the top-level connective
  is |, since & binds more tightly. A good pattern is to
  use unfold to split up all top-level conjunctions at
  once, and then apply orL as needed.

### 10.2 Common Proof Patterns:
- Sometimes, it is helpful to split out disjunctive cases
  in the antecedent using orL, because the disjuncts
  correspond to different control modes requiring
  different reasoning.
- When there are top-level conjunctions in the
  antecedent, you can either split them one-by-one using
  andL, or use unfold to split all at once. The latter
  is often more concise, while the former gives more
  control. It is often necessary to split conjunctions
  to expose individual underlying assumptions to which
  further rules apply, e.g., disjunctions for orL case
  splitting, or equations on which to apply allL2R.
- When you see errors about inapplicability of a tactic
  due to formula shape mismatch, try to work around by
  using a different tactic or by introducing cuts to
  reshape the goal into a more convenient form.
- For time-triggered controllers, ‘t>=0‘ is often a
  useful invariant to cut in using dC, as it ensures we
```

```
  only have to reason about ODE properties after
  nonnegative time durations.

### 10.3 Other Advice:
- Do not assume a branch is closed until you print and
  verify that it shows "proved". Therefore, after
  applying a terminal tactic like auto or QE, it is good
  practice to print the state right after to check that
  the tactic indeed closed the goal.
- ‘unfold‘ steps are important and should not be
  forgotten. They often change the shape of the subgoal,
  and their can lead subsequent tactics to fail because
  the positions of sequents they expect are no longer
  present. It is a good idea to start tactic scripts
  with ‘unfold‘ to normalize the formula first.
- For the full list of tactics, see https://github.com
/LS-Lab/KeYmaeraX-release/tree/master/keymaerax-webui
/src/main/resources/help/axiomsrules.
- Duals are usually handled by ‘unfold‘, which generally
  eliminates the outermost dual by flipping the modality
  (box to diamond or vice versa). Otherwise the
  individual tactics are ‘duald(i)‘ that handles diamond
  modality containing a dual game at position i and
  similarly and ‘dualb(i)‘ for box modality.
- unfold generally handles implications, so implyR(1) is
  usually not necessary after unfold.
- dbx is strict about the shape it allows: the
  postcondition formula after the box must be a single
  inequality or equation with 0 on the RHS. Rewrite the
  goal using a differential cut if needed to get it into
  this shape.
- When you have two subsequent formulas, e.g., \Gamma |-
  [\alpha1*]\varphi, [\alpha2*]\varphi,
  (i.e., [\alpha1*]\varphi at position 1 and
  [\alpha2*]\varphi at position 2), remember that you
  only need to prove one of the subsequents.
  You can think carefully about which one is more
  promising to prove, and get
  rid of the other using hide(2) or hide(1) as
  appropriate. But don’t eliminate options too early: if
  you have a disjunction ‘f1 | f2‘ in your assumptions,
  it may be that [\alpha1*]\varphi is provable when f1
  holds, and [\alpha2*]\varphi is provable when f2
  holds, so you need to keep
```

both options until you have split on the disjunction using orL.
- If an arithmetic proof obligation times out on auto or QE, it is a good idea ot check if the goal is even valid by internally checking if a solver like Z3 can find a counterexample within some small timeout.

10.4 On Loop Invariants:

identifying the right loop invariant for proving properties of loops (using tactic ‘loop("invariant", i)‘) can be tricky, yet critical to allowing the subsequent proof branches succeed. A common pattern is to start with an invariant, and then strengthen or weaken it as required as one encounters counterexamples or difficulties in closing the subgoals, init, step and post, out of which step is usually the hardest to close.

The invariant can be no stronger than the initial assumptions, that is, ‘\Gamma1 & \Gamma2‘ in
‘\Gamma1, \Gamma2 |- [\alpha*]\varphi‘.
Anything stronger than ‘\Gamma1 & \Gamma2‘ would not allow the init subgoal to close.
The invariant cannot be weaker than \varphi, the postcondition. Anything weaker than \varphi would not allow the post subgoal to close.
The challenge is then to weaken the assumptions enough, or strengthen the postcondition enough, that the resulting invariant is *inductive*, allowing the step subgoal to close.
For example, consider the simple loop goal
‘x=1 |- [{?x>=1;^@; {x'=1}}*]x>=0‘.
x=1 is too strong as an invariant because as time elapses, x becomes larger than 1, so it is not true that x=1 |- [?x>=1;^@; {x'=1}]x=1.
x>=0 is too weak as an invariant because Demon loses within the loop at the test when, e.g., x=0, so it is not true that x>=0 |- [?x>=1;^@; {x'=1}]x>=0.
A suitable invariant is x>=1, which is weaker than the assumptions, stronger than the postcondition, and inductive in that if it holds initially, after running one iteration of the loop, it will continue to hold.

Other advice for identifying an invariant:

- A good heuristic is to start with the initial assumptions as the invariant, and then weaken them as needed, or start with the postcondition \varphi and strengthen it. Ask yourself: do the initial assumptions already look inductive? Does the postcondition already look inductive?
- It is usually a good idea to retain all assumptions on signs of constants. These often end up being critical to inductiveness. You lose nothing by including assumptions in the invariant, unless they are non-inductive and stop holding after some time.
- It is easy to get the invariant subtly wrong because of small inaccuracies or missing edge cases. But in the course of doing the proof, if you carefully analyze proof failures, you will discover the missing requirements, so do not hesitate to modify the invariant accordingly.
- Finding the right invariant is the only way to do proofs about Angel loops in Demon modality games or Dually, Demon loops in Angel games (i.e. ‘[\alpha*]\varphi‘ or ‘<\alpha*>\varphi‘). The only way to prove the latter is to first convert to ‘[\alpha*]\varphi‘ form, e.g. using unfold, and then apply the loop tactic.

10.5 On proving Angel choices in Angel games:

This section applies dually also to Demon choices in Demon games.
Consider an Angel choice in an Angel game, e.g., ‘x=1 | x=-1 |- <{ {v:=1; ++ v:=-1;}^@; {x’=v}}^@>(x>=1 | x<=-1)‘.
After applying ‘unfold‘, we get a two subsequents: ‘x = 1 | x = -1 |- [{x’=1}](x >= 1 | x <= -1), [{x’=-1}](x >= 1 | x <= -1)‘
This reflects how we can choose which subsequent to prove based on which choice Angel would have strategically made.
In this case, Angel would choose the first branch if x=1 initially, and the second branch if x=-1 initially.
Thus we could apply the following tactic to complete the proof, with the orL splitting on initial conditions and hideR choosing the right subsequent to then prove:
‘‘‘

```
unfold; orL(-1); <(
    hideR(2); dC("x>=1", 1); <(
        dW(1); QE,
        dIRule(1); <(
            QE,
            unfold; QE
        )
    ),
    hideR(1); dC("x<=-1", 1); <(
      dW(1); QE,
      dIRule(1); <(
          QE,
          unfold; QE
      )
    )
)
```

10.6 Common Error Messages:

Subgoal mismatch.

Error messages of the form "<(e)(v) is only defined when len(e) = len(v), but 2!=0 subgoals (v)" indicates a branching combinator `<( ... )` has the wrong number of branches.
In this example, a branching combinator for two branches was used, but the goal was actually already closed (0 subgoals), and the fix is to remove the branching combinator since the subgoal is already proved.

Sometimes, this error message is also reached by auto. This is a bug in the implementation of auto that is triggered on some formula shapes. If this happens, since you cannot control auto, just try to work around by using a different tactic or breaking down the proof into smaller steps.

dW Bug.

Sometimes, dW fails with an error message like

```
    Not found: locator ’R=="a&b->c" of position tactic ...
    does not match anywhere in succedent of ... from ==>
    1: a -> c
```

where typically a is the domain constraint of the ODE to which we are applying dW, c is the postcondition, and b consists of some formulas from the antecedent with only constants.

This is a bug in KeYmaera X.
A trick that often works around this bug is cutting in "1=1". Replace `dW(i); *subsequent tactic*` with `cut("1=1"); <(dW(i); *subsequent tactic*, QE("Z3") using "1=1")`.
The syntax "using "1=1"" tells QE to ignore all antecedent and consequent formulas in this subgoal except "1=1".

If the error persists, you can try first removing antecedent formulas with only constants that are appearing in the error message, and only then apply dW.
Use hide(i) to hide the formula at position i.

End of Bellerophon Tactic Guide

Output Instructions

Output Format

In your response, first briefly summarize you reasoning.
Then write a newline.
Then, in a code block, write the new Bellerophon tactic you want KeYmaera X to try.

Advice

If you are not sure what to how to complete the proof, you can provide some initial steps.
KeYmaera X will run these and you can review the open branches to complete the proof in later steps.
Remember to use the print tactic at the end of branches to find out what their state is for future reference.

Tactics can be sequentially combined using semicolons (e.g., `unfold; auto`. Note that there is no semicolon after the last tactic in a sequence).
If the proof branches, e.g., after dC, then use the

```
    combinator `<(,)`, e.g., `dC; <( print("this is branch
    one."); print("this is a second print still in branch
    one."), print("this is branch two.") )`.

    Common mistakes to avoid:
    - When expanding on a previous tactic script, do not
      forget the initial `unfold` tactic because this can
      lead to subsequent tactic failing. You should retain
      the entire previous tactic and expand on it, unless
      there is an error requiring you to backtrack.
    - Do not modify the old tactic any more than strictly
      necessary. Tactics can be fragile and a single
      misplaced andL can break the proof.
    - Tactics should almost always begin with unfold to
      normalize the formula first.
    - Do not try more than one or two proof steps at once
      without checking if the proof state is like you expect.
      Do not try to solve more than one branch at a time
      because the complexity can quickly become unmanageable.
    - Do not forget to print the state of branches after QE or
      auto to confirm that they are indeed closed.
    - No unicode. Only ascii characters are allowed.

    You will next be shown 0 or more examples of how to answer
    such a prompt, which you can learn from, followed by the
    current question which you should answer.
- role: user
  content: |-
    The formula that you are trying to prove is:
    (t <= T & v = 0 & w = 0 & !(x = 0 & y = 0)) ->
    <{x’ = v, y’ = w, v’ = om * w, w’ = - om * v, t’ = 1 &
    t <= T}^@>(!(x = 0 & y = 0) & (!x*w - y*v = 0 | (v = 0 &
    w = 0)))

    Now, please provide a Bellerophon tactic to try next.
    You should retain the entire previous tactic and expand
    on it, unless there is an error requiring you to
    backtrack.
    In your response, first briefly summarize your reasoning.
    Then write a newline.
    Then, in a code block, write the entire Bellerophon tactic
    you want KeYmaera X to try.
- role: assistant
  answer:
```

~~~~
    mode: null
    content: |
      Intuitively we expect this formula to be valid because
      if initially v=w=0, then v and w remain 0, so x and y
      remain constant, so if we start not at the origin we
      stay not at the origin. In this way, it should in the
      end hold that !(x = 0 & y = 0) & (v = 0 & w = 0) which
      implies the postcondition.
      First lets unfold and differential-cut in the invariant
      v=0 & w=0.
      We can then print the state of the second branch to
      check that things look right so far.

      ```
      unfold;
      dC("v = 0 & w = 0", 1) <(
        label("Use cut"),
        label("Show cut");
        print("Show branch state.")
      )
      ```
- role: user
  content: |-
    The formula that you are trying to prove is:
    (t <= T & v = 0 & w = 0 & !(x = 0 & y = 0)) ->
    <{x’ = v, y’ = w, v’ = om * w, w’ = - om * v, t’ = 1 &
    t <= T}^@>(!(x = 0 & y = 0) & (!x*w - y*v = 0 | (v = 0 &
    w = 0)))

    This is what you have been trying so far and the results.
    Pay careful attention to not repeat past mistakes and make
    progress based on what you have learned.

    We expect this formula to be valid because if initially
    v=w=0, then v and w remain 0, so x and y remain constant,
    so if we start not at the origin we stay not at the
    origin. In this way, it should in the end hold that
    !(x = 0 & y = 0) & (v = 0 & w = 0) which implies the
    postcondition.
    We have differential-cut in the invariant v=0 & w=0 and
    shown that is holds using the tactic below. We have
    printed the proof state of the "Show cut" branch and it
    was successfully closed.
~~~~

```
    unfold;
    dC("w=0&v=0", 1); <(
      label("Use cut"); print("Use branch state."),
      label("Show cut");
      dC("v^2+w^2=0", 1); <(
        label("Use second cut"); dW(1) ; QE,
        label("Show second cut"); dIRule(1); <(
          unfold; QE("Z3");
          print("Show branch of dC, dI init. State after QE."),
          unfold; QE("Z3");
          print("Show branch of dC, dI inductiveness subgoal.
          State after QE.")
        )
      )
    )

    Both dIRule branches are proved.
    The proof state corresponding to the "Use cut" branch is:
    -1:  t<=T    LessEqual
    -2:  v=0     Equal
    -3:  w=0     Equal
    -4:  !(x=0&y=0) Not
     ==> 1:  [{x’=v,y’=w,v’=om*w,w’=-om*v,t’=1&t<=T&w=0&v=0}]
             (!(x=0&y=0)&(!x*w-y*v=0|v=0&w=0)) Box})
    So it remains to prove this goal.

    Now, please provide a Bellerophon tactic to try next.
    You should retain the entire previous tactic and expand
    on it, unless there is an error requiring you to
    backtrack.
    In your response, first briefly summarize your reasoning.
    Then write a newline.
    Then, in a code block, write the entire Bellerophon tactic
    you want KeYmaera X to try.
- role: assistant
  answer:
    mode: null
    content: |
      We want to show that after the ODE, !(x = 0 & y = 0) &
      (v = 0 & w = 0) and already have v=0&w=0 in the domain
      constraint. Intuitively, !(x = 0 & y = 0) because this
      is true initially neither x nor y changes. To cut in
      the fact that x and y do not change, lets dC in
      (x=x0 & y=y0) where x0 and y0 are the initial values of
```

~~~
      x and y, which we can introduce via discreteGhost
      (which should also automatically change the initial
      assumption !(x = 0 & y = 0) into !(x0 = 0 & y0 = 0)).
      Later, we can then show the "show" branch by dIRule
      because x and y do not change, and the "use" branch by
      dW followed by QE because x=x0, y=y0 and v=0 & w=0
      should be in the antecedents from the domain constraint
      and !(x0=0 & y0=0) should be in the antecedents because
      it has only free variables x0 and y0 which are not
      changed by the ODE.
      But first, lets print to make sure the subgoals look as
      expected.

      ```
      unfold;
      dC("w=0&v=0", 1); <(
          "Use":
          discreteGhost("x", "x0", 1);
          discreteGhost("y", "y0", 1);
          dC("x=x0&y=y0", 1); <(
              "Use-use":
              print("State of the use-use branch dW."),
              "Use-Show":
              print("State of use-show branch.")
          ),
          "Show":
          dC("v^2+w^2=0", 1); <(
              "Use":
              dW(1) ; QE,
              "Show":
               dIRule(1); <(
                unfold; QE("Z3"),
                unfold; QE("Z3")
          )
      )
      ```
- role: user
  content: |-
    The formula that you are trying to prove is:
    (((a >= 0 & a >= (-v)/T & end - (p + v*(T - t) + 1/2*a*
    (T - t)^2 + (v + a*(T - t))^2/(2*B)) >= 0) | (a < 0 &
    a >= (-B) & end - (p - v^2/(2*a)) >= 0) | (a < (-B) &
    end - (p + v^2/(2*B)) >= 0)) & A > 0 & B > 0 & T > 0 &
    v >= 0) -> <{p’ = v, v’ = a, t’ = 1 & t <= T & v >= 0}^@>
~~~

```
(end - (p + v^2/(2*B)) >= 0 & A > 0 & B > 0 & T > 0 &
v >= 0)
```

This is what you have been trying so far and the results. Pay careful attention to not repeat past mistakes and make progress based on what you have learned.

```
Proposed Tactic:
unfold;
boxAnd(1); andR(1);
print("State")
.
Outcome:   ===== State ==== ElidingProvable(Provable{
==> 1:  (a>=0&a>=(-v)/T&end-(p+v*(T-t)+1/2*a*(T-t)^2+
       (v+a*(T-t))^2/(2*B))>=0|a < 0&a>=-B&end-(p-v^2/
       (2*a))>=0|a < -B&end-(p+v^2/(2*B))>=0)&A>0&B>0&
       T>0&v>=0-><{{p'=v,v'=a,t'=1&t<=T&v>=0}}^@>
       (end-(p+v^2/(2*B))>=0&A>0&B>0&T>0&v>=0) Imply
  from
   -1:  a>=0&a>=(-v)/T&end-(p+v*(T-t)+1/2*a*(T-t)^2+
        (v+a*(T-t))^2/(2*B))>=0|a < 0&a>=-B&end-(p-v^2/
        (2*a))>=0|a < -B&end-(p+v^2/(2*B))>=0 Or
   -2:  A>0 Greater
   -3:  B>0 Greater
   -4:  T>0 Greater
   -5:  v>=0 GreaterEqual
==> 1:  [{p'=v,v'=a,t'=1&t<=T&v>=0}]end-(p+v^2/(2*B))>=0
        Box
  with
   -1:  a>=0&a>=(-v)/T&end-(p+v*(T-t)+1/2*a*(T-t)^2+
        (v+a*(T-t))^2/(2*B))>=0|a < 0&a>=-B&end-(p-v^2/
        (2*a))>=0|a < -B&end-(p+v^2/(2*B))>=0 Or
   -2:  A>0 Greater
   -3:  B>0 Greater
   -4:  T>0 Greater
   -5:  v>=0 GreaterEqual
==> 1:  [{p'=v,v'=a,t'=1&t<=T&v>=0}](A>0&B>0&T>0&v>=0)
        Box}) =====
print done (no progress, 1s)
Done .
```

Now, please provide a Bellerophon tactic to try next. You should retain the entire previous tactic and expand on it, unless there is an error requiring you to

```
    backtrack.
    In your response, first briefly summarize your reasoning.
    Then write a newline.
    Then, in a code block, write the entire Bellerophon tactic
    you want KeYmaera X to try.
- role: assistant
  answer:
    mode: null
    content: |
      We see the proof has split into two goals, the first
      with consequent ‘[{p’=v,v’=a,t’=1&t<=T&v>=0}]end-(p+v^2/
      (2*B))>=0` and the second with consequent
      ‘[{p’=v,v’=a,t’=1&t<=T&v>=0}](A>0&B>0&T>0&v>=0)`. The
      second one should be easy to prove, so we will try that
      first with the auto tactic.
      We will print the state after trying auto to see if it
      was successful, or if more work is required. We should
      also print the state of the harder branch so that we
      can plan our next steps there.
      Our tactic will carefully retain the initial
      ‘unfold; boxAnd(1); andR(1)` to ensure we don’t lose the
      progress we have already made.

      ```
      unfold; boxAnd(1); andR(1) <(
        label("harder");
          print("State in harder branch"),
        label("easier");
          auto; print("State after auto on easier branch")
)
      ```
- role: user
  content: |-
    The formula that you are trying to prove is:
    (kA > 0 & kB > 0 & kC > 0 & kT >= 0 & T > 0 &
    A >= 0 & B >= 0 & C >= 0 & Temp >= 0 &
    (kB*A <= kA*B & kA*Temp + kT*A <= kA*Tmax |
     kA*B <= kB*A & kB*Temp + kT*B <= kB*Tmax)) ->
    < { { { isOn:=0; ++ isOn:=1; }; t:=0;
        { A’ = - A*B*Temp*isOn*kA,
          B’ = - A*B*Temp*isOn*kB,
          C’ = A*B*Temp*isOn*kC,
          Temp’ = A*B*Temp*isOn*kT,
          t’ = 1 & t<=T & A>=0 & B>=0 & C>=0 & Temp>=0
```

```
        };
      }* }^@ > Temp<=Tmax

    Now, please provide a Bellerophon tactic to try next.
    You should retain the entire previous tactic and expand
    on it, unless there is an error requiring you to
    backtrack.
    In your response, first briefly summarize your reasoning.
    Then write a newline.
    Then, in a code block, write the entire Bellerophon tactic
    you want KeYmaera X to try.
```

3. Summarize summarizes the proof state in preparation for the next iteration of the verification loop. It is only used for the Qwen model evaluation.

```
- role: system
content: |
    You are given a step in an attempt to prove a
    differential game logic (dGL) formula in KeYmaera X,
    along with context about the previous state of the proof
    before the step, including the global plan and the
    things we have tried so far.
    Your task is to provide a new summary of the current
    state of the proof that now accounts for the latest step
    towards completing the proof and its outcome.

    Your proof summary will be used to then decide what the
    next step in the proof should be (if you decide to
    continue the proof) or how to change the formula (if you
    choose to revise the formula). Therefore it should
    remember all relevant information: previous mistakes and
    how you resolved them, location of sequents in subgoals
    that currently remain open, counterexamples discovered at
    the end of previous proof directions, etc. Expect that
    after many steps, the summary will be long. Be sure to
    include all relevant information.

    ## Input:
    1. The dGL formula that you are trying to prove.
    2. The summary of the proof state before this step.
    3. The proof tactic that you then ran this time to get
       one step closer to completing the proof.
    4. The outcome of running that tactic.
```

Here, "step" means not a literal single step in the proof, but rather a new tactic that you applied that modifies the last one that was applied, either resolving errors or making progress towards completing the proof.

More information:

Here is more information about dGL:
[This section is the same as in the GetTactic query.]

Here is more information about KeYmaera X tactics:
[This section is that same as in the Bellerophon Tactic guide in the GetTactic query.]

Output Shape:

You should return a detailed summary of the proof state. In the summary, you should include the following sections, many of which should carry over from the previous summary provided to you.

1. The global proof plan so far.
Mostly carries over from previous summary unless the latest step surfaced an issue or new insight.
2. (New) Previous Tactic (that is, the currently applied tactic in the latest step).
Just copies over from input item 3. Do not make any changes or omissions.
3. The current open subgoals with exact sequent locations if we know them.
Copies over from input item 4.
4. Any mistakes made so far and how we fixed them. This is important to avoid repeating the same mistakes in future steps. Guide the reader so they know what they must do to avoid repeating these mistakes.
Partly carries over from previous summary. Might need to add new information if the latest step surfaced a new mistake or insight on how to fix a mistake.
Remember to include all relevant context, such as the tactic leading up to the issue, the reason for the problem, and how to avoid it in the future.
5. Previous proof directions that failed, if any, along with details such as the counterexamples discovered at the end of those directions.

```
    Mostly carries over from previous summary unless the
    latest step surfaced a new issue or insight.
    6. Any other relevant information about the proof state
    that will let a reader work on the proof from here
    assuming no other information besides the original
    formula to prove is provided to them.
    Partly carries over from previous summary. Might need to
    add new information if the latest step surfaced new
    relevant information. Remember that your summary and the
    original formula will be all the reader has to work from
    to proceed with the proof, so do not omit useful details.
    Some of these sections may be empty if the proof is still
    in early stages, but you should still include the
    section headers for completeness."
- role: user
content: |
    The formula that you are trying to prove is:
    (kA > 0 & kB > 0 & kC > 0 & kT >= 0 & T > 0 & A >= 0 &
    B >= 0 & C >= 0 & Temp >= 0 & (kB*A <= kA*B & kA*Temp +
    kT*A <= kA*Tmax | kA*B <= kB*A & kB*Temp + kT*B <= kB*
    Tmax)) -> < { { { isOn:=0; ++ isOn:=1; }; t:=0;
    { A' = - A*B*Temp*isOn*kA, B' = - A*B*Temp*isOn*kB,
    C' = A*B*Temp*isOn*kC, Temp' = A*B*Temp*isOn*kT,
    t' = 1 & t<=T & A>=0 & B>=0 & C>=0 & Temp>=0 }; }* }^@ >
    Temp<=Tmax

    Remember that nested duals alternate control back. For
    example, in the game `{(a ++ b)^@; (c++d);}^@`, the
    choice between `a` and `b` is controlled by Angel, while
    the choice between `c` and `d` is controlled by Demon.

    Here is an analysis of the formula, discussing what Angel
    vs. Demon controls and the game's control modes.
        1) Angel-controlled actions
    - None within the game body. The entire loop is dualized,
      so all otherwise-Angel constructs inside it are under
      Demon control.

    2) Demon-controlled actions
    - Looping: { ... }*^@
      | Demon chooses how many iterations
      to execute (including zero; any finite number).
    - Discrete choice each iteration: { isOn := 0; ++
      isOn := 1; } | Within the single ^@, Demon chooses
```

whether to set isOn to 0 (OFF) or 1 (ON) for that iteration.
- ODE duration each iteration: { A’ = −A B Temp isOn kA, B’ = −A B Temp isOn kB, C’ = A B Temp isOn kC, Temp’ = A B Temp isOn kT, t’ = 1 & t <= T & A >= 0 & B >= 0 & C >= 0 & Temp >= 0 }
  | Demon controls how long
  to flow, subject to the evolution domain. He may stop immediately or at any time before hitting t = T or a state-constraint boundary (he cannot cross A,B,C,Temp < 0 or t > T).

  - Non-controlled deterministic assignments
- t := 0 at the start of each iteration (resets the per-iteration clock; not a choice).
- The chosen isOn := 0 or 1 is a deterministic assignment after Demon picks the branch.

3) Control modes (qualitative behaviors)
The per-iteration discrete choice creates two qualitative modes; Demon may switch modes between iterations by choosing isOn.

- OFF mode (isOn = 0)
- Dynamics: A’ = B’ = C’ = Temp’ = 0, t’ = 1.
- Behavior: State (A,B,C,Temp) is frozen; only the timer t increases up to T. This is an equilibrium for the chemistry/temperature.
- Invariants: All polynomial combinations below remain constant (trivially, since derivatives are zero).

- ON mode (isOn = 1)
- Dynamics: A’ = −kA A B Temp <= 0, B’ = −kB A B Temp <= 0, C’ = kC A B Temp >= 0, Temp’ = kT A B Temp >= 0, with t’ = 1 and domain t <= T, A,B,C,Temp >= 0.
- Monotonicity:
  A(t), B(t) are nonincreasing; C(t), Temp(t)
  are nondecreasing. If any of A, B, or Temp is zero, all chemical rates are zero (equilibrium subset).
- Polynomial invariants (valid in both modes because every derivative carries the common factor A B Temp isOn):
  - Stoichiometry: kB A − kA B = const
    (so the sign of kB A − kA B never changes; the "limiting reagent"

```
      relation stays fixed).
    - Thermal-reagent invariants:
    - kA Temp + kT A = const
    - kB Temp + kT B = const
    - Yield invariants (optional but useful):
      kC A + kA C = const and kC B + kB C = const.
- Bounds from invariants (safety envelopes chosen by initial
  branch):
    - If initially kB A <= kA B and
      kA Temp + kT A <= kA Tmax,
      then for all future time Temp <= Tmax
      because A >= 0 and
      kA Temp + kT A stays constant.
    - If initially kA B <= kB A and
      kB Temp + kT B <= kB Tmax,
      then likewise Temp <= Tmax from the second invariant.
    - These two cases are disjoint branches selected by the
      initial stoichiometric ordering; the ordering is
      preserved by kB A - kA B = const.
- Asymptotes/equilibria: Trajectories move along
  invariant manifolds until one reagent depletes
  (A -> 0 or B -> 0), then
  all rates drop to zero. Temperature approaches a finite
  limit bounded by the corresponding invariant, never
  exceeding Tmax under the given initial branch. If kT = 0,
  Temp is constant.

4) Overall control pattern
- Preconditions (outside the game) ensure positivity
  and pick
  one of two invariant safety envelopes: either
  kA Temp + kT A <= kA Tmax when kB A <= kA B, or
  kB Temp + kT B <= kB Tmax when kA B <= kB A. The
  stoichiometric ordering is invariant, so the chosen
  envelope remains applicable.
- Iteration (repeats a Demon-chosen finite number of times):
1) Demon picks isOn \in {0, 1}
   (OFF or ON for this iteration).
2) Deterministic reset t := 0.
3) Demon chooses how long to flow within the ODE domain
   (0 <= duration <= T).
    - OFF: no state change except t.
    - ON: A,B decay; C,Temp grow; all along the invariant
      manifolds keeping Temp within the selected bound.
```

- Demon controls both switching (isOn) and dwell time (ODE duration) as well as the number of repetitions; Angel has no in-game choices. The qualitative modes are OFF (equilibrium) and ON (reaction with monotone decay/growth along invariant manifolds), with safety enforced by the polynomial invariants regardless of Demon’s switching and timing.

Carefully account for who controls which choice while constructing any counterexamples.
The counterexample cannot, e.g., require Demon to make a choice that Angel controls.

Then you performed the following proof attempt:
We need an inductive invariant for the Demon loop. A natural invariant is the conserved linear bounds: either kA*Temp+kT*A<=kA*Tmax when kB*A<=kA*B, or kB*Temp+kT*B<=kB*Tmax when kA*B<=kB*A. Together with sign assumptions kA>0,kB>0,kT>=0 and A>=0,B>=0, this invariant implies Temp<=Tmax for the post branch. We set up the loop with this invariant, automatically discharge the init and post branches (splitting the disjunction in the post), and print the step branch after unfolding so we can handle ODE preservation next.

```
unfold;
loop("kA>0 & kB>0 & kT>=0 & A>=0 & B>=0 &
      ( (kB*A <= kA*B & kA*Temp + kT*A <= kA*Tmax) |
        (kA*B <= kB*A & kB*Temp + kT*B
<= kB*Tmax) )", 1); <(
/* init: assumptions imply invariant */
unfold; auto; print("Init branch after auto."
    ),
/* step: unfold body, we will handle preservation next */
unfold; print("
    Step branch after unfolding body."),
/* post: invariant implies Temp<=Tmax */
andL(-1); andL(-1); andL(-1); andL(-1); andL(-1);
orL(-1); <(
    andL(-1); QE("
    Z3"); print("Post branch case 1 closed (from kA*Temp +
    kT*A <= kA*Tmax)."),
```

```
    andL(-1); QE("Z3"); print("Post branch case 2 closed
    (from kB*Temp + kT*B <= kB*Tmax)."
    )
)
)
```

The outcome was:

```
Tactic andL applied at -1 on a non-matching expression in
ElidingProvable(Provable{
-1:  kA>0&kB>0&kT>=0&A>=0&B>=0&(kB*A<=kA*B&kA*Temp+kT*A<=
kA*Tmax|kA*B<=kB*A&kB*Temp+kT*B<=kB*Tmax)
And
-2:  kA>0 Greater
-3:  kB>0 Greater
-4:  kC>0 Greater
-5:  kT>=0
GreaterEqual
-6:  T>0 Greater
==> 1:  [{isOn:=0;++isOn:=1;}t:=0;{A'=-A*B*Temp*isOn*kA,
B'=-A*B*Temp*isOn*kB,C'=A*B*Temp*isOn*kC,Temp'=A*B*Temp*
isOn*kT,t'=1&t<=T&A>=0&B>=0&C>=0&Temp>=0}](kA>0&kB>0&kT
>=0&A>=0&B>=0&%28kB*A<=kA*B&kA*Temp+kT*A<=kA*Tmax|kA*B
<=kB*A&kB*Temp+kT*B<=kB*Tmax%29)
Box
from
-1:  kA>0 Greater
-2:  kB>0 Greater
-3:  kC>0 Greater
-4:  kT>=0 GreaterEqual
-5:  T>0 Greater
-6:  kA>0 Greater
-7:  kB>0&kT>=0&A>=0&B>=0&(kB*A<=kA*B&kA*Temp+kT*A<=kA*
Tmax|kA*B<=kB*A&kB*Temp+kT*B<=kB*Tmax)
And
==> 1:  [{isOn:=0;++isOn:=1;}t:=0;{A'=-A*B*Temp*isOn*kA,
B'=-A*B*Temp*isOn*kB,C'=A*B*Temp*isOn*kC,Temp'=A*B*Temp
*isOn*kT,t'=1&t<=T&A>=0&B>=0&C>=0&Temp>=0}](kA>0&kB>0&
kT>=0&A>=0&B>=0&%28kB*A<=kA*B&kA*Temp+kT*A<=kA*Tmax|kA*
B<=kB*A&kB*Temp+kT*B<=kB*Tmax%29)
Box})

in <( (unfold;auto;print("Init branch after auto.")),
(unfold;print("Step branch after unfolding body.")),
```

```
>>_<< )
(Failed on branch (andL(-1);andL(-1);andL(-1);andL(-1);
andL(-1);orL(-1);<( (andL(-1);QE("Z3");
print("Post branch case 1 closed (from kA*Temp + kT*A <=
kA*Tmax)."
)), (andL(-1);QE("Z3");
print("Post branch case 2 closed (from kB*Temp + kT*B <=
kB*Tmax)."
)) )))

in <( (unfold;auto;print("Init branch after auto.")),
(unfold;print("Step branch after unfolding body.")),
>>_<< )
[Full trace elided]
at edu.cmu.cs.ls.keymaerax.bellerophon.CoreLeftTactic.
computeResult(BellerophonSyntax.scala:623)
t... 40 more

[launcher] Restarting KeYmaera X with sufficient stack
space
/usr/lib/jvm/java-17-openjdk-amd64/bin/java -Xss20M -da
-jar /home/.../keymaerax.jar
-launch -prove /home/.../keymaerax_conjectures/kA0.kyx
-tactic unfold;
loop("kA>0 & kB>0 & kT>=0 & A>=0 & B>=0 & ( (kB*A <= kA*B &
kA*Temp + kT*A <= kA*Tmax) | (kA*B <= kB*A & kB*Temp + kT*B
<= kB*Tmax) )", 1); <(
/* init: assumptions imply invariant */
unfold; auto; print("Init branch after auto."
),
/* step: unfold body, we will handle preservation next */
unfold; print("
Step branch after unfolding body."),
/* post: invariant implies Temp<=Tmax */
andL(-1); andL(-1); andL(-1); andL(-1); andL(-1);
orL(-1); <(
andL(-1); QE("
Z3"); print("Post branch case 1 closed (from kA*Temp + kT*A
<= kA*Tmax)."),
andL(-1); QE("Z3"); print("Post branch case 2 closed (from
kB*Temp + kT*B <= kB*Tmax)."
)
)
)
```

```
-verbose -timeout 30
KeYmaera X Prover 5.1.2
Use option -help for usage and license information
Proving entries from 1 files
Proving /home/.../keymaerax_conjectures/kA0.kyx#Delphyne/kA0
...
unfold... unfold done (transformed goal(s), 44ms)
loop("kA>0&kB>0&kT>=0&A>=0&B>=0&
      (kB*A<=kA*B&kA*Temp+kT*A<=kA*Tmax|
       kA*B<=kB*A&kB*Temp+kT*B<=kB*Tmax)"
, 1)... loop done (added 2 goal(s), 60ms)
unfold... unfold done (no progress, 2ms)
auto... auto done (proved, 16ms)
print("Init branch after auto.")... INFO  ===== Init branch
after auto. ==== ElidingProvable(Provable{   -1:  kA>0
Greater
-2:  kB>0 Greater
-3:  kC>0 Greater
-4:  kT>=0 GreaterEqual
-5:  T>0 Greater
-6:  A>=0 GreaterEqual
-7:  B>=0 GreaterEqual
-8:  C>=0 GreaterEqual
-9:  Temp>=0 GreaterEqual
-10:  kB*A<=kA*B&kA*Temp+kT*A<=kA*Tmax|
      kA*B<=kB*A&kB*Temp+kT*B
<=kB*Tmax
Or
==> 1:  kA>0&kB>0&kT>=0&A>=0&B>=0&(kB*A<=kA*B&kA*Temp+kT*A
<=kA*Tmax|kA*B<=kB*A&kB*Temp+kT*B<=kB*Tmax)
And proved}) =====
print done (proved, 4ms)
INFO  ===== Step branch after unfolding body. ====
ElidingProvable(Provable{
-1:  kA>0&kB>0&kT>=0&A>=0&B>=0&(kB*A<=kA*B&kA*Temp+kT*A
<=kA*Tmax|kA*B<=kB*A&kB*Temp+kT*B<=kB*Tmax)
And
-2:  kA>0 Greater
-3:  kB>0 Greater
-4:  kC>0 Greater
-5:  kT>=0
GreaterEqual
-6:  T>0 Greater
==> 1:  Temp<=Tmax LessEqual
```

```
    from
    -1:  kA>0
    Greater
    -2:  kB>0 Greater
    -3:  kC>0 Greater
    -4:  kT>=0 GreaterEqual
    -5:  T>0 Greater
    -6:  kA>0 Greater
    -7:  kB>0 Greater
    -8:  kT>=0
    GreaterEqual
    -9:  A>=0 GreaterEqual
    -10:  B>=0 GreaterEqual
    -11:  kB*A<=kA*B&kA*Temp+kT*A<=kA*Tmax|
          kA*B<=kB*A&kB*Temp+kT*B<=kB*Tmax
    Or
    ==> 1:  Temp<=Tmax LessEqual}) =====
    Done /home/.../keymaerax_conjectures/kA0.kyx#Delphyne/kA0
    (failed)
    FAILED Delphyne/kA0: tactic=user,tacticsize=-1,budget=30[s],
    duration=-1[ms],qe=-1[ms],rcf=-1,steps=-1


    Now please provide your summary.
```

This ends the listing of the Verification Prompts.

C.4 Synthesis Prompts

Similarly to Appendix C.3, this section lists the synthesis prompts. Again prompts are listed in the order in which they are called during the synthesis process.

1. AnalyzeGameLoop is the same as in Appendix C.3.

2. PlanControlStrategy uses the analysis so far and the provided informal guideline to come upw ith a strategy that Angel should play.

```
- role: system
  content: |
    You are given a differential game logic (dGL) loop
    and a desired postcondition that Angel (the controller)
    aims to maintain.

    Your task is to propose a succinct plan describing the
    Angel’s control strategy to guarantee the postcondition.
    You just need to find one control strategy that works,
    not necessarily the optimal or most general one. You may
```

be provided guidelines about what kind of control solution to find. If so, the chose strategy should be in accordance with those guidelines.

Background on dGL:
[Identical to the dGL syntax guide of AnalyzeGameLoop followed
by the Bellerophon Tactic guide of GetTactic.]

You are given:
- The full loop.
- The postcondition that Angel aims to maintain.
- A prior analysis of Angel vs. Demon controls and control modes.
- Optionally, guidelines about what kind of control solution to find.

Goal:
Produce a strategy that maintains safety. If the control involves a Demon controlled loop, carefully account for infinite iterations.

Keep the strategy simple and minimalist. Avoid unnecessarily long or involved control strategies.

It is OK if your strategy requires special initial conditions or assumptions to work, such as starting at specific equilibrium point or assuming the sign of some constants. Just be clear about what these assumptions are.

Requirements for the plan:
- State any initial conditions/assumptions the plan relies on, and what control mode it starts in. You must choose exactly one control mode to start in so as to make subsequent analysis easier. If the control mode requires specific initial conditions, this should be part of your assumptions.
- After this, list Angel’s strategy as a sequence of steps, corresponding to control modes.
  * Each step should start with an action word (e.g., "Choose", "Set", "Switch to"), possibly with a conditional clause (e.g., "When X holds"). End with a temporal indicator (e.g., "repeatedly",

```
          "until Y", "forever").
        * Each step should focus on Angel’s actions. Mention
          Demon only when necessary to motivate Angel’s
          tactics, or when he forces a mode change.
        * Expect to transition between modes mainly by using
          Angel’s controllable actions and choices.
        * Use clear English, no mathematical notation unless
          necessary for clarity. This will later be
          translated into formal constraints.
        * Produce a numbered list of Angel’s decisions.
        * Be as concise as possible without superfluous
          words. Aim for no more than ~120 words.
        * Avoid hedging or verbose meta-commentary; write
          only Angel’s action steps.

    The input includes the game and a prior analysis
    identifying Angel vs. Demon controls. Use that analysis
    to ensure that the plan accounts for any possible
    adversarial behavior.

    - Carefully consider the differential equation. There
      can be coupling between variables that affects your
      reasoning.
    - Based on the dynamics, you may some times need to come
      up with upper/lower bounds of quantities and use that
      to derive sound preconditions. It’s best to use
      simplistic, polynomial bounds so that they verify more
      easily.

    Output Structure:
    - Start with a one-line comment on what new region the
      strategy is trying to cover.
    - Then write section per the requirements listed above.
        * Write a description of initial
          conditions/assumptions, and mode.
        * Then proceed with the plan steps.

    You will be shown zero or more examples of how to
    answer, followed by a new question to answer.
- role: user
content: |-
    Full Game:
    {? A > 0 & B > 0 & kA > 0 & kB > 0 & kC > 0 & kT >= 0 &
    T > 0;^@}; {subgame_c: {{{{{isOn := 0;} ++ {isOn := 1;}}};
```

```
{t := 0;}}}; {subgame_j: {A’ = - A * B * Temp * isOn * kA,
B’ = - A * B * Temp * isOn * kB, C’ = A * B * Temp * isOn *
kC, Temp’ = A * B * Temp * isOn * kT, t’ = 1 & t <= T &
A >= 0 & B >= 0 & C >= 0 & Temp >= 0}
}}*^@}

Postcondition Angel aims to maintain after the loop:
Temp <= Tmax

Prior control analysis:
1) Angel-controlled actions
- None. The only potentially Angelic constructs (the
  choice isOn := 0 ++ isOn := 1 and the ODE duration) are
  inside a loop that is wrapped by a single dual ^@, so
  they are Demon-controlled.

2) Demon-controlled actions
- Initial test {? A > 0 & B > 0 & kA > 0 & kB > 0 & kC > 0 &
  kT >= 0 & T > 0}^@ (subsequent play assumes these hold;
  otherwise Demon would have lost).
- Loop count {subgame_c: {...}*}^@: Demon chooses how many
  iterations (including zero).
- Inside each iteration (because the whole loop is under
  one ^@):
- Branch choice {{{isOn := 0;} ++ {isOn := 1;}}: Demon
  chooses whether the reaction is Off or On for that
  iteration’s flow.
- ODE duration in subgame_j: Demon controls how long to
  flow along
    A’ = - A B Temp isOn kA,
    B’ = - A B Temp isOn kB,
    C’ =   A B Temp isOn kC,
    Temp’ = A B Temp isOn kT,
    t’ = 1
    subject to the evolution domain t <= T, A >= 0, B >= 0,
    C >= 0, Temp >= 0. He may stop anytime before hitting
    the domain boundary; he cannot exceed it.
- Deterministic assignment t := 0 is not a choice (it
  always executes).

3) Control modes (qualitative behaviors and invariants)
- OFF mode (isOn = 0, or equivalently any boundary where
  A = 0 or B = 0 or Temp = 0 makes the vector field zero):
- A’ = B’ = C’ = Temp’ = 0, t’ = 1. Chemical state
```

```
  (A,B,C,Temp) is frozen; only the timer t increases until
  t = T.
- Equilibrium for the chemical variables.
- ON mode (isOn = 1, with A,B,Temp > 0):
- Monotonicity: A’ = −kA A B Temp <= 0, B’ = −kB A B Temp
  <= 0, C’ = kC A B Temp >= 0, Temp’ = kT A B Temp >= 0
  (Temp’ = 0 when kT = 0). Thus A,B decrease; C increases;
  Temp is nondecreasing.
- Domain bounds: A,B,C,Temp remain nonnegative; 0 <= t <= T
  within each flow.
- Polynomial invariants (valid in both ON and OFF, hence
  globally):
    - kB A − kA B = const.
    - C + (kC/kA) A = const and C + (kC/kB) B = const.
    - Temp + (kT/kA) A = const and Temp + (kT/kB) B = const
      (these degenerate correctly when kT = 0).
- Equilibria within ON: if A = 0 or B = 0 or Temp = 0 is
  reached, the system becomes stationary (same as OFF for
  the chemical variables).
- Asymptotic/terminal behavior: Trajectories move along the
  invariant line kB A − kA B = const toward the nonnegative
  axes; one reactant may be exhausted (A=0 or B=0) while the
  other remains positive. C accumulates according to the
  invariants; Temp can only increase (if kT>0) or remain
  constant (if kT=0).

Special parameter submode:
- Isothermal ON (kT = 0): Same as ON but Temp’ = 0, so Temp
  is constant; other invariants and monotonicity still hold.

4) Overall control pattern
- Start with a Demon test ensuring A > 0, B > 0, kA,kB,kC
  > 0, kT >= 0, T > 0 (Angel may assume these).
- Then repeat zero or more iterations (Demon loop):
1) Demon chooses isOn \in {0,1}.
2) Deterministically reset t := 0.
3) Demon evolves the ODE for any duration up to the domain
   boundary (notably t <= T), stopping at will.
- By sequencing iterations, Demon effectively applies a
  piecewise-constant ON/OFF control with segment durations
  at most T, steering the monotone reaction dynamics
  (consume A,B and produce C, increase Temp if kT>0) or
  freezing them when OFF.
```

```
    More information from previous proof attempts:
    Find the control solution for the situation where the
    reactant runs out before temperature can become too large.

    Provide the control strategy now.
```

3. The next query, which gets the LLM to guess a subvalue for a subgame, has three variants, one each for loops, nondeterministic assignments, and ODEs. We show the loop variant, GuessLoopInvControl, but the others are similar.

```
- role: system
  content: |
    Your task is to guess a loop invariant for a game loop
    given the loop body and the postcondition.

    A loop of the form {\alpha}*^@ executes the body \alpha
    zero or more times (and we must account for every
    possibility). A loop invariant I is a property that:
    1. Holds before the loop starts (precondition implies I)
    2. Is preserved by each iteration of the loop body (if I
       holds before \alpha, then I holds after \alpha)
    3. Combined with loop termination, implies the
       postcondition

    A loop of the form {\alpha}* executes the body \alpha
    zero or more times, and how many times is our choice.
    In this case, the invariant should be such if we run the
    loop enough (finite) times, the postcondition will be
    satisfied.

    While choosing the precondition/invariant, you need to
    consider the essential properties that remain true
    throughout all possible iterations of the loop body,
    considering both angel and demon choices.

    The loop invariant should be typically involves bounds,
    safety properties, or relationships between variables

    Return your answer as a logical formula in KeYmaera X
    syntax:
    - Use & for logical AND
    - Use | for logical OR
    - Use ! for logical NOT
    - Use = for equality
    - Use <, <=, >, >= for comparisons
```

```
    - Use parentheses for grouping
    Use only variables you see in the problem. For example,
    if the problem has `x`, use `x`, not `x(0)`.

    Provide up to 4 lines describing your reasoning. Then
    after a blank line, write a code block with the loop
    invariant in KeYmaera X syntax.

    Some syntax instructions to keep in mind are:
    - Do not add comments to your code blocks, as this may
      cause parsing issues.

    Some information about dGL:
    "@" is the dual symbol that flips who controls the action
    between Angel and Demon.

    For example:
    In game "x := *", Angel gets to choose what value is
    assigned.
    The weakest precondition for such an assignment with
    postcondition P is the formula that holds *for some*
    value that could be assigned to x.

    In game "{x := *;}^@", any real number can be assigned,
    per Demon's decision.
    The weakest precondition for such an assignment with
    postcondition P is the formula that holds *for all*
    possible values that could be assigned to x.

    In game "{{ x:= *;}^@}^@" the control flips twice, so
    Angel gets to choose the assignment to x.

    - Carefully consider the differential equation. There
      can be coupling between variables that affects your
      reasoning.
    - Based on the dynamics, you may some times need to come
      up with upper/lower bounds of quantities and use that
      to derive sound preconditions. It's best to use
      simplistic, polynomial bounds so that they verify more
      easily.
- role: user
  content: |
    Loop body:
    {isOn := 0; ++ isOn := 1;}^@; t := 0; {A' = - A * B *
```

```
Temp * isOn
* kA, B’ = - A * B * Temp * isOn * kB, C’ = A * B * Temp *
isOn * kC, Temp’ = A *
B * Temp * isOn * kT, t’ = 1 & t <= T & A >= 0 & B >= 0 &
C >= 0 & Temp >= 0}^@

Postcondition:
Temp <= Tmax

Overall game in which this loop appears (for context):
? A > 0 & B > 0 & kA > 0 & kB > 0 & kC > 0 & kT >= 0 &
T > 0;^@; {{isOn := 0; ++
isOn := 1;}; t := 0; {A’ = - A * B * Temp * isOn * kA,
B’ = - A * B * Temp * isOn
* kB, C’ = A * B * Temp * isOn * kC, Temp’ = A * B * Temp
* isOn * kT, t’ = 1 & t
<= T & A >= 0 & B >= 0 & C >= 0 & Temp >= 0}}*^@

Here is an analysis of the game, the strategy that Angel
is trying to play, and a log of things that you have
already tried in the past.
Carefully consider this information.

Analysis of game:
1) Angel-controlled actions
- None. All controllable constructs in the body lie
  inside a single ^@ (the outer loop dual), so their
  control is flipped to Demon. There are no remaining
  Angel decisions.

2) Demon-controlled actions
- Initial test {? A > 0 & B > 0 & kA > 0 & kB > 0 & kC > 0 &
  kT >= 0 & T > 0}^@: Demon test. Angel may assume these
  hold initially; otherwise she already wins.
- Loop {subgame_c; subgame_j}*^@: Demon loop. Demon
  decides how many iterations to execute (including
  zero).
- Within each iteration:
- Choice isOn := 0 ++ isOn := 1: Demon chooses whether
  the reaction is off or on during this iteration’s flow
  (the ++ is inside the loop’s ^@).
- ODE duration in subgame_j: Demon controls how long to
  flow subject to the evolution domain (t <= T,
  A,B,C,Temp >= 0). He may stop immediately or any time
```

before hitting the boundary. - Deterministic assignments (no player control once chosen to execute):

- t := 0 each iteration (resets the local clock)
- The chosen assignment to isOn is deterministic once the branch is selected.

3) Control modes (qualitative behavior)

- OFF mode (isOn = 0):
- Dynamics: A′ = B′ = C′ = Temp′ = 0, t′ = 1. State (A,B,C,Temp) is constant while time advances to at most T.
- Equilibria: Entire state subspace is an equilibrium for the state variables.

- ON mode (isOn = 1):
- Dynamics: A′ = -kA A B Temp <= 0, B′ = -kB A B Temp <= 0, C′ = kC A B Temp >= 0, Temp′ = kT A B Temp >= 0 (with kT >= 0). If any of A=0 or B=0 or Temp=0, then A′=B′=C′=Temp′=0 (equilibrium).
- Monotonicity (when A,B,Temp > 0): A and B strictly decrease; C and Temp strictly increase (Temp is constant if kT = 0).
- Global linear invariants (valid in both ON and OFF, since OFF has zero derivatives):
  - kC A + kA C = const
  - kC B + kB C = const
  - Temp - (kT/kC) C = const
- Bounds and asymptotes:

- A(t), B(t) remain >= 0 and decrease to 0 at most.
- From invariants, C is bounded above by
  C <= min(C0 + (kC/kA) A0, C0 + (kC/kB) B0).
- Temp follows Temp = Temp0 + (kT/kC)(C - C0), hence
  Temp <= min(Temp0 + (kT/kA) A0, Temp0 + (kT/kB) B0).
- Reaction effectively ceases upon reaching any equilibrium surface A=0 or B=0 or Temp=0; otherwise Demon may stop earlier at t = T. There are no oscillations; trajectories move toward these saturating bounds.
- One-dimensional reduction:
  Using the invariants, C′ becomes a cubic polynomial in C with roots at the limiting
  values where A or B hits zero (and a nonphysical root at negative C from Temp - (kT/kC)C

```
    if kT>0), confirming growth of C to a finite
    saturation.

Special sub-modes:
- Temperature-constant case (kT = 0): Temp’ = 0; Temp is
  invariant. A,B decrease and C increases until A=0 or
  B=0 or Demon stops at t=T.

4) Overall control pattern
- Initial Demon test enforces positive parameters and
  T > 0.
- The game proceeds through Demon-controlled iterations:
1) Demon sets isOn \in {0,1}.
2) Reset t := 0.
3) Demon lets the ODE evolve for some duration up to T
   within the nonnegativity domain (or stops immediately).
- Demon also decides how many such iterations occur.
- Angel has no direct control; Demon’s only lever is to
  toggle the reaction ON/OFF and pick
durations/number of pulses. The state evolves along the
linear invariant manifolds,
with OFF holding the state and ON pushing it monotonically
toward reagent depletion
(A or B) with bounded increases in C and Temp.

Angel control strategy:
Covering the region where reactant depletion caps
temperature growth before hitting Tmax.

Initial conditions/assumptions and starting mode:
- Assume Demon’s test holds: A>0, B>0, kA,kB,kC>0, kT>=0,
  T>0.
- Start in ON mode (isOn=1) for worst-case analysis of Temp
  growth.
- Safety margin:
  Temp0 + min((kT/kA).A0, (kT/kB).B0) <= Tmax.
  (Special
    case: if kT=0, it suffices that Temp0 <= Tmax.)

Angel’s strategy:
1) Do nothing;
permit any Demon choices of ON/OFF, durations, and
iterations, forever. The invariants
cap Temp by Temp0 + min((kT/kA)A0, (kT/kB)B0), and reagent
```

```
    depletion (A or B→0) stops
    further increase, ensuring Temp <= Tmax throughout.

    Starting precondition computation.

    What is a suitable loop invariant for this loop?
```

4. GetNextActionCtrl is the query that decides what to do next: prove the current guess with the verification pipeline, or backtrack, and if so, to where?

```
- role: system
  content: |-
    You are trying to assign the preconditions of a
    differential game logic game.
    You have been proposing preconditions using backwards
    reasoning where you first proposed something for subgames
    closest to the end and then worked backwards recursively.
    Now you have reached a cross-roads and must choose a next
    action.
    You are given the original program, a log of what you
    were doing last, and the options of actions available to
    you.
    Reply with a brief explanation, followed by a newline,
    followed by a code block containing only the word for the
    action.

    ## Background information

    Here is some information about dGL.

    In differential game logic (dGL), there are two players:
    - **Angel**: The controllable player who makes choices we
      can control. Angel constructs include:
      - Angel choices: `a ++ b` (we choose between a and b)
      - Angel loops: `{a}*` (we choose how many iterations)
      - Angel tests: `?P` (we can proceed if P holds,
        otherwise we immediately lose)
      - Angel ODEs: `{x'=f(x) & Q}` (we control the duration)
      - Angel nondeterministic assignments: `x := *`
        (we choose the value)

    There is the dual operator `^@` that switches Angel and
    Demon roles in a game construct.
    Duals can be nested arbitrarily, so carefully check
    nesting levels to determine which player controls each
```

```
action.
Being within an odd number of `^@` operators indicates
Demon control, while an even number (including zero)
indicates Angel control.
For example, in the game `{(a ++ b)^@; (c++d)}^@`,
the choice between `a` and `b` is controlled by Angel,
being within two `^@` operators, while the choice
between `c` and `d` is controlled by Demon.
Using this notation, we go over the Demon constructs
below, indicating that they are Demon-controlled by
wrapping them in a single `^@` operator.

- **Demon**: The adversarial player who makes choices we
  cannot control. Demon has all the same constructs as
  Angel.
  - Demon choices: `{a ++ b}^@` (adversary chooses between
    a and b)
  - Demon loops: `{a*}^@` (adversary chooses how many
    iterations)
  - Demon tests: `{?P}^@` (adversary can proceed if P
    holds, and immediately loses otherwise. So Angel can
    *assume* in subsequent game play that P held at this
    point; otherwise she would already have won.)
  - Demon ODEs: `{x'=f(x) & Q}^@` (adversary controls the
    duration)
  - Demon nondeterministic assignments: `{x := *}^@`
     (adversary chooses the value)

Additionally, the non-controlled operations are:
- Assignment: `x := e` (deterministic assignment like in
  regular programs)
- Sequential composition: `a; b` (first play a, then b)

## Notes
- ODEs can sometimes introduce unmatched single quotes,
  so account for that when parsing.
- In syntax such as { a++b; x:=* }*^@, the `^@` applies to
  the entire loop including its body. In this example,
  Demon controls the choice within the loop between a and
  b, as well as the assignment to x, because both are
  inside one `^@`.
Some subgames are labeled with unique identifiers with
the following syntax:
    <subgame_id>: <subgame>
```

These subgame ids will help make your answers more specific.
A *precondition* for a given game is a logical formula that, when it is true before the game starts, there exists a strategy for Angel to ensure that the *postcondition* holds after the game ends, regardless of how Demon plays.

Input

1. The current subgame for which you were trying to propose preconditions.
2. overall_game: The overall game in which the current subgame appears.
3. Analysis of the game and the control strategy Angel is trying to execute that the chosen preconditions should support.

This is followed by a log of previous attempts at identifying preconditions for subgames and verifying whether they hold by stepping through the game and checking if the preconditions compose to ensure the postcondition is satisfied. Examine this history to figure out what just happened. Did you propose a bad ODE precondition? Did the precondition you proposed end up not being compatible with an earlier chosen loop invariant? Did you come up with a good invariant but you think it can be improved upon?

4. The options you can choose for your next action.

Possible Actions

The preconditions that you proposed most recently didn’t check out.
Either it was not provably a precondition of the current subgame, or if part of a loop, your provided preconditions led to a precondition for the loop body that was not implied by the loop invariant.

The error could lie in the most recent condition, or earlier in the process where you may have made a choice previously that backed you into a corner for this subgame, or maybe the condition is correct but too difficult for the automatic theorem prover to check. Now

~~~~
    you must decide till what point to backtrack, or whether
    to try harder to prove the current subgame result using a
    tactic with KeYmaera X.

    You are given the original program, in which some points
    in the game are labeled as subgame_id.
    - `backtrack-to:id`: Backtrack to the subgame with the
      given id and propose a new precondition there. If you
      choose this option, in your reply, write
      `backtrack-to:id` where id is the id of the subgame you
      want to backtrack to. When backtracking, you can even
      choose the current subgame to retry. Loops are
      considered to occur "later" than their body, so if you
      backtrack to a subgame inside a loop, you won’t be
      asked for a new loop invariant and the existing
      invariant will be used.
    - `try-proof`: Try to prove the current subgame using
      KeYmaera X with the precondition you proposed most
      recently. If you aren’t sure why the most recent
      precondition was bad, this could be a good option
      because the manual proof attempt can give you feedback
      on what’s wrong with the precondition.

    ## Output

    Write a few lines of explanation, followed by a newline,
    and then a code block containing the action you choose.
    For example, a valid code block is:

    ```
    try-proof
    ```

    I will now show you zero of more examples of how to
    respond, followed by the current case.
  - role: user
   content: |-
    The current subgame is:
    subgame_p: {x’ = a * x - b * x * y, y’ = d * x * y - g *
    y}^@


    The overall game with labels is:
    {? x > 0 & y > 0 & a > 0 & b > 0 & d > 0 & g > 0 & x >=
~~~~

```
xmin & y >= ymin;^@}; {subgame_c: {{{{{{{subgame_j:
xadd := *;^@
}; {? xadd >= 0;^@}}; {subgame_l: yadd := *;^@
}}; {? yadd >= 0;^@}}; {x := x + xadd;}}; {y := y +
yadd;}}; {subgame_p: {x' = a * x - b * x * y, y' = d *
x * y - g * y}
}}*^@}

Here is an analysis of the game followed by logs of
precondition computation and checking so far:
Analysis of game:
1) Angel-controlled actions
- ODE duration (subgame_p): In each iteration, Angel
  controls how long to flow along x' = a x - b x y, y' =
  d x y - g y (no evolution domain constraint, so any
  nonnegative duration, including 0).

2) Demon-controlled actions
- Initial test: {? x > 0 & y > 0 & a > 0 & b > 0 & d > 0 &
  g > 0 & x >= xmin & y >= ymin}^@. Angel may assume these
  hold initially; otherwise Demon would have already lost.
- Looping: {...}*^@. Demon chooses how many iterations to
  perform (finite but unbounded, including zero).
- Nondeterministic increments with constraints:
  - subgame_j: {xadd := *}^@ followed by {? xadd >= 0}^@:
    Demon chooses xadd and must satisfy xadd >= 0.
  - subgame_l: {yadd := *}^@ followed by {? yadd >= 0}^@:
    Demon chooses yadd and must satisfy yadd >= 0.
- Deterministic updates x := x + xadd; y := y + yadd then
  apply each iteration.

3) Control modes (qualitative dynamics)
The hybrid iteration has two distinct phases per round:

A. Discrete jump (Demon): x := x + xadd, y := y + yadd
with xadd,yadd >= 0
- Monotone, upward-only resets of both coordinates. Demon
  can make x,y arbitrarily large in a single iteration.
- Effect on the continuous invariant (see below): H(x,y)
  = g ln x - d x + a ln y - b y has gradient
  - \delta H/\delta x = g/x - d, positive if x < g/d
    and negative if x > g/d.
  - \delta H/\delta y = a/y - b, positive if y < a/b
    and negative if y > a/b.
```

So increasing a coordinate raises H when that coordinate is below its nullcline threshold, and lowers H when above. Demon’s nonnegative jumps can thus move the state to higher or lower H-level sets depending on where (x,y) is relative to (g/d, a/b).

B. Continuous Lotka{Volterra flow (Angel-controlled duration):
- Nullclines:
  - x′ = 0 when y = a/b (horizontal line).
  - y′ = 0 when x = g/d (vertical line).
- Unique interior equilibrium: E* = (g/d, a/b).
- First integral (conserved along the ODE): H(x,y) = g ln x - d x + a ln y - b y = constant. Orbits are closed, encircling E*; no asymptotic convergence or divergence within the ODE phase.
- Sign structure by regions (x>0,y>0 always preserved):
  - y < a/b, x > g/d: x′ > 0, y′ > 0 (both grow).
  - y > a/b, x > g/d: x′ < 0, y′ > 0.
  - y > a/b, x < g/d: x′ < 0, y′ < 0 (both decay).
  - y < a/b, x < g/d: x′ > 0, y′ < 0.

  Angel can stop the flow at any point on the current H-level set (including immediately with 0 time).
- Simple bounds during any flow (using positivity and a,b,d,g>0):
  - x′ = x(a - b y) <= a x \implies x(t) <= x0 e^{a t}.
  - y′ = y(d x - g) >= -g y \implies y(t) >= y0 e^{-g t}.

  Positivity is invariant: x(t), y(t) stay > 0 for all flow times.

Special submodes within the ODE:
- Exact equilibrium: If (x,y) = (g/d, a/b), then x′ = y′ = 0 (rest).
- Oscillation: Otherwise, neutral cycles on H-level sets around E*; Angel chooses how far to traverse each cycle segment by choosing the duration.

4) Overall control pattern
- Start: Demon test establishes initial positivity and parameter bounds.
- Iteration (repeats a Demon-chosen number of times):
  1) Demon picks xadd >= 0 (subgame_j) and yadd >= 0 (subgame_l) and applies the discrete increases to x,y.
  2) Angel chooses a duration to flow along the

```
     Lotka{Volterra ODE (subgame_p), moving along the
     current closed orbit (conserved H).
- Interaction:
  - Demon’s lever: choose when to repeat and how large the
    nonnegative jumps are, thereby reselecting the H-level
    set before each flow.
  - Angel’s lever: select the stopping point along the
    chosen H-level set each iteration (including skipping
    the flow by choosing zero duration).
  - Because Demon can pick unbounded xadd,yadd in any
    iteration, x and/or y can be made arbitrarily large;
    the ODE phase then oscillates around E* on the newly
    determined level set until Angel stops.

Angel control strategy:
Covering the \freeze the flow" region so Demon’s
nonnegative jumps alone preserve safety.

Initial conditions/assumptions and start mode:
- Assume the initial test holds: x>0, y>0, a,b,d,g>0, and
  x>=xmin, y>=ymin.
- Start and remain in Immediate-stop ODE mode (Angel always
  ends the flow instantly).

Angel’s strategy:
1) Choose zero ODE duration immediately whenever the
   Lotka{Volterra flow is offered, ending the continuous
   phase instantly, repeatedly.
2) Repeat step 1 after every Demon discrete increment and
   update, forever (or until Demon stops the loop).

Rationale: Discrete steps only increase x and y
(xadd,yadd >= 0). By never flowing, x and y never
decrease. Since the initial test ensures x>=xmin and
y>=ymin, these inequalities remain true throughout all
iterations.

Starting precondition computation.
Proposed the following loop invariant for loop
{xadd := *;^@; ? xadd >= 0;^@; yadd := *;^@;
 ? yadd >= 0;^@; x := x + xadd; y := y + yadd;
 {x’ = a * x - b * x * y, y’ = d * x * y - g * y}}*^@.
x >= xmin & y >= ymin
Attempting to check invariant by precondition computation
```

```
on loop body.
Proposed precondition for subgame {x’ = a * x - b * x * y,
y’ = d * x * y - g * y}^@ and postcondition x >= xmin &
y >= ymin.
Precondition or invariant: x >= xmin & y >= ymin.
Check succeeded: False.
Feedback: The formula check failed, indicating it is
wrong, or too complex for the current level of
automation. Please revise your guess or try a more
detailed proof tactic.

The options you can choose are:
[’try-proof’, ’backtrack-to:p’, ’backtrack-to:c’]
```

Appendix D

Proofs

Lemma 5.1. *I^{opt} is a controllable invariant and it is optimal in the sense that $\vDash I \to I^{\text{opt}}$ for any controllable invariant I.*

Proof. Let us first prove that $I^{\text{opt}} \equiv [((\cap_i \mathsf{act}_i)\,;\,\mathsf{plant})^*]\,\mathsf{safe}$ is a controllable invariant. First, note that axiom [*] derives

$$\vdash I^{\text{opt}} \leftrightarrow \mathsf{safe} \land [(\cap_i \mathsf{act}_i)\,;\,\mathsf{plant}]\, I^{\text{opt}} \tag{D.1}$$

because that is $\vdash I^{\text{opt}} \leftrightarrow \mathsf{safe} \land [(\cap_i \mathsf{act}_i)\,;\,\mathsf{plant}]\,[((\cap_i \mathsf{act}_i)\,;\,\mathsf{plant})^*]\,\mathsf{safe}$.

Safety $\vdash I^{\text{opt}} \to \mathsf{safe}$ derives from Eq. (D.1) propositionally.

Controllable invariance $\vdash I^{\text{opt}} \to \bigvee_i [\mathsf{act}_i\,;\,\mathsf{plant}]\, I^{\text{opt}}$ derives from Eq. (D.1):

$$\dfrac{\mathrm{WL},[;]\;\dfrac{[\cap]\;\dfrac{\mathrm{id}\;\dfrac{*}{\bigvee_i [\mathsf{act}_i]\,[\mathsf{plant}]\, I^{\text{opt}} \vdash \bigvee_i [\mathsf{act}_i]\,[\mathsf{plant}]\, I^{\text{opt}}}}{[\cap_i \mathsf{act}_i]\,[\mathsf{plant}]\, I^{\text{opt}} \vdash \bigvee_i [\mathsf{act}_i]\,[\mathsf{plant}]\, I^{\text{opt}}}}{\mathsf{safe} \land [(\cap_i \mathsf{act}_i)\,;\,\mathsf{plant}]\, I^{\text{opt}} \vdash \bigvee_i [\mathsf{act}_i\,;\,\mathsf{plant}]\, I^{\text{opt}}}}{I^{\text{opt}} \vdash \bigvee_i [\mathsf{act}_i\,;\,\mathsf{plant}]\, I^{\text{opt}}}$$

This concludes the proof that I^{opt} is a controllable invariant. Let us now prove that I^{opt} is optimal. That is, let us consider a controllable invariant I and derive $\vdash I \to I^{\text{opt}}$:

$$\dfrac{\mathrm{loop}\;\dfrac{\mathrm{id}\;\dfrac{*}{I \vdash I} \qquad [;],[\cap],[;]\;\dfrac{I \vdash \bigvee_i [\mathsf{act}_i\,;\,\mathsf{plant}]\, I}{I \vdash [(\cap_i \mathsf{act}_i)\,;\,\mathsf{plant}]\, I} \qquad I \vdash \mathsf{safe}}{I \vdash [((\cap_i \mathsf{act}_i)\,;\,\mathsf{plant})^*]\,\mathsf{safe}}}{\vdash I \to I^{\text{opt}}}$$

The two remaining premises are the two parts of the definition of I being a controllable invariant. This concludes the proof. □ □

Lemma 5.2. *If I is a controllable invariant, then $(I, \mathcal{G}(I))$ is a solution (Def. 5.1).*

Proof. Let us assume that I is a controllable invariant and prove that (I, G) is a solution with $G \equiv \mathcal{G}(I)$. We first need to prove that $\mathsf{assum} \land I$ is an invariant for $\mathsf{prob}(I, G)$. Since assum

is already assumed to be an invariant, it is enough to prove that I is an invariant itself. I holds initially since $I \to I$ is valid and it implies safe by definition of a controllable invariant. Preservation holds by the definition of G:

$$\text{[;],[}\cup\text{],[?],}\wedge\text{R}\;\dfrac{\dfrac{\text{id}\;\dfrac{*}{I, G_i \vdash G_i}}{I, G_i \vdash [\mathsf{act}_i \,;\, \mathsf{plant}]\, I}}{\vdash I \to [\cup_i\, (?G_i \,;\, \mathsf{act}_i) \,;\, \mathsf{plant}]\, I}$$

where the axioms [;],[$\cup$],[?] are used to unpack and repack the games leading to one conjunct for each action (treated separately via $\wedge$R). We also need to prove that an action is always available, which holds by virtue of I being a controllable invariant:

$$\dfrac{\vdash I \to \bigvee_i\, [\mathsf{act}_i \,;\, \mathsf{plant}]\, I}{\vdash I \to \bigvee_i\, G_i}$$

$\square$ $\square$

Lemma 5.3. *If (I, G) is a solution, then $I' \equiv (\mathsf{assum} \wedge I)$ is a controllable invariant. Moreover, we have $(I, G) \sqsubseteq (I', \mathcal{G}(I'))$.*

Proof. Consider a solution (I, G). We prove that $I' \equiv (\mathsf{assum} \wedge I)$ is a controllable invariant. Per Def. 5.1, I' is an invariant for $\mathsf{prob}(I, G)$ and so $\vDash I' \to \mathsf{safe}$. Also, we have the following derivation which repacks games via axioms [$\cup$],[?],[;] using their equivalences:

$$\text{cut}\;\dfrac{I' \vdash \bigvee_i G_i \qquad \vee\text{L,M}\;\dfrac{\to\text{R,[?],[;]}\;\dfrac{\text{M}\;\dfrac{[\cup]\;\dfrac{I' \vdash [\cup_i (?G_i \,;\, \mathsf{act}_i) \,;\, \mathsf{plant}]\, I'}{I' \vdash \bigwedge_i\, [?G_i \,;\, \mathsf{act}_i \,;\, \mathsf{plant}]\, I'}}{I' \vdash [?G_i \,;\, \mathsf{act}_i \,;\, \mathsf{plant}]\, I'}}{I', G_i \vdash [\mathsf{act}_i \,;\, \mathsf{plant}]\, I'}}{I', \bigvee_i G_i \vdash \bigvee_i [\mathsf{act}_i \,;\, \mathsf{plant}]\, I'}}{I' \vdash \bigvee_i [\mathsf{act}_i \,;\, \mathsf{plant}]\, I'}$$

where the open premises are part of the definition of (I, G) being a solution according to Def. 5.1. Let us now prove that $(I, G) \sqsubseteq (I', \mathcal{G}(I'))$. Trivially, we have $\vDash \mathsf{assum} \to (I \to (\mathsf{assum} \wedge I))$. Let us now derive $\vDash \mathsf{assum} \wedge I \to \bigwedge_i (G_i \to \mathcal{G}(I')_i)$:

$$\dfrac{[\cup]\text{,[?],[;]}\;\dfrac{\mathsf{assum} \wedge I \vdash [(\cup_i\, (?G_i \,;\, \mathsf{act}_i)) \,;\, \mathsf{plant}]\, (\mathsf{assum} \wedge I)}{\mathsf{assum} \wedge I \vdash \bigwedge_i (G_i \to [\mathsf{act}_i \,;\, \mathsf{plant}]\, (\mathsf{assum} \wedge I))}}{\mathsf{assum} \wedge I \vdash \bigwedge_i (G_i \to \mathcal{G}(I')_i)}$$

where the remaining premise is part of the definition of (I, G) being a solution. This concludes the proof. $\square$

$\square$

Lemma 5.4. *If I and I' are controllable invariants, then $(I, \mathcal{G}(I)) \sqsubseteq (I', \mathcal{G}(I'))$ if and only if $\vDash \mathsf{assum} \to (I \to I')$.*

Proof. Let us first assume that $\mathsf{assum} \vDash I \rightarrow I'$ and prove that $(I, \mathcal{G}(I)) \sqsubseteq (I', \mathcal{G}(I'))$. It remains to show either $\mathsf{assum} \vDash \neg(I' \rightarrow I)$ or $(\mathsf{assum} \wedge I) \rightarrow \bigwedge_i (G_i \rightarrow G'_i)$ is valid. We show that $\mathsf{assum} \vDash I \rightarrow (\mathcal{G}(I)_i \rightarrow \mathcal{G}(I')_i)$ for all i. To do so, we leverage the fact that assum is an invariant.

$$\dfrac{\text{[*]}\ \dfrac{[]\wedge,\mathrm{M}\ \dfrac{\dfrac{*}{\mathsf{assum},\ I \vdash I'}}{[\mathsf{act}_i\,;\,\mathsf{plant}]\,\mathsf{assum},\ [\mathsf{act}_i\,;\,\mathsf{plant}]\,I \vdash [\mathsf{act}_i\,;\,\mathsf{plant}]\,I'}}{\mathsf{assum},\ [\mathsf{act}_i\,;\,\mathsf{plant}]\,I \vdash [\mathsf{act}_i\,;\,\mathsf{plant}]\,I'}}{\mathsf{assum} \vdash I \rightarrow (\mathcal{G}(I)_i \rightarrow \mathcal{G}(I')_i)}$$

The reverse direction follows trivially from the definition of $\sqsubseteq$. $\square$ $\square$

Theorem 5.1. *$S^{\,\mathrm{opt}}$ is an optimal solution (i.e. a maximum w.r.t. $\sqsubseteq$) of Def. 5.1.*

Proof. We have $S^{\,\mathrm{opt}} \equiv (I^{\,\mathrm{opt}}, \mathcal{G}(I^{\,\mathrm{opt}}))$. From Lemma 5.1 and Lemma 5.2, $S^{\,\mathrm{opt}}$ is a solution. Let (I, G) be another solution. From Lemma 5.3, there exists a controllable invariant I' such that $(I, G) \sqsubseteq (I', \mathcal{G}(I'))$. Then, from Lemma 5.4 and from the optimality of $I^{\,\mathrm{opt}}$ (Lemma 5.1), we have $(I', \mathcal{G}(I')) \sqsubseteq (I^{\,\mathrm{opt}}, \mathcal{G}(I^{\,\mathrm{opt}}))$. By transitivity, $(I, G) \sqsubseteq S^{\,\mathrm{opt}}$. This concludes the proof. $\square$ $\square$

Lemma 5.5. *If act_i is permanent, $\vDash [(\mathsf{act}_i\,;\,\mathsf{plant})^*]\,\mathsf{safe} \leftrightarrow [\mathsf{act}_i\,;\,\mathsf{plant}_\infty]\,\mathsf{safe}$.*

Proof. We first prove that $(\mathsf{act}_i\,;\,\mathsf{plant})^n \equiv (\mathsf{act}_i\,;\,\mathsf{plant}^n)$ by induction on $n \geq 1$. The base case is trivial. Regarding the induction case, we have

$$\begin{aligned}
(\mathsf{act}_i\,;\,\mathsf{plant})^{n+1} &\equiv (\mathsf{act}_i\,;\,\mathsf{plant}\,;\,(\mathsf{act}_i\,;\,\mathsf{plant})^n) \\
&\equiv (\mathsf{act}_i\,;\,\mathsf{plant}\,;\,\mathsf{act}_i\,;\,\mathsf{plant}^n) \\
&\equiv (\mathsf{act}_i\,;\,\mathsf{plant}\,;\,\mathsf{act}_i\,;\,\mathsf{plant}^n) \\
&\equiv (\mathsf{act}_i\,;\,\mathsf{plant}\,;\,\mathsf{plant}^n) \\
&\equiv (\mathsf{act}_i\,;\,\mathsf{plant}^{n+1}).
\end{aligned}$$

From this, we get $(\mathsf{act}_i\,;\,\mathsf{plant})^* \equiv ?\mathrm{true} \cup (\mathsf{act}_i\,;\,\mathsf{plant}^*)$ from the semantics of loops in dGL. Thus, we have $\vDash [(\mathsf{act}_i\,;\,\mathsf{plant})^*]\,\mathsf{safe} \;\leftrightarrow\; \mathsf{safe} \wedge [\mathsf{act}_i\,;\,\mathsf{plant}_\infty]\,\mathsf{safe}$ since t does not appear free in safe. From this, we prove our theorem by noting that $\mathsf{safe} \wedge [\mathsf{act}_i\,;\,\mathsf{plant}_\infty]\,\mathsf{safe} \leftrightarrow [\mathsf{act}_i\,;\,\mathsf{plant}_\infty]\,\mathsf{safe}$ since act_i cannot write any variable that appears in safe. $\square$ $\square$

Theorem 5.2. *I^0 is a controllable invariant.*

Proof. Trivially, we have $\vDash I^0 \rightarrow \mathsf{safe}$. More interestingly, let us prove that $I^0 \rightarrow \vee_i\,[\alpha_i]\,I^0$ where $\alpha_i \equiv (\mathsf{act}_i\,;\,\mathsf{plant})$. The proof crucially leverages the permanence assumption via the identity $\vDash I^0 \leftrightarrow \vee_{i\in\mathsf{P}} [\alpha_i{}^*]\,\mathsf{safe}$.

$$\begin{prooftree}
\AxiomC{$*$}
\LeftLabel{[*]}
\UnaryInfC{$[\alpha_i{}^*]\,\mathsf{safe} \vdash [\alpha_i]\,[\alpha_i{}^*]\,\mathsf{safe}$}
\LeftLabel{M[$\cdot$]}
\UnaryInfC{$[\alpha_i{}^*]\,\mathsf{safe} \vdash [\alpha_i]\,(\vee_{j\in\mathsf{P}}\,[\alpha_j{}^*]\,\mathsf{safe})$}
\UnaryInfC{$[\alpha_i{}^*]\,\mathsf{safe} \vdash [\alpha_i]\,I^0$}
\LeftLabel{$\vee$L,M[$\cdot$]}
\UnaryInfC{$\vee_{i\in\mathsf{P}}\,[\alpha_i{}^*]\,\mathsf{safe} \vdash \vee_{i\in\mathsf{P}}\,[\alpha_i]\,I^0$}
\UnaryInfC{$I^0 \vdash \vee_{i\in\mathsf{P}}\,[\alpha_i]\,I^0$}
\LeftLabel{WR}
\UnaryInfC{$I^0 \vdash \vee_i\,[\alpha_i]\,I^0$}
\end{prooftree}$$

□ □

Theorem 5.3. *I^n is a controllable invariant for all $n \geq 0$.*

Proof. We proceed by induction on n. The base case is covered by Theorem 5.2. Assume that I^n is a controllable invariant and prove that I^{n+1} is one also. Abbreviate $\alpha_i \equiv (\mathsf{act}_i\,;\,\mathsf{plant})$. Without loss of generality, assume that all actions are permanent since non-permanent actions play no role in computing I^n. The hard part is in proving that $\vDash I^n \rightarrow \vee_i\,[\alpha_i]\,I^{n+1}$.

$$\begin{prooftree}
\AxiomC{$I^n \vdash \vee_i\,[\alpha_i]\,I^n$}
\AxiomC{$\vdash I^n \rightarrow I^{n+1}$}
\LeftLabel{M[$\cdot$]}
\BinaryInfC{$I^n \vdash \vee_i\,[\alpha_i]\,I^{n+1}$}
\AxiomC{$[\mathsf{step}]\,I^n \vdash \vee_i\,[\alpha_i]\,I^{n+1}$}
\LeftLabel{$\vee$L}
\BinaryInfC{$I^n \vee [\mathsf{step}]\,I^n \vdash \vee_i\,[\alpha_i]\,I^{n+1}$}
\UnaryInfC{$I^{n+1} \vdash \vee_i\,[\alpha_i]\,I^{n+1}$}
\end{prooftree}$$

The first premise is a consequence of (I^n, G^n) being a solution (our induction hypothesis) and the second one is a trivial consequence of the definition of I^{n+1}. We can now focus on proving the last premise.

To do so, it is useful to introduce the following predicate:

$$R(a,b) \;\equiv\; [\mathsf{plant}_b]\,(\mathsf{safe} \wedge (t \geq a \rightarrow I^n))$$

Intuitively, $R(a,b)$ is true if following the dynamics leads to reaching I^n within time interval $[a,b]$ while being safe the whole time. Using this predicate, we can reformulate $[\mathsf{step}]\,I^n$ as follows:

$$[\mathsf{step}]\,I^n \;\equiv\; \vee_i\,\exists\theta\!\geq\!0\,[\mathsf{act}_i]\,R(\theta,\theta+T). \tag{D.2}$$

In addition, Lemma 5.6 gives us the following key property of R:

$$\vdash c \leq b \wedge R(a,b) \rightarrow [\mathsf{plant}_c]\,R(a-t,b-t). \tag{D.3}$$

We can now complete the proof using Eq. (D.2):

$$\begin{prooftree}
\AxiomC{$\Gamma \vdash [\alpha_i]\,(t \leq \theta \rightarrow I^{n+1})$}
\AxiomC{$\Gamma \vdash [\alpha_i]\,(t \geq \theta \rightarrow I^{n+1})$}
\BinaryInfC{$\theta \geq 0,\; [\mathsf{act}_i]\,R(\theta,\theta+T) \vdash [\alpha_i]\,I^{n+1}$}
\LeftLabel{$\vee$R,$\vee$L}
\UnaryInfC{$\vee_i\,\exists\theta\!\geq\!0\,[\mathsf{act}_i]\,R(\theta,\theta+T) \vdash \vee_i\,[\alpha_i]\,I^{n+1}$}
\UnaryInfC{$[\mathsf{step}]\,I^n \vdash \vee_i\,[\alpha_i]\,I^{n+1}$}
\end{prooftree}$$

where we abbreviate $\Gamma \equiv \theta \geq 0, [\mathsf{act}_i]\, R(\theta, \theta + T)$. In the case where $t \geq \theta$ after a control cycle, the agent has reached I^n and therefore I^{n+1}:

$$\dfrac{\dfrac{\dfrac{*}{\Gamma \vdash [\mathsf{act}_i]\, R_i(\theta, \theta + T)}}{\Gamma \vdash [\mathsf{act}_i]\, [\mathsf{plant}_{\theta+T}]\, (\mathsf{safe} \wedge (t \geq \theta \rightarrow I^n))}}{{}^{[;]}\Gamma \vdash [\mathsf{act}_i \,;\, \mathsf{plant}]\, (t \geq \theta \rightarrow I^{n+1})}$$

In the case where $t \leq \theta$ after a control cycle, the agent must perform the same action again with a timeout of $\theta - t$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\Gamma \vdash [\mathsf{act}_i]\, R(\theta, \theta + T)}}{\Gamma \vdash [\mathsf{act}_i \,;\, \mathsf{plant}]\, R(\theta - t, \theta - t + T)}}{\Gamma \vdash [\mathsf{act}_i \,;\, \mathsf{plant} \,;\, \mathsf{act}_i]\, R(\theta - t, \theta - t + T)}}{\Gamma \vdash [\alpha_i]\, (t \leq \theta \rightarrow [\mathsf{act}_i]\, R(\theta - t, \theta - t + T))}}{\Gamma \vdash [\alpha_i]\, (t \leq \theta \rightarrow (\exists \rho \leq 0\, [\mathsf{act}_i]\, R(\rho, \rho + T)))}}{\Gamma \vdash [\alpha_i]\, (t \leq \theta \rightarrow I^{n+1})}$$

This concludes the proof. □ □

Lemma 5.6. *Consider a property of the form $R(a, b) \equiv [\alpha_b]\, (S \wedge (t \geq a \rightarrow I))$ with $\alpha_b \equiv (t := 0 \,;\, \{x' = f(x), t' = 1 \,\&\, Q \wedge t \leq b\})$. Then this formula is valid:*

$$c \leq b \wedge R(a, b) \rightarrow [\alpha_c]\, R(a - t, b - t).$$

Proof. This follows from the semantics of dL since all involved differential equations are the same and $t \leq c \leq b$ is the duration that passes during α_c, and thus explaining the offset of $-t$ on the time interval arguments of $R(a, b)$ □. □

Lemma 5.7. *If $I^n \leftrightarrow I^{n+1}$ is valid for some $n \geq 0$, then $I^n \leftrightarrow I^m$ is valid for all $m \geq n$ and $I^n \leftrightarrow I^\omega$ is valid where $I^\omega \equiv [\mathsf{step}^\times \,;\, \mathsf{forever}]\, \mathsf{safe}$.*

Proof. The first part is simply a case of a recursive sequence $I^{n+1} \equiv F(I^n)$ reaching a fixpoint ($F(I) \equiv I \vee [\mathsf{step}]\, I$). Let us then prove the $\vDash I^n \leftrightarrow I^\omega$ equivalence, or rather the nontrivial direction $\vDash I^\omega \rightarrow I^n$. From $\vDash I^n \leftrightarrow I^{n+1}$, we get $\vDash I^n \leftrightarrow I^n \vee [\mathsf{step}]\, I^n$ and so $\vDash [\mathsf{step}]\, I^n \rightarrow I^n$. In addition, by the monotonicity of $(I^n)_n$, we have $\vDash I^0 \rightarrow I^n$. The rest follows from the FP$^\times$ rule:

$$\dfrac{{}^{\mathrm{FP}^\times}\dfrac{{}^{\vee\mathrm{L}}\dfrac{I^0 \vdash I^n \qquad [\mathsf{step}]\, I^n \vdash I^n}{I^0 \vee [\mathsf{step}]\, I^n \vdash I^n}}{[\mathsf{step}^\times]\, I^0 \vdash I^n}}{I^\omega \vdash I^n}$$

□ □

Theorem 5.4. *Consider a finite number of discrete dL programs α_i such that $\vDash \langle \alpha_i \rangle$ true for all i and $p \equiv \{x' = f(x) \,\&\, q \geq 0\}$. Then, provided that $\mathsf{best}_j((\alpha_i)_i, p, m)$ and $\mathsf{best}_j((\alpha_i)_i, p, -q)$ (no other action stops earlier because of the domain constraint), we have:*

$$\vDash \langle ((\cap\, \alpha_i) \,;\, p)^* \rangle\, m \leq 0 \leftrightarrow \langle (\alpha_j \,;\, p)^* \rangle\, m \leq 0 \ .$$

Proof. The nontrivial implication to prove is:

$$\vDash \langle (\alpha_j \, ; \, p)^* \rangle \, m \le 0 \; \rightarrow \; \langle ((\cap \, \alpha_i) \, ; \, p)^* \rangle \, m \le 0.$$

We do so by proving:

$$\Gamma \vDash \langle ({\alpha_j}^{(1)} \, ; \, p^{(1)})^* \rangle \, m^{(1)} \le 0 \; \rightarrow \; \langle ((\cap_i {\alpha_i}^{(2)}) \, ; \, p^{(2)})^* \rangle \, m^{(2)} \le 0, \tag{D.4}$$

where $\Gamma \equiv \bigwedge_{x \in V} (x^{(1)} = x^{(2)})$ and $V \equiv \mathrm{BV}(p) \cup \bigcup_i \mathrm{BV}(\alpha_i)$. To prove Eq. (D.4), we chain together three implications:

1. $\Gamma \vDash \langle ({\alpha_j}^{(1)} \, ; \, p^{(1)})^* \rangle \, m^{(1)} \le 0 \; \rightarrow \; \langle ({\alpha_j}^{(1)} \, ; \, (\cap_i {\alpha_i}^{(2)}) \, ; \, p^{(1,2)})^* \rangle \, m^{(1)} \le 0$:
    (a) To prove the implication above, we consider a sequence of states $s_1 \ldots s_n$ such that $s_1 \in [\![\Gamma]\!]$, $s_n \in [\![m^{(1)} \le 0]\!]$ and $(s_i, s_{i+1} \in [\![{\alpha_j}^{(1)} \, ; \, p^{(1)}]\!])$ for all i. We must prove that $s_1 \in \langle ({\alpha_j}^{(1)} \, ; \, (\cap_i {\alpha_i}^{(2)}) \, ; \, p^{(1,2)})^* \rangle \, m^{(1)} \le 0$.
    (b) We say that two states s and s' are *1-equivalent* (written $s \sim_{(1)} s'$) if they only differ on variables tagged with 2. Using this definition, it is enough to prove the following fact: for all $i \le n$ and any state s such that $s \sim_{(1)} s_i$ and $s \in [\![q^{(1)} \le q^{(2)}]\!]$, we have $s \in \langle ({\alpha_j}^{(1)} \, ; \, (\cap_i {\alpha_i}^{(2)}) \, ; \, p^{(1,2)})^* \rangle \, m^{(1)} \le 0$.
    (c) We prove the fact above by descending induction on i. The base case for $i = n$ follows from our assumption on s_n. The inductive case considers a state s such that $s \sim_{(1)} s_{i-1}$ and $s \in [\![q^{(1)} \le q^{(2)}]\!]$. By assumption, $(s_{i-1}, s_i) \in [\![{\alpha_j}^{(1)} \, ; \, p^{(1)}]\!]$. Thus, there exists s' such that $(s, s') \in [\![{\alpha_j}^{(1)} \, ; \, (\cap_i {\alpha_i}^{(2)}) \, ; \, p^{(1,2)}]\!]$, $s' \sim_{(1)} s_i$ and $s' \in [\![q^{(1)} \le q^{(2)}]\!]$. Note that the assumption that $\mathsf{best}_j((\alpha_i)_i, p, -q)$ is critical in establishing the existence of s', by ensuring that $p^{(1,2)}$ can be run for at least as long as $p^{(1)}$. We conclude by using the induction hypothesis on s'.
2. $\Gamma \vDash \langle \gamma \rangle \, m^{(1)} \le 0 \rightarrow \langle \gamma \rangle \, m^{(2)} \le 0$ with $\gamma \equiv ({\alpha_j}^{(1)} \, ; \, (\cap_i {\alpha_i}^{(2)}) \, ; \, p^{(1,2)})^*$:
    (a) The key fact we are using here is that for any game β and formulas P, Q, we have $\vDash [\beta^{-\mathsf{d}}] \, (P \rightarrow Q) \rightarrow \langle \beta \rangle \, P \rightarrow \langle \beta \rangle \, Q$ where $\beta^{-\mathsf{d}}$ is obtained from β by removing all applications of the dual operator. Intuitively, this is true since $[\beta^{-\mathsf{d}}] \, (P \rightarrow Q)$ ensures that $P \rightarrow Q$ is true in every reachable game state, independently of both players' strategies.
    (b) Per the theorem assumption, we have $\Gamma \vDash [\gamma^{-\mathsf{d}}] \, (m^{(1)} \ge m^{(2)})$. Using the monotonicity rule M, we obtain $\Gamma \vDash [\gamma^{-\mathsf{d}}] \, (P \rightarrow Q)$ with $P \equiv m^{(2)} \le 0$ and $Q \equiv m^{(1)} \le 0$. We can then conclude using the previous point.
3. $\vDash \langle ({\alpha_j}^{(1)} \, ; \, (\cap_i {\alpha_i}^{(2)}) \, ; \, p^{(1,2)})^* \rangle \, m^{(2)} \le 0 \; \rightarrow \; \langle ((\cap_i {\alpha_i}^{(2)}) \, ; \, p^{(2)})^* \rangle \, m^{(2)} \le 0$:
    (a) This last implication can be proved in a similar way than (1.), although the $\mathsf{best}_j((\alpha_i)_i, p, -q)$ assumption is not needed since we are removing ODE domain constraints instead of adding them.

By chaining everything, we get:

$$\Gamma \vDash \langle ({\alpha_j}^{(1)} \, ; \, p^{(1)})^* \rangle \, m^{(1)} \le 0 \; \rightarrow \; \langle ((\cap_i {\alpha_i}^{(2)}) \, ; \, p^{(2)})^* \rangle \, m^{(2)} \le 0,$$

which closes the proof. □ □

Lemma 6.1 (Game Suffixes are Operational Gameplay Suffixes)**.** *For subgame $b{:}\beta$ in overall game $a{:}\alpha$, let the game suffix be $a{:}\alpha_{b:}$. Let σ be some state. The states reachable from playing $a{:}\alpha$ such that σ is reached at subgame $b{:}\beta$ at some point during the gameplay are the same as the states reachable from playing $a{:}\alpha_{b:}$ starting in σ. That is,*

$$\{\pi_{\neg\mathfrak{S}}(s)\ :\ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma'), \lfloor p \rfloor_{\sigma'} = \sigma \text{ and } \pi_{\neg\mathfrak{S}}(s) \neq ()\} = \pi_{\neg\mathfrak{S}}(\mathcal{T}(a{:}\alpha_{b:}, \sigma)),$$

where $()$ is the empty path.

Proof. We prove

$$\{\pi_{\neg\mathfrak{S}}(s)\ :\ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma'), \lfloor p \rfloor_{\sigma'} = \sigma \text{ and } \pi_{\neg\mathfrak{S}}(s) \neq ()\} = \pi_{\neg\mathfrak{S}}(\mathcal{T}(a{:}\alpha_{b:}, \sigma))$$

by induction on the structure of $a{:}\alpha$, and case analysis.

- When $a{:}\alpha$ is atomic and not controlled by Angel, i.e., $\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x)\ \&\ Q\}^d\}$, then the only subgame b in $\mathsf{subgames}(a{:}\alpha)$ is $a{:}\alpha$ itself. Thus the only possibility is $b = a$. $a{:}\alpha_{a:}$ is the game $a{:}\alpha$. $\mathcal{T}(a{:}\alpha_{a:}, \sigma)$ is then $\mathcal{T}(a{:}\alpha, \sigma)$.
  The left-hand side, $\pi_{\neg\mathfrak{S}}(\mathcal{T}(a{:}\alpha, \sigma))$, is $\mathcal{T}(\alpha, \sigma)$, the label-free game tree of the original game (see Def. B.4). Per the definition of labeled game trees for any atomic subgame α (Def. B.2), $\mathcal{T}(a{:}\alpha, \sigma)$ is $a \cdot t\ :\ t \in \mathcal{T}(\alpha, \sigma)$. Thus, the right-hand side is also $\mathcal{T}(\alpha, \sigma)$.
- Extending the previous case to compositional subgames, when $b = a$ and $a{:}\alpha$ is not a loop, then a similar argument holds.
  The *left-hand side* is $\pi_{\neg\mathfrak{S}}(\mathcal{T}(a{:}\alpha, \sigma))$.
  The *right-hand side* is $\{\pi_{\neg\mathfrak{S}}(s)\ :\ p \cdot a \cdot s \in \mathcal{T}(a{:}\alpha, \sigma'), \pi_{\neg\mathfrak{S}}(s) \neq ()$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$. Per the definition of (non-loop) labeled game trees, the only occurrence of label a is at the beginning of all plays and thus the right-hand side is equal to $\{\pi_{\neg\mathfrak{S}}(s)\ :\ a \cdot s \in \mathcal{T}(a{:}\alpha, \sigma), \pi_{\neg\mathfrak{S}}(s) \neq ()\}$. This is just $\pi_{\neg\mathfrak{S}}(\mathcal{T}(a{:}\alpha, \sigma))$.
- When $a{:}\alpha$ is a loop, i.e., $a{:}(g{:}\gamma)^*$ or $a{:}(g{:}\gamma)^\times$, and $b \neq a$, then $a{:}\alpha_{b:} = (g{:}\gamma_{b:})\ ;\ a{:}\alpha$.
  The *left-hand side* is then $\pi_{\neg\mathfrak{S}}(\mathcal{T}(g{:}\gamma_{b:}\ ;\ a{:}\alpha, \sigma))$.
  Per the inductive hypothesis, $\pi_{\neg\mathfrak{S}}(\mathcal{T}(g{:}\gamma_{b:}, \sigma))$ is equal to $\{\pi_{\neg\mathfrak{S}}(s)\ :\ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma')$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$.
  Thus, per the definition of labeled game trees for sequential composition games, the left-hand side is $\{\pi_{\neg\mathfrak{S}}(s)\ :\ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma')$ and $\lfloor p \rfloor_{\sigma'} = \sigma\} \cup \{\pi_{\neg\mathfrak{S}}(s \cdot t)\ :\ p \cdot b \cdot s \in \mathsf{leaf}(\mathcal{T}(g{:}\gamma, \sigma')), \lfloor p \rfloor_{\sigma'} = \sigma, \lfloor s \rfloor_{\sigma} = \sigma''$ and $t \in \mathcal{T}(a{:}\alpha, \sigma'')\}$, where leaf returns the set of leaf nodes in a tree as discussed in Appendix B.6.2.
  The *right-hand side* is $\{\pi_{\neg\mathfrak{S}}(s)\ :\ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma'), \pi_{\neg\mathfrak{S}}(s) \neq ()$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$. Per the definition of labeled game trees for loops, we can break this down by unrolling the loop: $\{\pi_{\neg\mathfrak{S}}(s)\ :\ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma')$ and $\lfloor p \rfloor_{\sigma'} = \sigma\} \cup \{\pi_{\neg\mathfrak{S}}(s \cdot t)\ :\ p \cdot b \cdot s \in \mathsf{leaf}(\mathcal{T}(g{:}\gamma, \sigma')), \lfloor p \rfloor_{\sigma'} = \sigma, \lfloor s \rfloor_{\sigma} = \sigma''$ and $t \in \mathcal{T}(a{:}\alpha, \sigma'')\}$.
- Otherwise, when $b = a$ and $a{:}\alpha$ is a loop, i.e., $a{:}(g{:}\gamma)^*$ or $a{:}(g{:}\gamma)^\times$, then $a{:}\alpha_{a:} = a{:}\alpha$.
  The *left-hand side* is $\pi_{\neg\mathfrak{S}}(\mathcal{T}(a{:}\alpha, \sigma))$.
  The *right-hand side* is $\{\pi_{\neg\mathfrak{S}}(s)\ :\ p \cdot a \cdot s \in \mathcal{T}(a{:}\alpha, \sigma'), \pi_{\neg\mathfrak{S}}(s) \neq ()$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$. By the choice of $\sigma' = \sigma$, $p = ()$ and the definition of labeled game trees, we can show that this is a superset of $\pi_{\neg\mathfrak{S}}(\mathcal{T}(a{:}\alpha, \sigma))$. Moreover, by the definition of labeled game trees

for loops, we know that any suffix s that follows label a must be generated by the loop fixed point function f from Def. B.2 for initial state σ. So, $a \cdot s$ must be within the game tree $\mathcal{T}(a{:}\alpha, \sigma)$. Therefore, the right-hand side is also a subset of $\pi_{\neg\mathfrak{S}}(\mathcal{T}(a{:}\alpha, \sigma))$. Thus, the left-hand side and right-hand side must be equal.

- When $a{:}\alpha$ is a choice, i.e., $a{:}(g{:}\gamma \cup d{:}\delta)$ or $a{:}(g{:}\gamma \cap d{:}\delta)$, and $b \neq a$, then there are two cases.
    1. If $b \in \mathsf{subgames}(g{:}\gamma)$, then $a{:}\alpha_{b:} = g{:}\gamma_{b:}$.
       The *left-hand side* is $\pi_{\neg\mathfrak{S}}(\mathcal{T}(g{:}\gamma_{b:}, \sigma))$. Per the inductive hypothesis, this is $\{\pi_{\neg\mathfrak{S}}(s) \ : \ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma'), \pi_{\neg\mathfrak{S}}(s) \neq ()$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$.
       The *right-hand side* is $\{\pi_{\neg\mathfrak{S}}(s) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma'), \pi_{\neg\mathfrak{S}}(s) \neq ()$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$. Per the definition of labeled game trees for choices, $\mathcal{T}(a{:}\alpha, \sigma')$ is the union of the game trees of $g{:}\gamma$ and $d{:}\delta$. However, b does not appear in $d{:}\delta$. Thus the right-hand side can be simplified to $\{\pi_{\neg\mathfrak{S}}(s) \ : \ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma'), \pi_{\neg\mathfrak{S}}(s) \neq ()$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$.
    2. If $b \in \mathsf{subgames}(d{:}\delta)$, then $a{:}\alpha_{b:} = d{:}\delta_{b:}$, and a symmetric argument applies, with $g{:}\gamma$ and $d{:}\delta$ swapped.
- When $a{:}\alpha$ is a sequential composition, i.e., $a{:}(g{:}\gamma \,;\, d{:}\delta)$, and $b \neq a$, then there are two cases.
    1. If $b \in \mathsf{subgames}(g{:}\gamma)$, then $a{:}\alpha_{b:} = (g{:}\gamma_{b:}) \,;\, d{:}\delta$. The argument is similar to the loop case.
       The *left-hand side* is $\pi_{\neg\mathfrak{S}}(\mathcal{T}((g{:}\gamma_{b:}) \,;\, d{:}\delta, \sigma))$.
       As per the inductive hypothesis, $\pi_{\neg\mathfrak{S}}(\mathcal{T}(g{:}\gamma_{b:}, \sigma))$ is equal to $\{\pi_{\neg\mathfrak{S}}(s) \ : \ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma'), \pi_{\neg\mathfrak{S}}(s) \neq ()$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$.
       Thus, per the definition of labeled game trees for sequential composition games, the left-hand side is $\{\pi_{\neg\mathfrak{S}}(s) \ : \ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma'), \lfloor p \rfloor_{\sigma'} = \sigma, \pi_{\neg\mathfrak{S}}(s) \neq ()\} \cup \{\pi_{\neg\mathfrak{S}}(s \cdot t) \ : \ p \cdot b \cdot s \in \mathsf{leaf}(\mathcal{T}(g{:}\gamma, \sigma')), t \in \mathcal{T}(d{:}\delta, \sigma''), \lfloor p \rfloor_{\sigma'} = \sigma, \pi_{\neg\mathfrak{S}}(s \cdot t) \neq ()$ and $\lfloor s \rfloor_{\sigma} = \sigma''\}$.
       The *right-hand side* is $\{\pi_{\neg\mathfrak{S}}(s) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma'), \pi_{\neg\mathfrak{S}}(s) \neq ()$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$. Per the definition of labeled game trees for sequential composition, since $b \notin \mathsf{subgames}(d{:}\delta)$, this is: $\{\pi_{\neg\mathfrak{S}}(s) \ : \ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma'), \lfloor p \rfloor_{\sigma'} = \sigma, \pi_{\neg\mathfrak{S}}(s) \neq ()\} \cup \{\pi_{\neg\mathfrak{S}}(s \cdot t) \ : \ p \cdot b \cdot s \in \mathsf{leaf}(\mathcal{T}(g{:}\gamma, \sigma')), t \in \mathcal{T}(d{:}\delta, \sigma''), \lfloor p \rfloor_{\sigma'} = \sigma, \pi_{\neg\mathfrak{S}}(s \cdot t) \neq ()$ and $\lfloor s \rfloor_{\sigma} = \sigma''\}$.
    2. If $b \in \mathsf{subgames}(d{:}\delta)$, then $a{:}\alpha_{b:} = d{:}\delta_{b:}$.
       The *left-hand side* is $\pi_{\neg\mathfrak{S}}(\mathcal{T}(d{:}\delta_{b:}, \sigma))$.
       Per the inductive hypothesis, this is $\{\pi_{\neg\mathfrak{S}}(s) \ : \ p \cdot b \cdot s \in \mathcal{T}(d{:}\delta, \sigma'), \pi_{\neg\mathfrak{S}}(s) \neq ()$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$.
       The *right-hand side* is $\{\pi_{\neg\mathfrak{S}}(s) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma'), \pi_{\neg\mathfrak{S}}(s) \neq ()$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$. Per the definition of labeled game trees for sequential composition, using the fact that $b \notin \mathsf{subgames}(g{:}\gamma)$, this can be simplified to $\{\pi_{\neg\mathfrak{S}}(s) \ : \ p \cdot b \cdot s \in \mathcal{T}(d{:}\delta, \sigma')$ and $\lfloor p \rfloor_{\sigma'} = \sigma\}$.

$\square$

Lemma 6.2 (Execution Prefixes are Original Gameplay Prefixes)**.** *For subgame $b{:}\beta$ in $a{:}\alpha$, for any initial state σ, the states reachable at $b{:}\beta$ while playing $a{:}\alpha$ are equal to those reachable by* *playing game prefix* $a{:}\alpha_{:b}$. *That is, prefix set* $\{\pi_{\neg\mathfrak{G}}(p) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma)\}$ *is equal, under prefix closure, to prefix game tree* $\pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha_{:b}, \sigma))$[1].

Proof. We show that

$$\pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha_{:b}, \sigma)) = \{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma) \text{ and } p = t \cdot u\}$$

Similar to Lemma 6.1, we perform induction on the structure of $a{:}\alpha$ and case analysis.

- When $b = a$ and $a{:}\alpha$ is not a loop, then $a{:}\alpha_{:a}$ is the empty game skip.
  The *left-hand side* $\pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha_{:a}, \sigma))$ is then $\emptyset$.
  Indeed, on the *right-hand side*, all plays in $\mathcal{T}(a{:}\alpha, \sigma)$ are of the form $a \cdot p$ where p does not contain a, so the condition of the theorem holds.
- When $a{:}\alpha$ is atomic and not controlled by Angel, i.e., $\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x) \ \& \ Q\}^d\}$, then the only subgame b in $\mathsf{subgames}(a{:}\alpha)$ is $a{:}\alpha$ itself. Thus the only possibility is $b = a$, handled already by the first case.
- When $a{:}\alpha$ is a choice, i.e., $a{:}(g{:}\gamma \cup d{:}\delta)$ or $a{:}(g{:}\gamma \cap d{:}\delta)$, then we analyze cases depending on the value of $b \neq a$.
  1. If b is in $g{:}\gamma$, then $a{:}\alpha_{:b} = g{:}\gamma_{:b}$.
     The *left-hand side* is $\pi_{\neg\mathfrak{G}}(\mathcal{T}(g{:}\gamma_{:b}, \sigma))$. Per the inductive hypothesis, this consists of

     $$\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}.$$

     The *right-hand side* is $\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}$. Per the definition of game trees for choices, since $\mathcal{T}(a{:}\alpha, \sigma)$ is the union of the game trees for $g{:}\gamma$ and $d{:}\delta$, and b does not appear in $d{:}\delta$, this simplifies to

     $$\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}.$$

  2. If b is in $d{:}\delta$, then the argument is symmetric to the previous case, but $g{:}\gamma$ swapped with $d{:}\delta$.
- When $a{:}\alpha$ is a sequential composition, i.e., $a{:}(g{:}\gamma \, ; \, d{:}\delta)$, then we analyze cases depending on the value of $b \neq a$.
  1. If b is in $g{:}\gamma$, then $a{:}\alpha_{:b} = g{:}\gamma_{:b}$.
     The *left-hand side* is $\pi_{\neg\mathfrak{G}}(\mathcal{T}(g{:}\gamma_{:b}, \sigma))$. Per the inductive hypothesis, this is

     $$\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}.$$

     The *right-hand side* is $\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}$. Since $b \notin \mathsf{subgames}(d{:}\delta)$, using the definition of labeled game trees for sequential composition, this can be simplified to

     $$\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}.$$

---

[1] The prefix closure is required to recover tree structure. Unlike with the suffix construction, the prefix set builder collects only *parent* subgames of $b{:}\beta$ and *not all predecessors*. Expanding the prefix closure results in the condition: $\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma) \text{ and } p = t \cdot u\} = \pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha_{:b}, \sigma))$

2. If b is in $d{:}\delta$, then $a{:}\alpha_{:b} = g{:}\gamma\,;\, d{:}\delta_{:b}$.
   The *left-hand side* is $\pi_{\neg\mathfrak{G}}(\mathcal{T}(g{:}\gamma\,;\, d{:}\delta_{:b}, \sigma))$.
   As per the inductive hypothesis, $\pi_{\neg\mathfrak{G}}(\mathcal{T}(d{:}\delta_{:b}, \sigma))$ is equal to $\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(d{:}\delta, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}$.
   Thus, per the definition of labeled game trees for sequential composition games, the left-hand side is

   $$\pi_{\neg\mathfrak{G}}(\mathcal{T}(g{:}\gamma, \sigma)) \cup \{\pi_{\neg\mathfrak{G}}(q \cdot t) \ : q \in \mathsf{leaf}(\mathcal{T}(g{:}\gamma, \sigma)), \lfloor q \rfloor_\sigma = \sigma', p \cdot b \cdot s \in \mathcal{T}(d{:}\delta, \sigma'), p = t \cdot u\}$$

   where leaf indicates a leaf node of a game tree, as indicated in Appendix B.6.2.
   The *right-hand side* is $\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma)\}, p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()$.
   Per the definition of labeled game trees for sequential composition, since
   $b \notin \mathsf{subgames}(g{:}\gamma)$, this can be written as

   $$\pi_{\neg\mathfrak{G}}(\mathcal{T}(g{:}\gamma, \sigma)) \cup \{\pi_{\neg\mathfrak{G}}(q \cdot t) \ : q \in \mathsf{leaf}(\mathcal{T}(g{:}\gamma, \sigma)), \lfloor q \rfloor_\sigma = \sigma', p \cdot b \cdot s \in \mathcal{T}(d{:}\delta, \sigma'), p = t \cdot u\}.$$

- When $a{:}\alpha$ is a loop, i.e., $a{:}(g{:}\gamma)^*$ or $a{:}(g{:}\gamma)^\times$, then we analyze cases depending on the value of b.
  1. If $b = a$, then $a{:}\alpha_{:b} = a{:}\alpha$.
     The *left-hand side* is then $\pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha, \sigma))$.
     The *right-hand side* is $\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot a \cdot s \in \mathcal{T}(a{:}\alpha, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}$. By the definition of labeled game trees (Def. B.2), all plays in $\mathcal{T}(a{:}\alpha, \sigma)$ end with $a \cdot \mathfrak{s}^d$ or $a \cdot \mathfrak{s}$. Thus, from the elements where play s is $\mathfrak{s}$ or $\mathfrak{s}^d$, $\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot a \cdot s \in \mathcal{T}(a{:}\alpha, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}$ is a superset of $\pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha, \sigma))$. Moreover, as labeled game trees are prefix-closed, $\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot a \cdot s \in \mathcal{T}(a{:}\alpha, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}$ is also a subset of $\pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha, \sigma))$. Thus, the left-hand side and right-hand side must be equal.
  2. If $b \neq a$, then $a{:}\alpha_{:b} = (a{:}\alpha\,;\, g{:}\gamma_{:b})$.
     The *left-hand side* is

     $$\pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha\,;\, (g{:}\gamma_{:b}), \sigma)).$$

     As per the inductive hypothesis, $\pi_{\neg\mathfrak{G}}(\mathcal{T}(g{:}\gamma_{:b}, \sigma))$ is equal to $\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}$. Thus, per the definition of labeled game trees for sequential composition games, the left-hand side is

     $$\pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha, \sigma)) \cup \{\pi_{\neg\mathfrak{G}}(q \cdot t) \ : q \in \mathsf{leaf}(\mathcal{T}(a{:}\alpha, \sigma)), \lfloor q \rfloor_\sigma = \sigma', p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma'), p = t \cdot u\}.$$

     The *right-hand side* is $\{\pi_{\neg\mathfrak{G}}(t) \ : \ p \cdot b \cdot s \in \mathcal{T}(a{:}\alpha, \sigma), p = t \cdot u, \pi_{\neg\mathfrak{G}}(t) \neq ()\}$. This is a subset of $\pi_{\neg\mathfrak{G}}(\mathcal{T}(a{:}\alpha, \sigma))$, and therefore the left-hand side, because game trees are prefix-closed. Next we show it is also a superset. Consider any play in the left-hand side set. For Angelic loops, it must be a prefix of some play of the form $q \cdot p$ where $q \in \mathsf{leaf}(f^n(a\,\hat{}\,\mathfrak{s}, a\,\hat{}\,\mathfrak{g}))$ for some n, $\lfloor q \rfloor_\sigma = \sigma'$, and $p \cdot b \cdot s \in \mathcal{T}(g{:}\gamma, \sigma')$ where f

and $\hat{}$ are from Def. B.2. Such a play and all its prefixes feature in the right-hand side set because they are in $f^{n+1}(a\,\hat{}\,\mathfrak{s}, a\,\hat{}\,\mathfrak{g})$ which is in $\mathcal{T}(a{:}\alpha, \sigma)$. For Demonic loops, the same argument applies but for $f^{n}(a\,\hat{}\,\mathfrak{s}^{d}, a\,\hat{}\,\mathfrak{g}^{d})$ as defined in Def. B.2.

□

Lemma 6.3 (Existential projection correspondence)**.** *For any Angelic subvalue map S for game $a{:}\alpha$ and compatible winning condition ϕ, in initial state $\sigma \models \langle \alpha \rangle\, \phi$, Angel has a winning strategy for the game $a{:}\alpha \mathbin{\exists\!\!\downarrow} S$ if and only if Angel can win $\langle \alpha \rangle\, \phi$ while following the policy of S. That is, $\sigma \models \langle a{:}\alpha \mathbin{\exists\!\!\downarrow} S \rangle\, \phi$ iff $\mathcal{S}_{a:\alpha}(S)(\sigma) \neq \emptyset$.*

Proof. Like Lemma 6.3, we prove a stronger lemma by induction over the structure of $a{:}\alpha$. The induction proves the correspondence statement below for any subvalue map S. Let

$$U_{\alpha,S}(\sigma) = \mathsf{leaf}(\mathcal{T}(a{:}\alpha \mathbin{\forall\!\!\downarrow} S, \sigma))$$

that is, the set of leaves generated by the universal projection, and

$$P_{\alpha,S}(\sigma) = \mathsf{leaf}(\mathcal{S}_{a:\alpha}(S)(\sigma))$$

be the set of policy-compliant leaves generated by following the policy of S from σ.

A leaf $u \in U_{\alpha,S}(\sigma)$ has an *inserted-guard deadlock* when it represents Demon deadlocking on an assertion $!S(c)$ introduced by $a{:}\alpha \mathbin{\forall\!\!\downarrow} S$ to guard what was before projection an Angel-controlled subgame. Inserted-guard deadlocks represent Demon attempting an action that the policy of S does not permit. A deadlock is *genuine* when the first undefined action belongs to a subgame inherited from the original game $a{:}\alpha$, rather than to such an inserted guard.

The strengthened lemma is: there is a relation $R_{\alpha,S,\sigma} \subseteq U_{\alpha,S}(\sigma) \times P_{\alpha,S}(\sigma)$ such that:

1. If $u\, R_{\alpha,S,\sigma}\, p$ and both $\lfloor u \rfloor_{\sigma}$ and $\lfloor p \rfloor_{\sigma}$ are defined, then
$$\lfloor u \rfloor_{\sigma} = \lfloor p \rfloor_{\sigma}.$$
2. If $u\, R_{\alpha,S,\sigma}\, p$ and one of u, p genuinely deadlocks at an Angel-controlled subgame, then the other also genuinely deadlocks at the corresponding Angel-controlled subgame.
3. If $u\, R_{\alpha,S,\sigma}\, p$ and one of u, p genuinely deadlocks at a Demon-controlled subgame, then the other also genuinely deadlocks at the corresponding Demon-controlled subgame.
4. Every $p \in P_{\alpha,S}(\sigma)$ is related to some $u \in U_{\alpha,S}(\sigma)$.
5. Every $u \in U_{\alpha,S}(\sigma)$ is either related to some $p \in P_{\alpha,S}(\sigma)$ or has an inserted-guard deadlock.

Thus the universal projection has exactly the same defined outcomes, the same genuine Angel-deadlock outcomes, and the same genuine Demon-deadlock outcomes as the policy-compliant leaves.

This stronger lemma implies the original lemma because:

(1) If $\sigma \models \langle a{:}\alpha \mathbin{\forall\!\!\downarrow} S \rangle\, \phi$, then every leaf in $U_{\alpha,S}(\sigma)$ either reaches a state satisfying ϕ, genuinely deadlocks at a Demon-controlled subgame, or deadlocks at an inserted guard. By the stronger lemma, there must be no Angelic deadlocks in $P_{\alpha,S}(\sigma)$, nor any leafs reaching a state not satisfying ϕ, because then there would be a related leaf in $U_{\alpha,S}(\sigma)$ with the same property, contradicting $\sigma \models \langle a{:}\alpha \mathbin{\forall\!\!\downarrow} S \rangle\, \phi$. Thus, every way to pursue $\mathcal{S}_{a:\alpha}(S)(\sigma)$ is winning for Angel.

(2) If $\sigma \not\models \langle a{:}\alpha \underset{\downarrow}{\forall} S\rangle\, \phi$, then some leaf in $U_{\alpha,S}(\sigma)$ either reaches a defined state not satisfying ϕ, or genuinely deadlocks at an Angel-controlled subgame. By the stronger lemma, this leaf is related to a leaf in $P_{\alpha,S}(\sigma)$ with the same defined state or the same genuine Angel-deadlock. Thus there is a play in $\mathcal{S}_{a:\alpha}(S)(\sigma)$ that is not winning for Angel.

We now prove the stronger lemma by induction over the structure of $a{:}\alpha$. In the case analysis below, we sometimes omit no-op labels and sequence markers for readability. The relation $R_{\alpha,S,\sigma}$ relates the corresponding concrete leaves with those no-op actions restored.

- If $a{:}\alpha$ is atomic and not controlled by Angel, i.e., $\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x)\ \&\ Q\}^d\}$, then $a{:}\alpha \underset{\downarrow}{\forall} S$ is α. The universal projection has exactly the same game tree as the original game, and the policy generator does not restrict any action because there is no Angel-controlled choice to externalize. $R_{\alpha,S,\sigma}$ can be defined trivially to relate corresponding leaves. Non-deadlocking leaves reach the same state, any genuine deadlock is inherited from the original atomic game with the same owner (i.e., Angel or Demon), and there are no inserted guards.
- If $a{:}\alpha$ is a non-deterministic Angel assignment $a{:}x := *$, then the policy-generated leaves are exactly the actions $(x := e)$ such that $\sigma(x \mapsto e) \models S(\mathsf{end})$. The universal projection is

  $$x := \otimes\,;\ !S(\mathsf{end}).$$

  Its leafs are $(x := e) \cdot (!S(\mathsf{end}))$ for all values e.
  We define $R_{\alpha,S,\sigma}$ by relating each policy leaf $(x := e)$ to the universal projection leaf $(x := e^d)\hat{}(!S(\mathsf{end}))$ with the same value e. Related leaves have the same defined final state because $(x := e)$ and $(x := e^d)$ have the same state transformer and the assertion is a no-op when $\sigma(x \mapsto e) \models S(\mathsf{end})$. Every policy leaf is related to such a universal projection leaf, and every universal projection leaf is either related in this way or deadlocks at the inserted guard $!S(\mathsf{end})$.
- If $a{:}\alpha$ is an Angelic ODE $a{:}\{x' = f(x)\ \&\ Q\}$, then the policy-generated leaves are legal ODE actions

  $$(x' = f(x)\ \&\ Q@t)$$

  whose solution $\varphi : [0, t] \to \mathcal{S}$ starts in σ, stays in Q, follows $x' = f(x)$, and satisfies $\varphi(t) \models S(\mathsf{end})$. The universal projection is

  $$\{x' = f(x)\ \&\ Q\}^d\,;\ !S(\mathsf{end}).$$

  Define $R_{\alpha,S,\sigma}$ by relating each policy leaf $(x' = f(x)\ \&\ Q@t)$ to the universal leaf $\{x' = f(x)\ \&\ Q@t\}^{d}\hat{}!S(\mathsf{end})$ with the same solution φ and time t. Related leaves have the same defined final state because Angelic and Demonic ODE actions have the same transition relation and the assertion is a no-op when it passes. Every policy leaf is related in this way, and every universal projection leaf is either related in this way or deadlocks at the inserted guard $!S(\mathsf{end})$.
- If $a{:}\alpha$ is an Angelic choice $a{:}(g{:}\gamma \cup d{:}\delta)$, then

  $$a{:}\alpha \underset{\downarrow}{\forall} S = (!S(g)\,;\ g{:}\gamma \underset{\downarrow}{\forall} S) \cap (!S(d)\,;\ d{:}\delta \underset{\downarrow}{\forall} S).$$

The policy-compliant leaves are the left-branch leaves $\mathfrak{l}{}^\wedge p$ with $p \in P_{\gamma,S}(\sigma)$ exactly when $\sigma \models S(g)$, and the right-branch leaves $\mathfrak{r}{}^\wedge p$ with $p \in P_{\delta,S}(\sigma)$ exactly when $\sigma \models S(d)$. Define $R_{\alpha,S,\sigma}$ branchwise: if $\sigma \models S(g)$, relate

$$\mathfrak{l}^d{}^\wedge !S(g){}^\wedge u \quad \text{to} \quad \mathfrak{l}{}^\wedge p$$

whenever $u\, R_{\gamma,S,\sigma}\, p$. If $\sigma \models S(d)$, relate the analogous right-branch leaves using $R_{\delta,S,\sigma}$. When the corresponding guard does not hold, the universal leaf stops at $!S(g)$ or $!S(d)$, which is an inserted-guard Demon deadlock. Branch controls and passing guards do not change state, so the required correspondences follow from the induction hypotheses for the branch subgames. Every universal projection leaf is either thus covered by branch reasoning, or deadlocks at a failed branch guard.

- If $a{:}\alpha$ is an Angelic loop $a{:}(g{:}\gamma)^*$: Let S_a abbreviate $S(\mathsf{end} \mapsto S(a))$. Then

  $$a{:}\alpha \underset{\downarrow}{\forall} S = (!S(g)\, ;\, (g{:}\gamma \underset{\downarrow}{\forall} S_a))^\times\, ;\, !S(\mathsf{end}).$$

  We prove the strengthened lemma for this case by an inner induction over finite loop unfoldings. The inner invariant after n unfoldings is:

  1. pending policy leaves $p{}^\wedge\mathfrak{g}$ are matched with pending universal leaves $u{}^\wedge\mathfrak{g}^d$ at the same state, and in that state, $S(g)$ holds;
  2. non-pending matched leaves satisfy the first three clauses of the strengthened lemma;
  3. unmatched universal leaves are inserted-guard deadlocks at $!S(g)$ or $!S(\mathsf{end})$, or inserted-guard deadlocks inherited from the body.

  The base case follows from Def. 6.4: $\mathfrak{g}$ is offered exactly when $S(g)$ holds, and $\mathfrak{s}$ exactly when $S(\mathsf{end})$ holds. The universal projection loop offers both dual actions; if both actions are allowed by the policy, they are matched, while a disallowed go fails at $!S(g)$ and a disallowed stop fails at $!S(\mathsf{end})$, thus following into the inserted-guard deadlocks category. For the inductive step, consider a matched pending pair $p{}^\wedge\mathfrak{g}$ and $u{}^\wedge\mathfrak{g}^d$. Their reached states agree and satisfy $S(g)$, so the universal guard $!S(g)$ passes. Apply the outer induction hypothesis to $g{:}\gamma$ with subvalue map S_a at that state. Body inserted-guard deadlocks remain unmatched inserted-guard deadlocks for the loop, and matched genuine body deadlocks remain genuine deadlocks at the same inherited subgame. Matched defined body leaves reach the same state τ. At τ, policy offers $\mathfrak{s}$ exactly under $S(\mathsf{end})$ and $\mathfrak{g}$ exactly under $S(g)$, while the universal loop offers both dual choices and checks the corresponding guard. Thus allowed stop and go choices are matched, and disallowed ones become inserted-guard deadlocks at $!S(\mathsf{end})$ or the next $!S(g)$.

  Taking the union over $n < \omega$ and applying the policy-side filter $\nexists u(u{}^\wedge\mathfrak{g} \in \mathsf{leaf}(t))$ removes unfinished strategies. The remaining leaves satisfy the outer lemma by the inner invariant.
- If $a{:}\alpha$ is a Demonic loop $a{:}(g{:}\gamma)^\times$, then Again let S_a abbreviate $S(\mathsf{end} \mapsto S(a))$. Then $a{:}\alpha \underset{\downarrow}{\forall} S = (g{:}\gamma \underset{\downarrow}{\forall} S_a)^\times$. Both sides have the same Demonic loop control skeleton: $\mathfrak{s}^d$, $\mathfrak{g}^d$, and, after each defined go-body execution, the next pair of dual loop controls. The only difference is in the loop body, where the leaf set is $P_{\gamma,S_a}(\tau)$ on the policy side and $U_{\gamma,S_a}(\tau)$ on the universal projection side.

Use an inner induction over the finite trees generated by the f^n construction, and then take the maximal unions of increasing chains from Def. B.7 on the policy side. The construction invariant pairs pending $\mathfrak{g}^d$ leaves on the two sides state-for-state, matches non-pending leaves by the first three clauses, and leaves unmatched only inserted-guard deadlocks inherited from the body. The base case is immediate. In the inductive step, take paired pending $\mathfrak{g}^d$ leaves reaching the same state τ. The outer induction hypothesis for $g{:}\gamma$ at τ supplies the body relation. Matched defined body leaves are extended on both sides by the same next loop actions; matched genuine body deadlocks remain genuine deadlocks at the same inherited subgame; unmatched body leaves remain inserted-guard deadlocks. Taking unions of increasing chains preserves the invariant. Maximality removes nonmaximal unfinished trees, preserving the desired outer lemma correspondence because it does not eliminate any genuine leaves that have no successors.

- If $a{:}\alpha$ is a Demonic choice $a{:}(g{:}\gamma \cap d{:}\delta)$, then

  $$a{:}\alpha \mathrel{\underset{\downarrow}{\forall}} S = g{:}\gamma \mathrel{\underset{\downarrow}{\forall}} S \cap d{:}\delta \mathrel{\underset{\downarrow}{\forall}} S.$$

  Define $R_{\alpha,S,\sigma}$ by relating $\mathfrak{l}^d\hat{}\,u$ to $\mathfrak{l}^d\hat{}\,p$ whenever $u\, R_{\gamma,S,\sigma}\, p$, and relating $\mathfrak{r}^d\hat{}\,u$ to $\mathfrak{r}^d\hat{}\,p$ whenever $u\, R_{\delta,S,\sigma}\, p$. Demon branch labels do not change state, so the required correspondence follows from the induction hypotheses applied to the branch subgames.
- If $a{:}\alpha$ is sequential composition $a{:}(g{:}\gamma;\, d{:}\delta)$, let S_d abbreviate $S(\mathsf{end} \mapsto S(d))$. Then

  $$a{:}\alpha \mathrel{\underset{\downarrow}{\forall}} S = (g{:}\gamma \mathrel{\underset{\downarrow}{\forall}} S_d)\ ;\ (d{:}\delta \mathrel{\underset{\downarrow}{\forall}} S).$$

  Policy-compliant leaves are $p_\gamma \in P_{\gamma,S_d}(\sigma)$ when $\lfloor p_\gamma \rfloor_\sigma$ is undefined, and $p_\gamma\hat{}\,p_\delta$ when $\lfloor p_\gamma \rfloor_\sigma = \tau$ and $p_\delta \in P_{\delta,S}(\tau)$.
  Define $R_{\alpha,S,\sigma}$ as follows. First, for p_γ where $\lfloor p_\gamma \rfloor_\sigma$ is undefined, let the leaf u_γ be the leaf such that $u_\gamma\, R_{\gamma,S_d,\sigma}\, p_\gamma$. Then relate u_γ to p_γ in $R_{\alpha,S,\sigma}$. Secondly, for $p_\gamma\hat{}\,p_\delta$, let u_γ be the leaf such that $u_\gamma\, R_{\gamma,S_d,\sigma}\, p_\gamma$, and u_δ be the leaf such that $u_\delta\, R_{\delta,S,\lfloor u_\gamma \rfloor_\sigma}\, p_\delta$. Then relate $u_\gamma\hat{}\,u_\delta$ to $p_\gamma\hat{}\,p_\delta$ in $R_{\alpha,S,\sigma}$.
  Universal projection leaves u_γ that are inserted-guard deadlocks in the first component and universal projection leaves $u_\gamma\hat{}\,u_\delta$ where u_δ is an inserted-guard deadlock in the second component are left unmatched. The required correspondence follows from induction hypotheses for subgames γ and δ, and the definition of sequential execution.

$\square$

Lemma 6.4 (Universal projection correspondence)**.** *For any Angelic subvalue map S for game $a{:}\alpha$, in initial state $\sigma \models \langle\alpha\rangle\, a$, for compatible winning condition ϕ, Angel has a winning strategy for the game $a{:}\alpha \mathrel{\underset{\downarrow}{\forall}} S$ if and only if all ways to pursue the policy of S for game $a{:}\alpha$ that complete the game end in Angel's winning region. That is, $\sigma \models \langle a{:}\alpha \mathrel{\underset{\downarrow}{\forall}} S\rangle\, \phi$ iff all elements of set $\mathcal{S}_{a:\alpha}(S)(\sigma)$ are Angel winning strategies.*

Proof. Like Lemma 6.3, we prove a stronger lemma by induction over the structure of $a{:}\alpha$. The induction proves the correspondence statement below for any subvalue map S. Let

$$U_{\alpha,S}(\sigma) = \mathsf{leaf}(\mathcal{T}(a{:}\alpha \mathrel{\underset{\downarrow}{\forall}} S, \sigma))$$

that is, the set of leaves generated by the universal projection, and

$$P_{\alpha,S}(\sigma) = \mathsf{leaf}(\mathcal{S}_{a:\alpha}(S)(\sigma))$$

be the set of policy-compliant leaves generated by following the policy of S from σ.

A leaf $u \in U_{\alpha,S}(\sigma)$ has an *inserted-guard deadlock* when it represents Demon deadlocking on an assertion $!S(c)$ introduced by $a{:}\alpha \overset{\forall}{\downarrow} S$ to guard what was before projection an Angel-controlled subgame. Inserted-guard deadlocks represent Demon attempting an action that the policy of S does not permit. A deadlock is *genuine* when the first undefined action belongs to a subgame inherited from the original game $a{:}\alpha$, rather than to such an inserted guard.

The strengthened lemma is: there is a relation $R_{\alpha,S,\sigma} \subseteq U_{\alpha,S}(\sigma) \times P_{\alpha,S}(\sigma)$ such that:

1. If $u \, R_{\alpha,S,\sigma} \, p$ and both $\lfloor u \rfloor_\sigma$ and $\lfloor p \rfloor_\sigma$ are defined, then
$$\lfloor u \rfloor_\sigma = \lfloor p \rfloor_\sigma .$$
2. If $u \, R_{\alpha,S,\sigma} \, p$ and one of u, p genuinely deadlocks at an Angel-controlled subgame, then the other also genuinely deadlocks at the corresponding Angel-controlled subgame.
3. If $u \, R_{\alpha,S,\sigma} \, p$ and one of u, p genuinely deadlocks at a Demon-controlled subgame, then the other also genuinely deadlocks at the corresponding Demon-controlled subgame.
4. Every $p \in P_{\alpha,S}(\sigma)$ is related to some $u \in U_{\alpha,S}(\sigma)$.
5. Every $u \in U_{\alpha,S}(\sigma)$ is either related to some $p \in P_{\alpha,S}(\sigma)$ or has an inserted-guard deadlock.

Thus the universal projection has exactly the same defined outcomes, the same genuine Angel-deadlock outcomes, and the same genuine Demon-deadlock outcomes as the policy-compliant leaves.

This stronger lemma implies the original lemma because:

(1) If $\sigma \models \langle a{:}\alpha \overset{\forall}{\downarrow} S \rangle \, \phi$, then every leaf in $U_{\alpha,S}(\sigma)$ either reaches a state satisfying ϕ, genuinely deadlocks at a Demon-controlled subgame, or deadlocks at an inserted guard. By the stronger lemma, there must be no Angelic deadlocks in $P_{\alpha,S}(\sigma)$, nor any leafs reaching a state not satisfying ϕ, because then there would be a related leaf in $U_{\alpha,S}(\sigma)$ with the same property, contradicting $\sigma \models \langle a{:}\alpha \overset{\forall}{\downarrow} S \rangle \, \phi$. Thus, every way to pursue $\mathcal{S}_{a:\alpha}(S)(\sigma)$ is winning for Angel.

(2) If $\sigma \not\models \langle a{:}\alpha \overset{\forall}{\downarrow} S \rangle \, \phi$, then some leaf in $U_{\alpha,S}(\sigma)$ either reaches a defined state not satisfying ϕ, or genuinely deadlocks at an Angel-controlled subgame. By the stronger lemma, this leaf is related to a leaf in $P_{\alpha,S}(\sigma)$ with the same defined state or the same genuine Angel-deadlock. Thus there is a play in $\mathcal{S}_{a:\alpha}(S)(\sigma)$ that is not winning for Angel.

We now prove the stronger lemma by induction over the structure of $a{:}\alpha$. In the case analysis below, we sometimes omit no-op labels and sequence markers for readability. The relation $R_{\alpha,S,\sigma}$ relates the corresponding concrete leaves with those no-op actions restored.

- If $a{:}\alpha$ is atomic and not controlled by Angel, i.e., $\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x) \,\&\, Q\}^d\}$, then $a{:}\alpha \overset{\forall}{\downarrow} S$ is α. The universal projection has exactly the same game tree as the original game, and the policy generator does not restrict any action because there is no Angel-controlled choice to externalize. $R_{\alpha,S,\sigma}$ can be defined trivially to relate corresponding leaves. Non-deadlocking leaves reach the same state, any genuine deadlock is inherited from the original atomic game with the same owner (i.e., Angel or Demon), and there are no inserted guards.

- If $a{:}\alpha$ is a non-deterministic Angel assignment $a{:}x := *$, then the policy-generated leaves are exactly the actions $(x := e)$ such that $\sigma(x \mapsto e) \models S(\mathsf{end})$. The universal projection is
  $$x := \otimes\,;\, !S(\mathsf{end}).$$
  Its leafs are $(x := e) \cdot (!S(\mathsf{end}))$ for all values e.
  We define $R_{\alpha,S,\sigma}$ by relating each policy leaf $(x := e)$ to the universal projection leaf $(x := e^d)\,\hat{}\,(!S(\mathsf{end}))$ with the same value e. Related leaves have the same defined final state because $(x := e)$ and $(x := e^d)$ have the same state transformer and the assertion is a no-op when $\sigma(x \mapsto e) \models S(\mathsf{end})$. Every policy leaf is related to such a universal projection leaf, and every universal projection leaf is either related in this way or deadlocks at the inserted guard $!S(\mathsf{end})$.
- If $a{:}\alpha$ is an Angelic ODE $a{:}\{x' = f(x)\ \&\ Q\}$, then the policy-generated leaves are legal ODE actions
  $$(x' = f(x)\ \&\ Q@t)$$
  whose solution $\varphi : [0,t] \to \mathcal{S}$ starts in σ, stays in Q, follows $x' = f(x)$, and satisfies $\varphi(t) \models S(\mathsf{end})$. The universal projection is
  $$\{x' = f(x)\ \&\ Q\}^d\,;\, !S(\mathsf{end}).$$
  Define $R_{\alpha,S,\sigma}$ by relating each policy leaf $(x' = f(x)\ \&\ Q@t)$ to the universal leaf $\{x' = f(x)\ \&\ Q@t\}^d\,\hat{}\,!S(\mathsf{end})$ with the same solution φ and time t. Related leaves have the same defined final state because Angelic and Demonic ODE actions have the same transition relation and the assertion is a no-op when it passes. Every policy leaf is related in this way, and every universal projection leaf is either related in this way or deadlocks at the inserted guard $!S(\mathsf{end})$.
- If $a{:}\alpha$ is an Angelic choice $a{:}(g{:}\gamma \cup d{:}\delta)$, then
  $$a{:}\alpha \underset{\downarrow}{\forall} S = (!S(g)\,;\, g{:}\gamma \underset{\downarrow}{\forall} S) \cap (!S(d)\,;\, d{:}\delta \underset{\downarrow}{\forall} S).$$
  The policy-compliant leaves are the left-branch leaves $\mathfrak{l}\,\hat{}\,p$ with $p \in P_{\gamma,S}(\sigma)$ exactly when $\sigma \models S(g)$, and the right-branch leaves $\mathfrak{r}\,\hat{}\,p$ with $p \in P_{\delta,S}(\sigma)$ exactly when $\sigma \models S(d)$.
  Define $R_{\alpha,S,\sigma}$ branchwise: if $\sigma \models S(g)$, relate
  $$\mathfrak{l}^d\,\hat{}\,!S(g)\,\hat{}\,u \quad \text{to} \quad \mathfrak{l}\,\hat{}\,p$$
  whenever $u\, R_{\gamma,S,\sigma}\, p$. If $\sigma \models S(d)$, relate the analogous right-branch leaves using $R_{\delta,S,\sigma}$. When the corresponding guard does not hold, the universal leaf stops at $!S(g)$ or $!S(d)$, which is an inserted-guard Demon deadlock. Branch controls and passing guards do not change state, so the required correspondences follow from the induction hypotheses for the branch subgames. Every universal projection leaf is either thus covered by branch reasoning, or deadlocks at a failed branch guard.
- If $a{:}\alpha$ is an Angelic loop $a{:}(g{:}\gamma)^*$: Let S_a abbreviate $S(\mathsf{end} \mapsto S(a))$. Then
  $$a{:}\alpha \underset{\downarrow}{\forall} S = (!S(g)\,;\, (g{:}\gamma \underset{\downarrow}{\forall} S_a))^\times\,;\, !S(\mathsf{end}).$$
  We prove the strengthened lemma for this case by an inner induction over finite loop unfoldings. The inner invariant after n unfoldings is:

1. pending policy leaves $p\,\hat{}\,\mathfrak{g}$ are matched with pending universal leaves $u\,\hat{}\,\mathfrak{g}^d$ at the same state, and in that state, $S(g)$ holds;
2. non-pending matched leaves satisfy the first three clauses of the strengthened lemma;
3. unmatched universal leaves are inserted-guard deadlocks at $!S(g)$ or $!S(\mathsf{end})$, or inserted-guard deadlocks inherited from the body.

The base case follows from Def. 6.4: $\mathfrak{g}$ is offered exactly when $S(g)$ holds, and $\mathfrak{s}$ exactly when $S(\mathsf{end})$ holds. The universal projection loop offers both dual actions; if both actions are allowed by the policy, they are matched, while a disallowed go fails at $!S(g)$ and a disallowed stop fails at $!S(\mathsf{end})$, thus following into the inserted-guard deadlocks category. For the inductive step, consider a matched pending pair $p\,\hat{}\,\mathfrak{g}$ and $u\,\hat{}\,\mathfrak{g}^d$. Their reached states agree and satisfy $S(g)$, so the universal guard $!S(g)$ passes. Apply the outer induction hypothesis to $g{:}\gamma$ with subvalue map S_a at that state. Body inserted-guard deadlocks remain unmatched inserted-guard deadlocks for the loop, and matched genuine body deadlocks remain genuine deadlocks at the same inherited subgame. Matched defined body leaves reach the same state τ. At τ, policy offers $\mathfrak{s}$ exactly under $S(\mathsf{end})$ and $\mathfrak{g}$ exactly under $S(g)$, while the universal loop offers both dual choices and checks the corresponding guard. Thus allowed stop and go choices are matched, and disallowed ones become inserted-guard deadlocks at $!S(\mathsf{end})$ or the next $!S(g)$.

Taking the union over $n < \omega$ and applying the policy-side filter $\nexists u(u\,\hat{}\,\mathfrak{g} \in \mathsf{leaf}(t))$ removes unfinished strategies. The remaining leaves satisfy the outer lemma by the inner invariant.

- If $a{:}\alpha$ is a Demonic loop $a{:}(g{:}\gamma)^{\times}$, then Again let S_a abbreviate $S(\mathsf{end} \mapsto S(a))$. Then $a{:}\alpha \mathbin{\underset{\downarrow}{\forall}} S = (g{:}\gamma \mathbin{\underset{\downarrow}{\forall}} S_a)^{\times}$. Both sides have the same Demonic loop control skeleton: $\mathfrak{s}^d$, $\mathfrak{g}^d$, and, after each defined go-body execution, the next pair of dual loop controls. The only difference is in the loop body, where the leaf set is $P_{\gamma,S_a}(\tau)$ on the policy side and $U_{\gamma,S_a}(\tau)$ on the universal projection side.

  Use an inner induction over the finite trees generated by the f^n construction, and then take the maximal unions of increasing chains from Def. B.7 on the policy side. The construction invariant pairs pending $\mathfrak{g}^d$ leaves on the two sides state-for-state, matches non-pending leaves by the first three clauses, and leaves unmatched only inserted-guard deadlocks inherited from the body. The base case is immediate. In the inductive step, take paired pending $\mathfrak{g}^d$ leaves reaching the same state τ. The outer induction hypothesis for $g{:}\gamma$ at τ supplies the body relation. Matched defined body leaves are extended on both sides by the same next loop actions; matched genuine body deadlocks remain genuine deadlocks at the same inherited subgame; unmatched body leaves remain inserted-guard deadlocks. Taking unions of increasing chains preserves the invariant. Maximality removes nonmaximal unfinished trees, preserving the desired outer lemma correspondence because it does not eliminate any genuine leaves that have no successors.
- If $a{:}\alpha$ is a Demonic choice $a{:}(g{:}\gamma \cap d{:}\delta)$, then

$$a{:}\alpha \mathbin{\underset{\downarrow}{\forall}} S = g{:}\gamma \mathbin{\underset{\downarrow}{\forall}} S \cap d{:}\delta \mathbin{\underset{\downarrow}{\forall}} S.$$

Define $R_{\alpha,S,\sigma}$ by relating $\mathfrak{l}^d\,\hat{}\,u$ to $\mathfrak{l}^d\,\hat{}\,p$ whenever $u\, R_{\gamma,S,\sigma}\, p$, and relating $\mathfrak{r}^d\,\hat{}\,u$ to $\mathfrak{r}^d\,\hat{}\,p$ whenever $u\, R_{\delta,S,\sigma}\, p$. Demon branch labels do not change state, so the required correspondence follows from the induction hypotheses applied to the branch subgames.

- If $a{:}\alpha$ is sequential composition $a{:}(g{:}\gamma;\, d{:}\delta)$, let S_d abbreviate $S(\mathsf{end} \mapsto S(d))$. Then

  $$a{:}\alpha \curlyveedownarrow S = (g{:}\gamma \curlyveedownarrow S_d)\ ;\ (d{:}\delta \curlyveedownarrow S).$$

  Policy-compliant leaves are $p_\gamma \in P_{\gamma,S_d}(\sigma)$ when $\lfloor p_\gamma \rfloor_\sigma$ is undefined, and $p_\gamma \hat{}\, p_\delta$ when $\lfloor p_\gamma \rfloor_\sigma = \tau$ and $p_\delta \in P_{\delta,S}(\tau)$.

  Define $R_{\alpha,S,\sigma}$ as follows. First, for p_γ where $\lfloor p_\gamma \rfloor_\sigma$ is undefined, let the leaf u_γ be the leaf such that $u_\gamma\, R_{\gamma,S_d,\sigma}\, p_\gamma$. Then relate u_γ to p_γ in $R_{\alpha,S,\sigma}$. Secondly, for $p_\gamma \hat{}\, p_\delta$, let u_γ be the leaf such that $u_\gamma\, R_{\gamma,S_d,\sigma}\, p_\gamma$, and u_δ be the leaf such that $u_\delta\, R_{\delta,S,\lfloor u_\gamma \rfloor_\sigma}\, p_\delta$. Then relate $u_\gamma \hat{}\, u_\delta$ to $p_\gamma \hat{}\, p_\delta$ in $R_{\alpha,S,\sigma}$.

  Universal projection leaves u_γ that are inserted-guard deadlocks in the first component and universal projection leaves $u_\gamma \hat{}\, u_\delta$ where u_δ is an inserted-guard deadlock in the second component are left unmatched. The required correspondence follows from induction hypotheses for subgames γ and δ, and the definition of sequential execution.

□

Theorem 6.1 (Subvalue Map stays in Winning Region)**.** *Suppose S is an Angelic subvalue map for game $a{:}\alpha$ compatible with winning condition ϕ, i.e., for every subgame $b \in \mathsf{subgames}(a{:}\alpha)$, $\models S(b) \to \langle a{:}\alpha_{b:} \rangle\, S(\mathsf{end})$ and $\models S(\mathsf{end}) \to \phi$. Upon starting in any state $\sigma \in [\![S(a)]\!]$, and reaching subgame b by following any strategy induced by the subvalue map S, there exists a winning strategy for Angel to win the remainder of the game. That is, $S(a) \models \langle (a{:}\alpha \curlyveedownarrow S)_{:b} \rangle\, \langle a{:}\alpha_{b:} \rangle\, \phi$.*

Proof. We perform induction on the structure of $a{:}\alpha$.

- When $a{:}\alpha$ is atomic and not controlled by Angel, i.e., $\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x)\ \&\ Q\}^d\}$, then The only subgame b in $\mathsf{subgames}(a{:}\alpha)$ is $a{:}\alpha$ itself. Thus the only possibility is $b = a$. $a{:}\alpha_{:a}$ is the empty game skip, and $a{:}\alpha_{a:}$ is the game $a{:}\alpha$, and $a{:}\alpha \curlyveedownarrow S$ is just $a{:}\alpha$. Thus, $\langle a{:}\alpha_{:a} \curlyveedownarrow S \rangle\, \langle a{:}\alpha_{a:} \rangle\, \phi$ is equivalent to $\langle \mathsf{skip} \rangle\, \langle \alpha \rangle\, \phi$. We need to show that $S(a) \models \langle a{:}\alpha \rangle\, \phi$ which holds because by definition of an Angelic subvalue map, $\models S(a) \to \langle \alpha \rangle\, \phi$.
- When $a{:}\alpha$ is an Angelic nondeterministic assignment, i.e., $a{:}x := *$, then again the only subgame b in $\mathsf{subgames}(a{:}\alpha)$ is $a{:}\alpha$ itself. Thus the only possibility is $b = a$. $a{:}\alpha_{:a}$ is the empty game skip, and $a{:}\alpha \curlyveedownarrow S$ is just $a{:}\alpha\ ;\ ?S(\mathsf{end})$, and $a{:}\alpha\ ;\ ?S(\mathsf{end})_{a:}$ is the game $a{:}\alpha\ ;\ ?S(\mathsf{end})$. Thus, $\langle a{:}\alpha_{:a} \curlyveedownarrow S \rangle\, \langle a{:}\alpha_{a:} \rangle\, \phi$ is equivalent to $\langle \mathsf{skip} \rangle\, \langle \alpha\ ;\ ?S(\mathsf{end}) \rangle\, \phi$. We need to show that $S(a) \models \langle \alpha \rangle\, (?S(\mathsf{end}) \land \phi)$. Since ϕ is a compatible winning condition, this is $S(a) \models \langle \alpha \rangle\, ?S(\mathsf{end})$ which holds by the definition of an inductive Angelic subvalue map.
- When $a{:}\alpha$ is an Angelic ODE, i.e., $a{:}\{x' = f(x)\ \&\ Q\}$, then the argument is similar to the previous case. As before the only possibility is $b = a$, and we need to show that $S(a) \models \langle \alpha \rangle\, ?S(\mathsf{end}) \land \phi$. Since ϕ is a compatible winning condition, this is $S(a) \models \langle a{:}\alpha \rangle\, ?S(\mathsf{end})$ which holds by the definition of an inductive Angelic subvalue map.
- When $a{:}\alpha$ is an Angelic choice, $a{:}(g{:}\gamma \cup d{:}\delta)$, then we analyze cases depending on the value of b.

1. If $b = a$, then $a{:}\alpha_{:a} \curlyveedownarrow S_{:b} = \mathsf{skip}$, and $a{:}\alpha_{b:} = a{:}\alpha$. Thus, $\langle a{:}\alpha_{:b} \curlyveedownarrow S\rangle \langle a{:}\alpha_{b:}\rangle \phi$ is equivalent to $\langle a{:}\alpha\rangle \phi$. $S(a) \models \langle a{:}\alpha\rangle \phi$ This can be proved by applying the axioms $\to$R along with $S(a) \to \langle a{:}\alpha\rangle \phi$ from the definition of an Angelic subvalue map.
2. Otherwise, if $b \in \mathsf{subgames}(g{:}\gamma)$, then $a{:}\alpha_{:a} \curlyveedownarrow S_{:b} = {!S(g)}\,;\, g{:}\gamma \curlyveedownarrow S_{:b}$ and $a{:}\alpha_{b:} = g{:}\gamma_{b:}$. We can show that $S(a) \models \langle {!S(g)}\,;\, g{:}\gamma \curlyveedownarrow S_{:b}\rangle \langle g{:}\gamma_{b:}\rangle \phi$ by first applying axioms $\langle;\rangle$, $[?]$, $[\cdot]$, $\langle^{d}\rangle$, $\to$R to get the goal $S(a), S(g) \models \langle g{:}\gamma \curlyveedownarrow S_{:b}\rangle \langle g{:}\gamma_{b:}\rangle \phi$ Notice that S is also an inductive Angelic subvalue map for $g{:}\gamma$ and winning condition ϕ. Thus by the inductive hypothesis, $S(g) \models \langle g{:}\gamma_{:b} \curlyveedownarrow S\rangle \langle g{:}\gamma_{b:}\rangle \phi$ completing the proof.
3. Otherwise, if $b \in \mathsf{subgames}(d{:}\delta)$, then the proof proceeds similarly to the previous case, but with $d{:}\delta$ replacing $g{:}\gamma$.

- When $a{:}\alpha$ is an Angelic loop, i.e., $a{:}(g{:}\gamma)^{*}$, then we analyze cases depending on the value of b.
    1. If $b = a$, then $a{:}\alpha_{b:} = a{:}\alpha$ and $a{:}\alpha \curlyveedownarrow S_{:b} = ({!S(g)}\,;\, g{:}\gamma \curlyveedownarrow S(\mathsf{end} \mapsto S(a)))^{\times}\,;\, {!S(\mathsf{end})}$. Thus,

       $$\langle a{:}\alpha \curlyveedownarrow S_{:b}\rangle \langle a{:}\alpha_{b:}\rangle \phi$$

       is by definition

       $$\langle ({!S(g)}\,;\, g{:}\gamma \curlyveedownarrow S(\mathsf{end} \mapsto S(a)))^{\times}\,;\, {!S(\mathsf{end})}\rangle \langle a{:}\alpha\rangle \phi.$$

       By the inductive hypothesis, $S(g) \models \langle g{:}\gamma \curlyveedownarrow S(\mathsf{end} \mapsto S(a))\rangle S(a)$. Then applying loop with invariant $S(a)$ we have

       $$S(a) \models \langle ({!S(g)}\,;\, g{:}\gamma \curlyveedownarrow S(\mathsf{end} \mapsto S(a)))^{\times}\rangle S(a).$$

       By the definition of an Angelic subvalue map, $\models S(a) \to \langle a{:}\alpha\rangle \phi$. Thus by M, $\to$R, and $\langle ?\rangle$, we prove the desired goal

       $$S(a) \models \langle ({!S(g)}\,;\, g{:}\gamma \curlyveedownarrow S(\mathsf{end} \mapsto S(a)))^{\times}\,;\, {!S(\mathsf{end})}\rangle \langle a{:}\alpha\rangle \phi.$$

    2. Otherwise, if $b \in \mathsf{subgames}(g{:}\gamma)$, then
       $a{:}\alpha \curlyveedownarrow S_{:b} = ({!S(g)}\,;\, g{:}\gamma \curlyveedownarrow S(\mathsf{end} \mapsto S(a)))^{\times}\,;\, {!S(g)}\,;\, (g{:}\gamma \curlyveedownarrow S(\mathsf{end} \mapsto S(a)))_{:b}$,
       $a{:}\alpha_{b:} = g{:}\gamma_{b:}\,;\, a{:}\alpha$.
       Want to show:

       $$\begin{aligned} S(a) \models \langle ({!S(g)}\,;\, g{:}\gamma \curlyveedownarrow S(\mathsf{end} \mapsto S(a)))^{\times}\,;\, {!S(g)}\,;\, (g{:}\gamma \curlyveedownarrow S(\mathsf{end} \mapsto S(a)))_{:b}\rangle& \\ \langle (g{:}\gamma)_{b:}\,;\, a{:}\alpha\rangle \phi.& \end{aligned}$$

       First, by Lemma B.5, we have

       $$S(a) \models \langle ({!S(g)}\,;\, g{:}\gamma \curlyveedownarrow S(\mathsf{end} \mapsto S(a)))^{\times}\,;\, {!S(\mathsf{end})}\rangle \top,$$

       so by axioms $[?]$ and $\langle;\rangle$,

       $$S(a) \models \langle ({!S(g)}\,;\, g{:}\gamma \curlyveedownarrow S(\mathsf{end} \mapsto S(a)))^{\times}\rangle \top.$$

Next, by the inductive hypothesis and applications of axioms $\langle ; \rangle$, $\langle ? \rangle$ and $\rightarrow$R, we have

$$\top \models \langle !S(g) \,;\, (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))_{:b} \rangle \, \langle g{:}\gamma_{b:} \rangle \, S(a)$$

Finally, since S is an Angelic subvalue map, we have $S(a) \models \langle a{:}\alpha \rangle \, \phi$. Thus by M and $\langle ; \rangle$ we can conclude the desired result.

- When $a{:}\alpha$ is a Demon loop, i.e., $a{:}(g{:}\gamma)^{\times}$, then we analyze cases depending on the value of b.

  1. If $b = a$, then $(a{:}\alpha \underset{\downarrow}{\forall} S)_{:b} = a{:}\alpha \underset{\downarrow}{\forall} S$ and $(a{:}\alpha)_{b:} = a{:}\alpha$.
     Thus we must show that $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\forall} S \rangle \, \langle a{:}\alpha \rangle \, \phi$.
     $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\forall} S \rangle \, S(a)$ because $S(a)$ is a loop invariant of $(g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))^{\times}$ by Lemma B.5.
     $S(a) \models \langle a{:}\alpha \rangle \, \phi$ by definition of a subvalue map. Thus by M, we can conclude the desired result.
  2. Otherwise, if $b \in \mathsf{subgames}(g{:}\gamma)$, then:
     $(a{:}\alpha \underset{\downarrow}{\forall} S)_{:b} = a{:}\alpha \underset{\downarrow}{\forall} S \,;\, (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))_{:b}$,
     $(a{:}\alpha)_{b:} = (g{:}\gamma)_{b:} \,;\, a{:}\alpha$.
     Thus we must show that

     $$S(a) \models \langle a{:}\alpha \underset{\downarrow}{\forall} S \,;\, (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))_{:b} \rangle \langle (g{:}\gamma)_{b:} \,;\, a{:}\alpha \rangle \, \phi$$

     First, we have $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\forall} S \rangle \, S(a)$ because $S(a)$ is a loop invariant of
     $(g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))^{\times}$ by Lemma B.5.
     Next, by the inductive hypothesis, we have

     $$S(a) \models \langle (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))_{:b} \,;\, (g{:}\gamma)_{b:} \rangle \, S(a).$$

     By definition of a subvalue map, we have $S(a) \models \langle a{:}\alpha \rangle \, \phi$.
     Thus by M and $\langle ; \rangle$, we can conclude the desired result.

- When $a{:}\alpha$ is sequential composition, i.e., $a{:}(g{:}\gamma \,;\, d{:}\delta)$, then we analyze cases depending on the value of b.

  1. If $b = a$, then $(a{:}\alpha \underset{\downarrow}{\forall} S)_{:b} = \mathsf{skip}$ and $a{:}\alpha_{b:} = a{:}\alpha$. Thus we must show that $S(a) \models \langle \mathsf{skip} \rangle \, \langle a{:}\alpha \rangle \, \phi$, i.e., $S(a) \models \langle a{:}\alpha \rangle \, \phi$, which hold by definition of a subvalue map.
  2. Otherwise, if $b \in \mathsf{subgames}(g{:}\gamma)$, then $(a{:}\alpha \underset{\downarrow}{\forall} S)_{:b} = (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(d)))_{:b}$ and $a{:}\alpha_{b:} = (g{:}\gamma)_{b:} \,;\, d{:}\delta$.
     Thus we must show that $S(a) \models \langle g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(d))_{:b} \rangle \, \langle (g{:}\gamma)_{b:} \,;\, d{:}\delta \rangle \, \phi$.
     This is $S(a) \models \langle (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(d)))_{:b} \rangle \, \langle (g{:}\gamma)_{b:} \rangle \, \langle d{:}\delta \rangle \, \phi$ applying $\langle ; \rangle$.
     This is $S(a) \models \langle (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(d)))_{:b} \,;\, (g{:}\gamma)_{b:} \rangle \, \langle d{:}\delta \rangle \, \phi$ again applying $\langle ; \rangle$.
     By the inductive hypothesis, $S(a) \models \langle (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(d)))_{:b} \rangle \, S(d)$ since $S(a) \rightarrow S(g)$ in an inductive subvalue map.
     By definition of an Angelic subvalue map, $S(d) \models \langle d{:}\delta \rangle \, \phi$.
     By rule M, we can conclude the desired result.
  3. Otherwise, if $b \in \mathsf{subgames}(d{:}\delta)$, then a similar argument applies.
     $a{:}\alpha \underset{\downarrow}{\forall} S_{:b} = \{g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(d)) \,;\, (d{:}\delta \underset{\downarrow}{\forall} S)_{:b}\}$, and $a{:}\alpha_{b:} = d{:}\delta_{b:}$.
     Thus we must show that $S(a) \models \langle g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(d)) \,;\, (d{:}\delta \underset{\downarrow}{\forall} S)_{:b} \rangle \, \langle d{:}\delta_{b:} \rangle \, \phi$.

By the $\langle ; \rangle$ axiom, this is $S(a) \models \langle g{:}\gamma \overset{\forall}{\downarrow} S(\mathsf{end} \mapsto S(d)) \rangle \langle (d{:}\delta \overset{\forall}{\downarrow} S)_{:b} \,;\, (d{:}\delta)_{b:} \rangle \,\phi$.
By Lemma B.5, and since in an inductive subvalue map, $\models S(a) \to S(g)$, $S(a) \models \langle g{:}\gamma \overset{\forall}{\downarrow} S(\mathsf{end} \mapsto S(d)) \rangle\, S(d)$.
By the inductive hypothesis, $S(d) \models \langle (d{:}\delta \overset{\forall}{\downarrow} S)_{:b} \,;\, (d{:}\delta)_{b:} \rangle \,\phi$.
By rule M, we can conclude the desired result.

□

Theorem 6.2 (Inductive subvalue map ensures winning actions)**.** *Suppose S is an inductive Angelic subvalue map for game $a{:}\alpha$ and compatible with winning condition ϕ, i.e., $a{:}\alpha \models S$ and $S(\mathsf{end}) \models \phi$. Upon starting in any state $\sigma \models S(a)$, and reaching subgame b by following any strategy induced by the subvalue map S, there is a way for Angel to win by continuing to follow S. That is, $S(a) \models \langle (a{:}\alpha \overset{\forall}{\downarrow} S)_{:b} \rangle \langle (a{:}\alpha \overset{\exists}{\downarrow} S)_{b:} \rangle \,\phi$.*

Proof. We perform induction on the structure of $a{:}\alpha$.

- When $a{:}\alpha$ is atomic and not controlled by Angel, i.e., $\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x) \;\&\; Q\}^d\}$, then The only subgame b in $\mathsf{subgames}(a{:}\alpha)$ is $a{:}\alpha$ itself. Thus the only possibility is $b = a$. $a{:}\alpha_{:a}$ is the empty game skip, and $a{:}\alpha_{a:}$ is the game $a{:}\alpha$, and $a{:}\alpha \overset{\forall}{\downarrow} S$ is just $a{:}\alpha$. Thus, $\langle (a{:}\alpha \overset{\forall}{\downarrow} S)_{:a} \rangle \langle (a{:}\alpha \overset{\exists}{\downarrow} S)_{a:} \rangle \,\phi$ is equivalent to $\langle \mathsf{skip} \rangle \langle \alpha \rangle \,\phi$. We need to show that $S(a) \models \langle a{:}\alpha \overset{\exists}{\downarrow} S \rangle \,\phi$ which holds because by definition of an inductive Angelic subvalue map along with the fact that $S(\mathsf{end}) \to \phi$, $S(a) \to \langle a{:}\alpha \rangle \,\phi$.
- When $a{:}\alpha$ is an Angelic nondeterministic assignment, i.e., $a{:}x := *$, then again the only subgame b in $\mathsf{subgames}(a{:}\alpha)$ is $a{:}\alpha$ itself. Thus the only possibility is $b = a$. $a{:}\alpha_{:a}$ is the empty game skip, and $a{:}\alpha \overset{\forall}{\downarrow} S$ is just $a{:}\alpha \,;\, ?S(\mathsf{end})$, and $a{:}\alpha \,;\, ?S(\mathsf{end})_{a:}$ is the game $a{:}\alpha \,;\, ?S(\mathsf{end})$. Thus, $\langle a{:}\alpha_{:a} \overset{\forall}{\downarrow} S \rangle \langle a{:}\alpha_{a:} \overset{\exists}{\downarrow} S \rangle \,\phi$ is equivalent to $\langle \mathsf{skip} \rangle \langle \alpha \,;\, ?S(\mathsf{end}) \rangle \,\phi$. We need to show that $S(a) \models \langle \alpha \rangle \,(?S(\mathsf{end}) \land \phi)$. Since ϕ is a compatible winning condition, this is $S(a) \models \langle \alpha \rangle \,?S(\mathsf{end})$ which holds by the definition of an inductive Angelic subvalue map.
- When $a{:}\alpha$ is an Angelic ODE, i.e., $a{:}\{x' = f(x) \;\&\; Q\}$, then the argument is similar to the previous case. As before the only possibility is $b = a$, and we need to show that $S(a) \models \langle \alpha \rangle \,?S(\mathsf{end}) \land \phi$. Since ϕ is a compatible winning condition, this is $S(a) \models \langle a{:}\alpha \overset{\exists}{\downarrow} S \rangle \,?S(\mathsf{end})$ which holds by the definition of an inductive Angelic subvalue map.
- When $a{:}\alpha$ is an Angelic choice, i.e., $a{:}(g{:}\gamma \cup d{:}\delta)$, then we analyze cases depending on the value of b.
  1. If $b = a$, then $(a{:}\alpha \overset{\forall}{\downarrow} S)_{:b} = \mathsf{skip}$ and $a{:}\alpha \overset{\exists}{\downarrow} S_{b:} = a{:}\alpha \overset{\exists}{\downarrow} S$. Thus we must show that $S(a) \models \langle \mathsf{skip} \rangle \langle a{:}\alpha \overset{\exists}{\downarrow} S \rangle \,\phi$, i.e., $S(a) \models \langle a{:}\alpha \overset{\exists}{\downarrow} S \rangle \,\phi$, which holds by Lemma B.6.
  2. Otherwise, if $b \in \mathsf{subgames}(g{:}\gamma)$, then $(a{:}\alpha \overset{\forall}{\downarrow} S)_{:b} = {!S(g)} \,;\, g{:}\gamma \overset{\forall}{\downarrow} S_{:b}$ and $a{:}\alpha \overset{\exists}{\downarrow} S_{b:} = g{:}\gamma_{b:}$. Thus we must show that $S(a) \models \langle !S(g) \,;\, g{:}\gamma \overset{\forall}{\downarrow} S_{:b} \rangle \langle g{:}\gamma_{b:} \rangle \,\phi$.
     This is $S(a) \land S(g) \models \langle g{:}\gamma \overset{\forall}{\downarrow} S_{:b} \rangle \langle g{:}\gamma_{b:} \rangle \,\phi$ using axioms $\langle ; \rangle$, $[?]$. This holds by the inductive hypothesis.
  3. Otherwise, if $b \in \mathsf{subgames}(d{:}\delta)$, then the proof proceeds similarly to the previous case, but with $d{:}\delta$ replacing $g{:}\gamma$.
- When $a{:}\alpha$ is a Demonic choice, i.e., $a{:}(g{:}\gamma \cap d{:}\delta)$, then the argument is similar to Angelic choice, except that we get assumptions $S(g)$ or $S(d)$ not from the execution prefix but from

$S(a)$ because in an inductive subvalue map, $\models S(a) \rightarrow S(g) \wedge S(d)$. We analyze cases depending on the value of b.

1. If $b = a$, then $(a{:}\alpha \forall\!\!\!\downarrow S)_{:b} = \mathsf{skip}$ and $a{:}\alpha \exists\!\!\!\downarrow S_{b:} = a{:}\alpha \exists\!\!\!\downarrow S$. Thus we must show that $S(a) \models \langle \mathsf{skip} \rangle \langle a{:}\alpha \exists\!\!\!\downarrow S \rangle\, \phi$, i.e., $S(a) \models \langle a{:}\alpha \exists\!\!\!\downarrow S \rangle\, \phi$, which holds by Lemma B.6.
2. Otherwise, if $b \in \mathsf{subgames}(g{:}\gamma)$, then $(a{:}\alpha \forall\!\!\!\downarrow S)_{:b} = g{:}\gamma \forall\!\!\!\downarrow S_{:b}$ and $a{:}\alpha \exists\!\!\!\downarrow S_{b:} = g{:}\gamma_{b:}$. Thus we must show that $S(a) \models \langle g{:}\gamma \forall\!\!\!\downarrow S_{:b} \rangle \langle g{:}\gamma_{b:} \rangle\, \phi$. This holds by the inductive hypothesis since as discussed, $S(a) \rightarrow S(g)$.
3. Otherwise, if $b \in \mathsf{subgames}(d{:}\delta)$, then the proof proceeds similarly to the previous case, but with $d{:}\delta$ replacing $g{:}\gamma$.

- When $a{:}\alpha$ is sequential composition, i.e., $a{:}(g{:}\gamma\,;\, d{:}\delta)$, then we analyze cases depending on the value of b.
  1. If $b = a$, then $(a{:}\alpha \forall\!\!\!\downarrow S)_{:b} = \mathsf{skip}$ and $a{:}\alpha \exists\!\!\!\downarrow S_{b:} = a{:}\alpha \exists\!\!\!\downarrow S$. Thus we must show that $S(a) \models \langle \mathsf{skip} \rangle \langle a{:}\alpha \exists\!\!\!\downarrow S \rangle\, \phi$, i.e., $S(a) \models \langle a{:}\alpha \exists\!\!\!\downarrow S \rangle\, \phi$, which holds by Lemma B.6.
  2. Otherwise, if $b \in \mathsf{subgames}(g{:}\gamma)$, then $(a{:}\alpha \forall\!\!\!\downarrow S)_{:b} = (g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(d)))_{:b}$
     and $a{:}\alpha \exists\!\!\!\downarrow S_{b:} = (g{:}\gamma \exists\!\!\!\downarrow S(\mathsf{end} \mapsto S(d)))_{b:}\,;\, d{:}\delta \exists\!\!\!\downarrow S$.
     Thus we must show that:
     $S(a) \models \langle g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(d))_{:b} \rangle \langle (g{:}\gamma \exists\!\!\!\downarrow S(\mathsf{end} \mapsto S(d)))_{b:}\,;\, d{:}\delta \exists\!\!\!\downarrow S \rangle\, \phi$.
     This is $S(a) \models \langle (g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(d)))_{:b} \rangle \langle (g{:}\gamma \exists\!\!\!\downarrow S(\mathsf{end} \mapsto S(d)))_{b:} \rangle \langle d{:}\delta \exists\!\!\!\downarrow S \rangle\, \phi$ applying $\langle ; \rangle$.
     This is $S(a) \models \langle (g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(d)))_{:b}\,;\, (g{:}\gamma \exists\!\!\!\downarrow S(\mathsf{end} \mapsto S(d)))_{b:} \rangle \langle d{:}\delta \exists\!\!\!\downarrow S \rangle\, \phi$ again applying $\langle ; \rangle$.
     By the inductive hypothesis, $S(a) \models \langle (g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(d)))_{:b} \rangle\, S(d)$ since $S(a) \rightarrow S(g)$ in an inductive subvalue map.
     By Lemma B.6, $S(d) \models \langle d{:}\delta \exists\!\!\!\downarrow S \rangle\, \phi$.
     By rule M, we can conclude the desired result.
  3. Otherwise, if $b \in \mathsf{subgames}(d{:}\delta)$, then a similar argument applies.
     $a{:}\alpha \forall\!\!\!\downarrow S_{:b} = \{g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(d))\,;\, (d{:}\delta \forall\!\!\!\downarrow S)_{:b}\}$, and $a{:}\alpha \exists\!\!\!\downarrow S_{b:} = d{:}\delta \exists\!\!\!\downarrow S_{b:}$.
     Thus we must show that $S(a) \models \langle g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(d))\,;\, (d{:}\delta \forall\!\!\!\downarrow S)_{:b} \rangle \langle d{:}\delta \exists\!\!\!\downarrow S_{b:} \rangle\, \phi$.
     By the $\langle ; \rangle$ axiom, this is $S(a) \models \langle g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(d)) \rangle \langle (d{:}\delta \forall\!\!\!\downarrow S)_{:b}\,;\, (d{:}\delta \exists\!\!\!\downarrow S)_{b:} \rangle\, \phi$.
     By Lemma B.7, and since in an inductive subvalue map, $\models S(a) \rightarrow S(g)$, $S(a) \models \langle g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(d)) \rangle\, S(d)$.
     By the inductive hypothesis, $S(d) \models \langle (d{:}\delta \forall\!\!\!\downarrow S)_{:b}\,;\, (d{:}\delta \exists\!\!\!\downarrow S)_{b:} \rangle\, \phi$.
     By rule M, we can conclude the desired result.
- When $a{:}\alpha$ is an Angelic loop, i.e., $a{:}(g{:}\gamma)^*$, then we analyze cases depending on the value of b.
  1. If $b = a$ then $(a{:}\alpha \forall\!\!\!\downarrow S)_{:b} = (!S(g)\,;\, g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(a)))^{\times}$ (with the terminal test missing compared to $a{:}\alpha \forall\!\!\!\downarrow S$) and $a{:}\alpha \exists\!\!\!\downarrow S_{b:} = a{:}\alpha \exists\!\!\!\downarrow S$.
     We need to show that $S(a) \models \langle (!S(g)\,;\, g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(a)))^{\times} \rangle \langle a{:}\alpha \exists\!\!\!\downarrow S \rangle\, \phi$.
     We first show that $S(a) \models \langle (!S(g)\,;\, g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(a)))^{\times} \rangle\, S(a)$. This follows from applying the loop rule with invariant $S(a)$ and then using the fact that per Lemma B.7, $S(g) \models \langle g{:}\gamma \forall\!\!\!\downarrow S(\mathsf{end} \mapsto S(a)) \rangle\, S(a)$.
     Then by Lemma B.6, $S(a) \models \langle a{:}\alpha \exists\!\!\!\downarrow S \rangle\, \phi$. Thus by M, we can conclude the desired result.

2. Otherwise, if $b \neq a$ then it is in $\mathsf{subgames}(g{:}\gamma)$.
   $(a{:}\alpha \underset{\downarrow}{\forall} S)_{:b} = (!S(g)\,;\, g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a))_{:b})^{\times}\,;\, !S(g)\,;\, (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))_{:b}$.
   $a{:}\alpha \underset{\downarrow}{\exists} S_{b:} = (g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(a))_{b:}\,;\, a{:}\alpha \underset{\downarrow}{\exists} S$.
   Want to show:

$$S(a) \models \langle (!S(g)\,;\, g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))^{\times}\,;\, !S(g)\,;\, (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))_{:b}\rangle$$
$$\langle (g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(a)))_{b:}\,;\, a{:}\alpha \underset{\downarrow}{\exists} S\rangle\, \phi.$$

   First, by Lemma B.7, we have

$$S(a) \models \langle (!S(g)\,;\, g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))^{\times}\,;\, !S(\mathsf{end})\rangle\, \top,$$

   so by axioms [?] and $\langle ; \rangle$,

$$S(a) \models \langle (!S(g)\,;\, g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))^{\times}\rangle\, \top.$$

   Next, by the inductive hypothesis and applications of axioms $\langle ; \rangle$, $\langle ? \rangle$ and $\rightarrow$R, we have

$$\top \models \langle !S(g)\,;\, (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))_{:b}\rangle\, \langle g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(a))_{b:}\rangle\, S(a)$$

   Finally, by Lemma B.6, we have $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\, \phi$. Thus by M and $\langle ; \rangle$ we can conclude the desired result.

- When $a{:}\alpha$ is a Demonic loop, i.e., $a{:}(g{:}\gamma)^{\times}$, then we analyze cases depending on the value of b.
  1. If $b = a$, then $(a{:}\alpha \underset{\downarrow}{\forall} S)_{:b} = a{:}\alpha \underset{\downarrow}{\forall} S$ and $(a{:}\alpha \underset{\downarrow}{\exists} S)_{b:} = a{:}\alpha \underset{\downarrow}{\exists} S$.
     Thus we must show that $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\forall} S\rangle\, \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\, \phi$.
     $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\forall} S\rangle\, S(a)$ because $S(a)$ is a loop invariant of $(g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))^{\times}$ by Lemma B.7.
     $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\, \phi$ by Lemma B.6. Thus by M, we can conclude the desired result.
  2. Otherwise, if $b \in \mathsf{subgames}(g{:}\gamma)$, then:
     $(a{:}\alpha \underset{\downarrow}{\forall} S)_{:b} = a{:}\alpha \underset{\downarrow}{\forall} S\,;\, (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))_{:b}$,
     $(a{:}\alpha \underset{\downarrow}{\exists} S)_{b:} = (g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(a)))_{b:}\,;\, a{:}\alpha \underset{\downarrow}{\exists} S$.
     Thus we must show that

$$S(a) \models \langle a{:}\alpha \underset{\downarrow}{\forall} S\,;\, (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))_{:b}\rangle \langle (g{:}\gamma \underset{\downarrow}{\exists} S(\mathsf{end} \mapsto S(a)))_{b:}\,;\, a{:}\alpha \underset{\downarrow}{\exists} S\rangle\, \phi$$

     First, we have $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\forall} S\rangle\, S(a)$ because $S(a)$ is a loop invariant of
     $(g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))^{\times}$ by Lemma B.7.
     Next, by the inductive hypothesis, we have $S(a) \models \langle (g{:}\gamma \underset{\downarrow}{\forall} S(\mathsf{end} \mapsto S(a)))_{:b}\rangle\, S(a)$.
     By Lemma B.6, we have $S(a) \models \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\, \phi$.
     Thus by M and $\langle ; \rangle$, we can conclude the desired result.

$\square$

Theorem 6.3 (Inductive subvalue maps are subvalue maps)**.** *For* dGL *game* $a{:}\alpha$*, if* S *is an inductive Angelic subvalue map (*$a{:}\alpha \models S$*), then it is also an Angelic subvalue map for* $a{:}\alpha$*, i.e., for every subgame* $b{:}\beta$ *in* $\mathsf{subgames}(a{:}\alpha)$*,* $\models S(b) \rightarrow \langle a{:}\alpha_{b:}\rangle S(\mathsf{end})$*. Dually, if* S *is an inductive Demonic subvalue map for* $a{:}\alpha$*, then* S *is a Demonic subvalue map for* $a{:}\alpha$*.*

Proof. The proof uses structural induction. When $a{:}\alpha$ is atomic, i.e. $\alpha \in \{x := e,\ x := *,\ ?Q,\ !Q,\ \{x' = f(x)\ \&\ Q\},\ \{x' = f(x)\ \&\ Q\}^d\}$, according to Def. 6.8, $a{:}\alpha \models S$ exactly when $\models S(a) \rightarrow \langle\alpha\rangle S(\mathsf{end})$. Since $a{:}\alpha_{a:} = a{:}\alpha$ and a is the only subgame in $\mathsf{subgames}(a{:}\alpha)$, we can conclude that S is a subvalue map. For the recursive cases, if $a{:}\alpha$ has the structure:

1. $g{:}\gamma \cup d{:}\delta$, for any subgame b in $\mathsf{subgames}(g{:}\gamma)$, $a{:}\alpha_{b:} = g{:}\gamma_{b:}$. Since per the inductive hypothesis, $\models S(b) \rightarrow \langle g{:}\gamma_{b:}\rangle S(\mathsf{end})$, we can conclude that $\models S(b) \rightarrow \langle a{:}\alpha_{b:}\rangle S(\mathsf{end})$. The same argument applies to all subgames in $\mathsf{subgames}(d{:}\delta)$. It remains to show that for the overall game a, $\models S(a) \rightarrow \langle a{:}\alpha_{a:}\rangle S(\mathsf{end})$. By the inductive hypothesis, because $g{:}\gamma \models S$, so $\models S(g) \rightarrow \langle\gamma\rangle S(\mathsf{end})$ and because $d{:}\delta \models S$, so $\models S(d) \rightarrow \langle\delta\rangle S(\mathsf{end})$. Using the disjunction of implications, $\models S(g) \vee S(d) \rightarrow \langle\gamma\rangle S(\mathsf{end}) \vee \langle\delta\rangle S(\mathsf{end})$. But according to dGL axiom $\langle\cup\rangle$, $\models \langle\gamma\rangle S(\mathsf{end}) \vee \langle\delta\rangle S(\mathsf{end}) \leftrightarrow \langle\gamma \cup \delta\rangle S(\mathsf{end})$, so $\models S(g) \vee S(d) \rightarrow \langle\gamma \cup \delta\rangle S(\mathsf{end})$. Because S is valid, $\models S(a) \rightarrow S(g) \vee S(d)$. By transitivity of implication, $\models S(a) \rightarrow \langle\gamma \cup \delta\rangle S(\mathsf{end})$, proving the desired result.
2. $g{:}\gamma \cap d{:}\delta$, for any subgame b in $\mathsf{subgames}(g{:}\gamma)$, $a{:}\alpha_{b:} = g{:}\gamma_{b:}$. Since per the inductive hypothesis, $\models S(b) \rightarrow \langle g{:}\gamma_{b:}\rangle S(\mathsf{end})$, we can conclude that $\models S(b) \rightarrow \langle a{:}\alpha_{b:}\rangle S(\mathsf{end})$. The same argument applies to all subgames in $\mathsf{subgames}(d{:}\delta)$. It remains to show that for the overall game a, $\models S(a) \rightarrow \langle a{:}\alpha_{a:}\rangle S(\mathsf{end})$. By the inductive hypothesis, because $g{:}\gamma \models S$, so $\models S(g) \rightarrow \langle\gamma\rangle S(\mathsf{end})$ and because $d{:}\delta \models S$, so $\models S(d) \rightarrow \langle\delta\rangle S(\mathsf{end})$. Using the conjunction of implications, $\models S(g) \wedge S(d) \rightarrow \langle\gamma\rangle S(\mathsf{end}) \wedge \langle\delta\rangle S(\mathsf{end})$. But according to dGL axiom $\langle\cap\rangle$, $\models \langle\gamma\rangle S(\mathsf{end}) \wedge \langle\delta\rangle S(\mathsf{end}) \leftrightarrow \langle\gamma \cap \delta\rangle S(\mathsf{end})$, so $\models S(g) \wedge S(d) \rightarrow \langle\gamma \cap \delta\rangle S(\mathsf{end})$. Because S is valid, $\models S(a) \rightarrow S(g) \wedge S(d)$. By transitivity of implication, $\models S(a) \rightarrow \langle\gamma \cap \delta\rangle S(\mathsf{end})$, proving the desired result.
3. $g{:}\gamma; d{:}\delta$, for any subgame b in $\mathsf{subgames}(d{:}\delta)$, $d{:}\delta_{b:} = a{:}\alpha_{b:}$. Since per the inductive hypothesis, $\models S(b) \rightarrow \langle d{:}\delta_{b:}\rangle S(\mathsf{end})$, we can conclude that $\models S(b) \rightarrow \langle a{:}\alpha_{b:}\rangle S(\mathsf{end})$.

   Consider now any subgame b in $\mathsf{subgames}(g{:}\gamma)$. Because $g{:}\gamma \models S$, $\models S(b) \rightarrow \langle g{:}\gamma_{b:}\rangle S(d)$. Now, because $d{:}\delta \models S$, $\models S(d) \rightarrow \langle\delta\rangle S(\mathsf{end})$. According to the dGL monotonicity rule M, this means that $\models S(b) \rightarrow \langle g{:}\gamma_{b:}\rangle\langle\delta\rangle S(\mathsf{end})$. According to the dGL axiom $\langle;\rangle$, $\models S(b) \rightarrow \langle g{:}\gamma_{b:}; \delta\rangle S(\mathsf{end})$. Since $a{:}\alpha_{b:} = g{:}\gamma_{b:}; d{:}\delta$, we can conclude that $\models S(b) \rightarrow \langle a{:}\alpha_{b:}\rangle S(\mathsf{end})$.

   It remains to show that for the overall game a, $\models S(a) \rightarrow \langle a{:}\alpha_{a:}\rangle S(\mathsf{end})$. By the inductive hypothesis, because $g{:}\gamma \models S$, so $\models S(g) \rightarrow \langle\gamma\rangle S(\delta)$ and because $d{:}\delta \models S$, so $\models S(d) \rightarrow \langle\delta\rangle S(\mathsf{end})$. According to the dGL monotonicity rule M, this means that $\models S(g) \rightarrow \langle\gamma\rangle\langle\delta\rangle S(\mathsf{end})$. According to the dGL axiom $\langle;\rangle$, $\models \langle\gamma\rangle\langle\delta\rangle S(\mathsf{end}) \leftrightarrow \langle\gamma\,;\,\delta\rangle S(\mathsf{end})$. Thus, $\models S(g) \rightarrow \langle\gamma\,;\,\delta\rangle S(\mathsf{end})$. Because S is valid, $\models S(a) \rightarrow S(g)$. By transitivity of implication, $\models S(a) \rightarrow \langle\gamma\,;\,\delta\rangle S(\mathsf{end})$, proving the desired result.
4. $g{:}\gamma^*$, then because $a{:}(g{:}\gamma)^* \models S$, then immediately as the first validity condition $\models S(a) \rightarrow \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle S(\mathsf{end})$. Per Lemma B.2, $\models \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle S(\mathsf{end}) \rightarrow \langle\alpha\rangle S(\mathsf{end})$. By the transitivity of implication, $\models S(a) \rightarrow \langle\alpha\rangle S(\mathsf{end})$.

5. $g{:}\gamma^\times$, then because $a{:}(g{:}\gamma^\times) \models S$, $\models S(a) \to S(g)$ and $\models S(a) \to S(\mathsf{end})$. Additionally, by the inductive hypothesis, because $g{:}\gamma \models S, \models S(g) \to \langle\gamma\rangle S(a)$. By the transitivity of implication, $\models S(a) \to \langle\gamma\rangle S(a)$. Applying the ind rule of dGL, this permits us to infer $\models S(a) \to \langle\gamma^\times\rangle S(a)$. Recall that because $a{:}(g{:}\gamma^\times) \models S$, $\models S(a) \to S(\mathsf{end})$. By the monotonicity rule M, $\models S(a) \to \langle\gamma^\times\rangle S(\mathsf{end})$, proving that $\models S(a) \to a{:}\alpha_{a:}$.
   It remains to show that for any subgame b in $\mathsf{subgames}(g{:}\gamma)$, $\models S(b) \to \langle a{:}\alpha_{b:}\rangle S(\mathsf{end})$. Because $g{:}\gamma \models S$, by the inductive hypothesis $\models S(b) \to \langle g{:}\gamma_{b:}\rangle S(a)$. Since $\models S(a) \to \langle\gamma^\times\rangle S(\mathsf{end})$, by the monotonicity rule M, $\models S(b) \to \langle g{:}\gamma_{b:}\rangle\langle\gamma^\times\rangle S(\mathsf{end})$. By dGL axiom $\langle;\rangle$, $\models S(b) \to \langle g{:}\gamma_{b:};\gamma^\times\rangle S(\mathsf{end})$. But, $a{:}\alpha_{b:} = g{:}\gamma_{b:}\,;\,(\gamma^\times)$. Thus, $\models S(b) \to \langle a{:}\alpha_{b:}\rangle S(\mathsf{end})$.

□

Corollary D.1 (Inductive subvalue map correctness). *For* dGL *game* $a{:}\alpha$ *and inductive Angelic subvalue map* S*, if* $a{:}\alpha \models S$*, then* $\models S(a) \to \langle\alpha\rangle\, S(\mathsf{end})$*. Dually, if* $a{:}\alpha[] \models S$*, then* $\models S(a) \to [\alpha]S(\mathsf{end})$.

Proof. Corollary of Theorem 6.3, which shows that inductive subvalue maps are subvalue maps. Def. 6.2 says that the winning subregion for every subgame in the subvalue map implies the winning region for starting at that subgame and running the rest of the game. For overall game a, this means $\models S(a) \to \langle\alpha\rangle S(\mathsf{end})$. □

Theorem 6.4 (Sound solving). $S := \langle\rangle\mathsf{map}(a{:}\alpha, \phi)$ *is an inductive Angelic subvalue map for game* $a{:}\alpha$ *(*$a{:}\alpha \models S$*) with all subvalues in* $\mathcal{P}_\mathbb{R}$*, compatible with Angel winning condition* $\phi \in \mathcal{P}_\mathbb{R}$*. Dually,* $S := []\mathsf{map}(a{:}\alpha, \phi)$ *is an inductive Demonic subvalue map with all subvalues in* $\mathcal{P}_\mathbb{R}$*, compatible with Demon winning condition* ϕ*.*

Proof. The proof is by structural induction on $a{:}\alpha$. We show the proof for $\langle\rangle\mathsf{map}$, the proof for $[]\mathsf{map}$ is symmetric. $S(\mathsf{end})$ is compatible with ϕ because $S(\mathsf{end})$ is set to ϕ. In the case where $a{:}\alpha$ is atomic, i.e., when $\alpha \in \{x := e, x := *, ?f, !f, \{x' = f(x)\&Q\}, \{x' = f(x)\&Q\}^d\}$, the result is immediate. According to Def. 6.8, $a{:}\alpha \models S$ exactly when $\models S(a) \to \langle\alpha\rangle\phi$. But per the definition of $\langle\rangle\mathsf{map}$, $S(a)$ is $\mathsf{simpl}(\langle\alpha\rangle\phi)$ which by definition implies $\langle\alpha\rangle\phi$.

In the compositional cases, we use structural induction. If $a{:}\alpha$ has the structure:

- $a{:}(g{:}\gamma \cup d{:}\delta)$, then let $S_1 := \langle\rangle\mathsf{map}(g{:}\gamma, \phi)$ and $S_2 := \langle\rangle\mathsf{map}(d{:}\delta, \phi)$. S is the disjoint union of S_1 and S_2 extended by mapping a to $S_1(g) \vee S_2(d)$. We must show that $\models S(a) \to S(g) \vee S(d)$ and $g{:}\gamma \models S$ and $d{:}\delta \models S$. The first implication is immediate as $S(g) = S_1(g)$ and $S(d) = S_2(d)$ by the construction of S. The second and third formulas hold by the inductive hypothesis.
- $a{:}(g{:}\gamma \cap d{:}\delta)$, then let $S_1 := \langle\rangle\mathsf{map}(g{:}\gamma, \phi)$ and $S_2 := \langle\rangle\mathsf{map}(d{:}\delta, \phi)$. S is the disjoint union of S_1 and S_2 extended by mapping a to $S_1(g) \wedge S_2(d)$. We must show that $\models S(a) \to S(g) \wedge S(d)$ and $g{:}\gamma \models S$ and $d{:}\delta \models S$. The first implication is immediate as $S(g) = S_1(g)$ and $S(d) = S_2(d)$ by the construction of S. The second and third formulas hold by the inductive hypothesis.

- $a{:}(g{:}\gamma\,;\,d{:}\delta)$, then let $S_1 := \langle\rangle\mathsf{map}(d{:}\delta, \phi)$ and $S_2 := \langle\rangle\mathsf{map}(g{:}\gamma, S_1(d))$. S is the disjoint union of S_1 and S_2 extended by mapping a to $S_2(g)$. We must show that $\models S(a) \rightarrow S(g)$ and $g{:}\gamma \models S$ and $d{:}\delta \models S$. The first implication is immediate as $S(g) = S_2(g)$ by the construction of S. The second and third formulas hold by the inductive hypothesis.
- $a{:}(g{:}\gamma)^*$, then let $S' := \langle\rangle\mathsf{map}(g{:}\gamma, \mathsf{Inv} \vee \phi)$. S is constructed by extending S' by mapping a to Inv. Additionally, per the side condition $\models \mathsf{Inv} \rightarrow \langle a{:}\alpha \underset{\downarrow}{\exists} S'\rangle\phi$. We must show $S(a) \rightarrow \langle a{:}\alpha \underset{\downarrow}{\exists} S\rangle\phi$ and $g{:}\gamma \models S$. The first implication follows from the side condition along with the fact that $S(g) = S'(g)$ by the construction of S. The second formula holds by the inductive hypothesis as it applies to S'.
- $a{:}(g{:}\gamma)^\times$, then let $S' := \langle\rangle\mathsf{map}(g{:}\gamma, \mathsf{Inv})$. S is constructed by extending S' by mapping a to Inv. Additionally, per the side condition $\models \mathsf{Inv} \rightarrow S(g) \wedge \phi$. We must show that $S(a) \rightarrow \phi \wedge S(g)$, and $g{:}\gamma \models S$. The first implication follows from the side condition along with the fact that $S(g) = S'(g)$ by the construction of S. The second formula holds by the inductive hypothesis as it applies to S'.

□

Theorem 6.5 (Maximal inductive subvalue map)**.** *Amongst the inductive Angelic subvalue maps for game $a{:}\alpha$ compatible with winning condition ϕ, the model predictive Angelic map given by Def. 6.9 is maximal under the ordering of Def. 6.10. That is, for all S such that $a{:}\alpha \models S$, model predictive Angelic subvalue map S' satisfies $S' \sqsupseteq S$. Dually, the model predictive Demonic subvalue map (Def. 6.9) is maximal amongst inductive Demonic subvalue maps per the ordering of Def. 6.10.*

Proof. Lemma B.4 shows that the MPC solution is valid. Next we show that it is maximal. We first show that $S' \succsim S$ for all S such that $a{:}\alpha \models S$ under the weaker ordering of Def. B.1. A symmetric proof applies when $a{:}\alpha \models S$ is replaced by $a{:}\alpha[] \models S$. The proof uses structural induction.

For end, if $a{:}\alpha \models S$ then $\models S(\mathsf{end}) \rightarrow \phi$. But for the MPC solution, $S'(\mathsf{end})$ is ϕ, so $\models S(a) \rightarrow S'(a)$.

When $a{:}\alpha$ is atomic, i.e. $\alpha \in \{x := e, x := *, ?f, !f, \{x' = f(x)\ \&\ Q\}, \{x' = f(x)\ \&\ Q\}^d\}$, the result is immediate: according to Def. 6.8, if $a{:}\alpha \models S$ then $\models S(a) \rightarrow \langle\alpha\rangle\phi$. But per the definition of the MPC solution, $S'(a)$ is $\langle\alpha\rangle\phi$. Thus, $\models S(a) \rightarrow S'(a)$.

For the recursive cases, if $a{:}\alpha$ has the structure:

1. $g{:}\gamma \cup d{:}\delta$, then observe that restrictions $S'|_{\mathsf{subgames}(\gamma)}$ and $S'|_{\mathsf{subgames}(\delta)}$ of S' are the MPC solutions for Angel win condition ϕ for games $g{:}\gamma$ and $d{:}\delta$ respectively. By the inductive hypothesis, for all subgames $b \in \mathsf{subgames}(g{:}\gamma)$, $S(b) \rightarrow S'(b)$. Similarly, for all subgames $b \in \mathsf{subgames}(d{:}\delta)$, $S(b) \rightarrow S'(b)$. The only remaining subgame in $\mathsf{subgames}(\alpha)$ is a. Per the inductive hypothesis, $\models S(g) \rightarrow S'(g)$ and $\models S(d) \rightarrow S'(g)$. Taking the disjunction of these implications, we get $\models S(g) \vee S(d) \rightarrow S'(g) \vee S'(d)$. Because S is valid, $\models S(a) \rightarrow S(g) \vee S(d)$. By transitivity of implication, $\models S(a) \rightarrow S'(g) \vee S'(d)$. By definition, $S'(a)$ is $S'(g) \vee S'(d)$. Thus we can conclude that $\models S(a) \rightarrow S'(a)$.
2. $g{:}\gamma \cap d{:}\delta$, then observe that restrictions $S'|_{\mathsf{subgames}(\gamma)}$ and $S'|_{\mathsf{subgames}(\delta)}$ of S' are the MPC solutions for Angel win condition ϕ for games $g{:}\gamma$ and $d{:}\delta$ respectively. By the inductive

hypothesis, for all nodes $b \in \mathsf{subgames}(g{:}\gamma)$, $S(b) \to S'(b)$. Similarly, for all subgames $b \in \mathsf{subgames}(d{:}\delta)$, $S(b) \to S'(b)$. The only remaining subgame in $\mathsf{subgames}(\alpha)$ is a. Per the inductive hypothesis, $\models S(g) \to S'(g)$ and $\models S(d) \to S'(g)$. Taking the conjunction of these implications, we get $\models S(g) \wedge S(d) \to S'(g) \wedge S'(d)$. Because S is valid, $\models S(a) \to S(g) \wedge S(d)$. By transitivity of implication, $\models S(a) \to S'(g) \wedge S'(d)$. By definition, $S'(a)$ is $S'(g) \wedge S'(d)$. Thus we can conclude that $\models S(a) \to S'(a)$.

3. $g{:}\gamma;\, d{:}\delta$, then observe that $S'|_{\mathsf{subgames}(\delta)}$ is the MPC solutions for Angel win condition ϕ for game $d{:}\delta$. By the inductive hypothesis, for all subgames $b \in \mathsf{subgames}(d{:}\delta)$, $\models S(b) \to S'(b)$. Now, let S'' be the MPC solution for Angel win condition $S(d)$ for game $g{:}\gamma$. By the inductive hypothesis, for all subgames $b \in \mathsf{subgames}(g{:}\gamma)$, $\models S(b) \to S''(b)$. By definition, for any subgame $b \in \mathsf{subgames}(g{:}\gamma)$, $S''(b)$ is $\langle g{:}\gamma_{b:}\rangle S(d)$ while $S'(b)$ is $\langle g{:}\gamma_{b:}\rangle \phi$. As argued already, by the inductive hypothesis applied to game δ, $\models S(d) \to S'(d)$. By the dGL monotonicity rule M, $\models \langle g{:}\gamma_{b:}\rangle S(d) \to \langle g{:}\gamma_{b:}\rangle S'(d)$. Thus, $\models S''(b) \to S'(b)$. By the transitivity of implication, $\models S(b) \to S'(b)$. We have shown the correct implication for all the nodes in $\mathsf{subgames}(g{:}\gamma)$ and $\mathsf{subgames}(d{:}\delta)$. The only subgame remaining is a. Because $a{:}\alpha \models S$, $\models S(a) \to S(g)$. We have shown that $\models S(g) \to S'(g)$. By the definition of the MPC solution, $S'(a)$ is $S'(g)$. Thus, $\models S(a) \to S'(a)$.
4. $g{:}\gamma^*$, then because $a{:}(\gamma^*) \models S$, $\models S(a) \to \langle a{:}\alpha \overset{\exists}{\downarrow} S\rangle \phi$. Further, by Lemma B.2, $\models S(a) \to \langle \gamma^* \rangle \phi$. But by the definition of the MPC solution, $S'(a)$ is $\langle \gamma^* \rangle \phi$. By transitivity of implication, $\models S(a) \to S'(a)$. The subgames that remain besides a all belong to γ. Consider S'', the MPC solution for Angel win condition $S(a)$ for game $g{:}\gamma$. By the inductive hypothesis, for every subgame b in γ, $\models S(b) \to S''(b)$. Now, we argue that $\models S''(b) \to S'(b)$, so that by transitivity of implication, $\models S(b) \to S'(b)$ completing the proof. Per the definition of the MPC solution, $S''(b)$ is $\langle g{:}\gamma_{b:}\rangle S(a)$ while $S'(b)$ is $\langle g{:}\gamma_{b:}\rangle S'(a)$. By the monotonicity rule M, since $\models S(a) \to S'(a)$, we have $\models \langle g{:}\gamma_{b:}\rangle S(a) \to \langle g{:}\gamma_{b:}\rangle S'(a)$. Thus, $\models S''(b) \to S'(b)$.
5. $g{:}\gamma^\times$, then because $a{:}(\gamma^\times) \models S$, per Theorem 6.3, $\models S(a) \to \langle \gamma^\times \rangle \phi$. By the definition of the MPC solution, $S'(a)$ is $\langle \gamma^\times \rangle \phi$. So, $\models S(a) \to S'(a)$. The nodes that remain besides a all belong to $\mathsf{subgames}(g{:}\gamma)$. Consider S'', the MPC solution for Angel win condition $S(a)$ for game $g{:}\gamma$. By the inductive hypothesis, for every subgame b in γ, $\models S(b) \to S''(b)$. Now, we argue that $\models S''(b) \to S'(b)$, so that by transitivity of implication, $\models S(b) \to S'(b)$, completing the proof. Per the definition of the MPC solution, $S''(b)$ is $\langle g{:}\gamma_{b:}\rangle S(a)$ while $S'(b)$ is $\langle g{:}\gamma_{b:}\rangle S'(a)$. Since $\models S(a) \to S'(a)$, by the dGL monotonicity rule M, $\models \langle g{:}\gamma_{b:}\rangle S(a) \to \langle g{:}\gamma_{b:}\rangle S'(a)$. Thus, $\models S''(b) \to S'(b)$.

We have shown that $S' \succsim S$ for all S such that $a{:}\alpha \models S$ under the weaker ordering of Def. B.1. From Lemma B.1, it also follows that $S' \sqsupseteq S$ under the stronger ordering of Def. 6.10. Thus, S' is maximal. ☐

Theorem D.1 (Subvalue maps are sound monitors)**.** *For* dGL *game* $a{:}\alpha$ *and subvalue map* S*, if* $a{:}\alpha \models S$*, then* $\models S(a) \to \langle a{:}\alpha \lightning S\phi \rangle \phi$*. Dually, if* $a{:}\alpha[] \models S$ *then* $\models S(a) \to [a{:}\alpha \lightning S\phi]\phi$*.*

Proof. The proof proceeds by induction on the structure of the game $a{:}\alpha$. We show this for the $a{:}\alpha \lightning S$ case; the dual case is analogous. If $a{:}\alpha$ has structure:

- atomic and Angel does not take any decision, i.e., $\alpha \in \{x := e, x := \otimes, ?Q, !Q, \{x' = f(x) \mathbin{\&} Q\}^d\}$, then $\langle a{:}\alpha \lightning S\rangle\phi$ is $\langle\alpha\rangle\phi$. Because $a{:}\alpha \models S$, we know that $\models S(a) \to \langle\alpha\rangle\phi$.
- $x := *$, then $\langle a{:}\alpha \lightning S\rangle\phi$ is $\langle ?\langle\alpha\rangle\phi\,;\, x := \otimes\,;\, !\phi\rangle\phi$. By the $\langle;\rangle$, $[?]$ and $\langle ?\rangle$ axioms, this is equivalent to $\langle\alpha\rangle\phi \wedge \langle x := \otimes\rangle(\phi \to \phi)$. Because $a{:}\alpha \models S$, we know that $\models S(a) \to \langle\alpha\rangle\phi$. By $\langle :=\rangle$, $\langle x := \otimes\rangle\top$ is $\forall x\top$ which is $\top$.
- $\{x' = f(x) \mathbin{\&} Q\}$, then $\langle a{:}\alpha \lightning S\rangle\phi$ is $\langle ?\langle\alpha\rangle\phi\,;\, \{x' = f(x) \mathbin{\&} Q\}^d\,;\, !\phi\rangle\phi$. By the $\langle :=\rangle$, $\langle ?\rangle$ and $[?]$ axioms, this is equivalent to $\langle\alpha\rangle\phi \wedge \langle\{x' = f(x) \mathbin{\&} Q\}^d\rangle(\phi \to \phi)$. Because $a{:}\alpha \models S$, we know that $\models S(a) \to \langle\alpha\rangle\phi$. By $[']$ and $\langle^d\rangle$, $\langle\{x' = f(x) \mathbin{\&} Q\}^d\rangle\top$ is $\forall t \geq 0[x := y(t)]\top$ where $y'(t) = f(y)$. By G, $[x := y(t)]\top$ is $\top$, so $\forall t \geq 0\top$ is $\top$.
- $a{:}(g{:}\gamma \cup d{:}\delta)$ then $\langle a{:}\alpha \lightning S\rangle\phi$ is $\langle ?S(g) \vee S(d)\,;\, \big((g{:}\gamma \lightning S) \cap (d{:}\delta \lightning S)\big)\rangle\phi$. By the $\langle ?\rangle$, $\langle\cup\rangle$, $[?]$ and $\langle;\rangle$ axioms, this is equivalent to $(S(g) \vee S(d)) \wedge (S(g) \to \langle g{:}\gamma \lightning S\rangle\phi) \wedge (S(d) \to \langle d{:}\delta \lightning S\rangle\phi)$. Because $a{:}\alpha \models S$, we know that $\models S(a) \to S(g) \vee S(d)$, and further, $g{:}\gamma \models S$ and $d{:}\delta \models S$. By the inductive hypothesis, $\models S(g) \to \langle g{:}\gamma \lightning S\rangle\phi$ and $\models S(d) \to \langle d{:}\delta \lightning S\rangle\phi$. Thus, $\models S(a) \to \langle a{:}\alpha \lightning S\rangle\phi$.
- $a{:}(g{:}\gamma \cap d{:}\delta)$ then $\langle a{:}\alpha \lightning S\rangle\phi$ is $\langle\big((g{:}\gamma \lightning S) \cap (d{:}\delta \lightning S)\big)\rangle\phi$. We want to show $S(a) \to \langle(g{:}\gamma \lightning S) \cap (d{:}\delta \lightning S)\rangle\phi$. By the inductive hypothesis, $\models S(g) \to \langle g{:}\gamma \lightning S\rangle\phi$ and $\models S(d) \to \langle d{:}\delta \lightning S\rangle\phi$. By $\langle\cup\rangle$ and M, then, $\models (S(g) \wedge S(d)) \to \langle(g{:}\gamma \lightning S) \cap (d{:}\delta \lightning S)\rangle\phi$. By the definition of a valid solution, $S(a) \to S(g) \wedge S(d)$. By transitivity of implication, $\models S(a) \to \langle(g{:}\gamma \lightning S) \cap (d{:}\delta \lightning S)\rangle\top$.
- $a{:}(g{:}\gamma\,;\, d{:}\delta)$ then $\langle a{:}\alpha \lightning S\rangle\phi$ is $\langle(g{:}\gamma \lightning S(\mathsf{end} \mapsto S(d)))\,;\, (d{:}\delta \lightning S)\rangle\phi$. After applying the $\langle;\rangle$ axiom, we want to show $S(a) \to \langle g{:}\gamma \lightning S(\mathsf{end} \mapsto S(d))\rangle\langle d{:}\delta \lightning S\rangle\phi$. By the inductive hypothesis, $\models S(g) \to \langle g{:}\gamma \lightning S(\mathsf{end} \mapsto S(d))\rangle S(d)$ and $\models S(d) \to \langle d{:}\delta \lightning S\rangle\phi$. Since $\models S(d) \to \langle d{:}\delta \lightning S\rangle\phi$, by M, it remains to show that $\models S(a) \to \langle g{:}\gamma \lightning S(\mathsf{end} \mapsto S(d))\rangle S(d)$. Since $a{:}\alpha \models S$, we know that $\models S(a) \to S(g)$. Thus, by transitivity of implication, $\models S(a) \to \langle g{:}\gamma \lightning S(\mathsf{end} \mapsto S(d))\rangle S(d)$, completing the proof.
- $a{:}(g{:}\gamma)^*$ then $\langle a{:}\alpha \lightning S\rangle\phi$ is $\langle ?(S(a))\,;\, (!S(g)\,;\, g{:}\gamma \lightning S(\mathsf{end} \mapsto S(a))\,;\, ?S(a))^\times\,;\, !S(\mathsf{end})\rangle\phi$.
  By the $\langle ?\rangle$, $[?]$ and $\langle;\rangle$ axioms, this is equivalent to
  $(S(a)) \wedge \langle(!S(g)\,;\, g{:}\gamma \lightning S(\mathsf{end} \mapsto S(a))\,;\, ?S(a))^\times\rangle(\phi \to \phi)$.
  Thus, we want to show $\models S(a) \to (S(a)) \wedge \langle(!S(g)\,;\, g{:}\gamma \lightning S(\mathsf{end} \mapsto S(a))\,;\, ?S(a))^\times\rangle\top$.
  We want to show that $S(a) \to \langle(!S(g)\,;\, g{:}\gamma \lightning S(\mathsf{end} \mapsto S(a))\,;\, ?S(a))^\times\rangle\top$ holds. We prove this using the loop rule.
  $\top$ serves as an invariant of this loop game since it holds initially ($\models \top$), implies the postcondition ($\models \top \to \top$), and is inductive ($\langle !S(g)\,;\, g{:}\gamma \lightning S(\mathsf{end} \mapsto S(a))\,;\, ?S(a)\rangle\top$) as argued next. Since $a{:}\alpha \models S$, so $g{:}\gamma \models S(\mathsf{end} \mapsto S(a))$, By the inductive hypothesis, $\models S(g) \to \langle g{:}\gamma \lightning S(\mathsf{end} \mapsto S(a))\rangle(S(a))$. Applying the axioms $[?]$, $\langle;\rangle$ and $\langle ?\rangle$, this is equivalent to
  $\models \langle !S(g)\,;\, g{:}\gamma \lightning S(\mathsf{end} \mapsto S(a))\,;\, ?S(a)\rangle\top$
  showing that $\top$ is an inductive invariant.
- $a{:}(g{:}\gamma)^\times$ then $\langle a{:}\alpha \lightning S\rangle\phi$ is $\langle(g{:}\gamma \lightning S)^\times\rangle\phi$.
  We want to show $\models S(a) \to \langle(g{:}\gamma \lightning S(\mathsf{end} \mapsto S(a)))^\times\rangle\phi$.
  Since $a{:}\alpha \models S$, it must be the case that $\models S(a) \to (S(g) \wedge \phi)$ and $g{:}\gamma \models S$. By the inductive hypothesis, $\models S(g) \to \langle g{:}\gamma \lightning S(\mathsf{end} \mapsto S(a))\rangle S(a)$. We apply the loop rule to prove the conclusion $\models S(a) \to \langle(g{:}\gamma \lightning S)^\times\rangle\phi$. $S(a)$ serves as an invariant of this loop. It holds

initially. It is inductive per the inductive hypothesis (using weakening and $\models S(a) \rightarrow S(g)$), and implies the postcondition ($\models S(a) \rightarrow \phi$).

$\square$

Theorem A.1 (Correctness of reduce)**.** *For any loop-free* dGL *formula F and assumptions $A \in \mathcal{P}_{\mathbb{R}}$ the function* `odereduce` *either sets* $\mathsf{exact}{=}\mathit{true}$ *and the formula $A \rightarrow (\mathsf{reduce}(F, A) \leftrightarrow F)$ is valid, or else it sets* $\mathsf{exact}{=}\mathit{false}$ *and the formula $A \rightarrow (\mathsf{reduce}(F, A) \rightarrow F)$ is valid.*

Proof. Follows from dGL axioms being defined in terms of the decidable fragment of $\mathrm{FOL}_{\mathbb{R}}$, quantifier elimination being decidable, and the properties of `odereduce` (Def. A.1). $\square$

Bibliography

[1] Alessandro Abate, Iury Bessa, Lucas C. Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Automated formal synthesis of provably safe digital controllers for continuous plants. *Acta Informatica*, 57(1-2):223–244, 2020. doi: `10.1007/s00236-019-00359-1`. 1.3, 5.5

[2] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. Safe reinforcement learning via shielding. *Proceedings of the Aaai Conference on Artificial Intelligence*, 32, 2018. doi: `10.1609/aaai.v32i1.11797`. 1.3, 5.5

[3] Matthias Althoff. Guaranteed state estimation in CORA 2021. In Goran Frehse and Matthias Althoff, editors, *8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21), Brussels, Belgium, July 9, 2021*, volume 80 of *EPiC Series in Computing*, pages 161–175. EasyChair, 2021. doi: `10.29007/7M2K`. 4.1

[4] Aaron D. Ames, Samuel Coogan, Magnus Egerstedt, Gennaro Notomista, Koushil Sreenath, and Paulo Tabuada. Control barrier functions: Theory and applications. In *17th European Control Conference, ECC 2019, Naples, Italy, June 25-28, 2019*, pages 3420–3431. IEEE, 2019. doi: `10.23919/ECC.2019.8796030`. 1.3, 6.11

[5] M. Antoniotti and B. Mishra. Discrete event models+temporal logic=supervisory controller: automatic synthesis of locomotion controllers. In *Proceedings of 1995 IEEE International Conference on Robotics and Automation*, volume 2, pages 1441–1446 vol.2, 1995. doi: `10.1109/ROBOT.1995.525480`. 1.3, 5.5

[6] Zvi Artstein. Stabilization with relaxed controls. *Nonlinear Analysis: Theory, Methods & Applications*, 7(11):1163–1173, 1983. ISSN 0362-546X. 1.3, 6.11

[7] Robert Atkey, Bruno Gavranović, Neil Ghani, Clemens Kupke, Jérémy Ledent, and Fredrik Nordvall Forsberg. Compositional Game Theory, Compositionally. In *3rd international Applied Category Theory Conference*, volume 333, pages 198 – 214, Online, United States, July 2020. doi: `10.4204/eptcs.333.14`. URL `https://hal.science/hal-04470659`. 6.11

[8] Robert J. Aumann. Backward induction and common knowledge of rationality. *Games and Economic Behavior*, 8(1):6–19, 1995. ISSN 0899-8256. doi: `10.1016/S0899-8256(05)80015-6`. 6.7

[9] G. Basile and Giovanni Marro. Controlled and conditioned invariant subspaces in linear system theory. *Journal of Optimization Theory and Applications*, 3:306–315, 05 1969. doi: `10.1007/BF00931370`. 5.3.1

[10] Richard Bellman. *Dynamic Programming*. Dover Publications, 1957. ISBN 9780486428093. 1.1, 1.2, 6.3, 6.11

[11] Richard E. Bellman and Stuart E Dreyfus. *Applied Dynamic Programming*. Princeton University Press, Princeton, 1962. ISBN 9781400874651. doi: `10.1515/9781400874651`. 6.3

[12] Calin Belta, Boyan Yordanov, and Ebru Aydin Gol. *Formal Methods for Discrete-Time Dynamical Systems*. Springer Cham, 2017. ISBN 978-3-319-50763-7. 1.3, 5.5

[13] Massimo Benerecetti and Marco Faella. Automatic synthesis of switching controllers for linear hybrid systems: Reachability control. *ACM Trans. Embed. Comput. Syst.*, 16(4), May 2017. ISSN 1539-9087. doi: `10.1145/3047500`. URL `https://doi.org/10.1145/3047500`. 1.3, 6.10, 6.11, 1

[14] Ulrich Berger, Phillip James, Andrew Lawrence, Markus Roggenbach, and Monika Seisenberger. Verification of the European Rail Traffic Management System in Real-Time Maude. *Science of Computer Programming*, 154:61–88, 2018. doi: `https://doi.org/10.1016/j.scico.2017.10.011`. 3.2

[15] Julien Bernet, David Janin, and Igor Walukiewicz. Permissive strategies: from parity games to safety games. *RAIRO Theor. Informatics Appl.*, 36(3):261–275, 2002. doi: `10.1051/ITA:2002013`. URL `https://doi.org/10.1051/ita:2002013`. 6.11

[16] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 339–352, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605584799. doi: `10.1145/1706299.1706339`. URL `https://doi.org/10.1145/1706299.1706339`. 1.3, 6.11

[17] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Kostiantyn Potomkin, and Christian Schilling. Juliareach: a toolbox for set-based reachability. In Necmiye Ozay and Pavithra Prabhakar, editors, *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 39–44. ACM, 2019. doi: `10.1145/3302504.3311804`. 4.1

[18] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. VeriPhy: Verified controller executables from verified cyber-physical system models. In *PLDI*, pages 617–630, 2018. doi: `10.1145/3192366.3192406`. 3.9

[19] Rose Bohrer. Chemical case studies in keymaera x. In *Formal Methods for Industrial Critical Systems: 27th International Conference, FMICS 2022, Warsaw, Poland, September 14-15, 2022, Proceedings*, pages 103–120, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-15007-4. doi: `10.1007/978-3-031-15008-1_8`. C.2

[20] Rose Bohrer and André Platzer. Constructive hybrid games. In *IJCAR*, pages 454–473, Berlin, Heidelberg, 2020. Springer. ISBN 978-3-030-51073-2. doi: `10.1007/978-3-030-51074-9_26`. 1.3, 6.9, 6.11

[21] Andrea Bonacchi and Alessandro Fantechi. On the validation of an interlocking system by model-checking. pages 94–108, 09 2014. doi: `10.1007/978-3-319-10702-87`. 3.2

[22] Michele Boreale. Complete algorithms for algebraic strongest postconditions and weakest preconditions in polynomial ODE's. In A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jirí Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science - 44th International Conference on Current Trends in Theory and Practice of Computer Science, Krems, Austria, January 29 - February 2, 2018, Proceedings*, volume 10706 of *LNCS*, pages 442–455. Springer, 2018. 5.3.4

[23] Joe Brosseau, Bill Moore Ede, Shad Pate, RB Wiley, and Joe Drapa. Development of an operationally efficient PTC braking enforcement algorithm for freight trains. Technical Report DOT/FRA/ORD-13/34, Federal Railroad Administration, 2013. 3.1, 3.1, 3.3, 3.4, 3.8, 3.8

[24] Joseph Brosseau and Bill Moore Ede. Development of an adaptive predictive braking enforcement algorithm. Technical Report FRA/DOT/ORD-9/13, Federal Railroad Administration, 2009. 1.2, 3.1, 3.3, 3.3, 3.4, 3.7, 3.8, 3.8, 3.8.1, 3.2a, 3.2b, 3.4a, 3.4c

[25] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964. ISSN 0004-5411. doi: `10.1145/321239.321249`. 1.1, 1, 2

[26] Lei Bu, You Li, Linzhang Wang, and Xuandong Li. Bach: Bounded reachability checker for linear hybrid automata. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08. IEEE Press, 2008. ISBN 9781424427352. 4.1

[27] Christof Budnik, Marco Gario, Georgi Markov, and Zhu Wang. Guided test case generation through AI enabled output space exploration. pages 53–56, 05 2018. doi: `10.1145/3194733.3194740`. 3.2

[28] Gustavo Carvalho, Ana Cavalcanti, and Augusto Sampaio. Modelling timed reactive systems from natural-language requirements. *Formal Aspects of Computing*, 28(4):725–765, 2016. doi: `10.1007/s00165-016-0387-x`. 8.3

[29] Bob F Caviness and Jeremy R Johnson. *Quantifier elimination and cylindrical algebraic decomposition*. Springer Science & Business Media, 2012. 5.3.7

[30] Yongchao Chen, Rujul Gandhi, Yang Zhang, and Chuchu Fan. NL2TL: Transforming natural languages to temporal logics using large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 15880–15903, Singapore, December 2023. Association for Computational Linguistics. doi: `10.18653/v1/2023.emnlp-main.985`. URL `https://aclanthology.org/2023.emnlp-main.985/`. 8, 8.3

[31] Alessandro Cimatti, Raffaele Corvino, Armando Lazzaro, Iman Narasamdya, Tiziana Rizzo, Marco Roveri, Angela Sanseviero, and Andrei Tchaltsev. Formal verification and validation of ERTMS industrial railway train spacing system. In *CAV*, pages 378–393, 2012. doi: `10.1007/978-3-642-31424-729`. 3.2

[32] Mariano Di Claudio, Alessandro Fantechi, Giacomo Martelli, Simone Menabeni, and Paolo Nesi. Model-based development of an automatic train operation component for communication based train control. In *ITSC*, pages 1015–1020, 2014. doi: `10.1109/ITSC.2014.6957821`. 3.2

[33] Max H. Cohen and Calin Belta. Approximate optimal control for safety-critical systems with control barrier functions. In *2020 59th IEEE Conference on Decision and Control (CDC)*, pages 2062–2067, 2020. doi: `10.1109/CDC42340.2020.9303896`. 1.3, 6.11

[34] Charles R. Cutler and Brian L. Ramaker. Dynamic matrix control—a computer control algorithm. *IEEE Transactions on Automatic Control*, 17:72, 1979. doi: `10.1109/JACC.1980.4232009`. 4

[35] J. Daafouz, P. Riedinger, and C. Iung. Stability analysis and control synthesis for switched systems: a switched Lyapunov function approach. *IEEE Transactions on Automatic Control*, 47(11):1883–1887, 2002. doi: `10.1109/TAC.2002.804474`. 1.3, 6.11

[36] Hongkai Dai, Benoit Landry, Marco Pavone, and Russ Tedrake. Counter-example guided synthesis of neural network lyapunov functions for piecewise linear systems. *2020 59th IEEE Conference on Decision and Control (CDC)*, pages 1274–1281, 2020. 1.3, 5.5

[37] Jeremy Dasher and Kenny Morrison. Evaluation of ptc braking enforcement algorithms for passenger and commuter trains. Technical report, Federal Railroad Administration, 2020. 3.3, 3.5.1

[38] James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *J. Symb. Comput.*, 5(1/2):29–35, 1988. 3.5.1, 1, 5.3.7, 3, 9.1.3

[39] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 2.1, 4.5.1, 5.4, 7.2, 8.2

[40] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. ISSN 0001-0782. doi: `10.1145/360933.360975`. 1.1, 6.3, 6.7, 5, 7.2

[41] Catalin Dima, Mariem Hammami, Youssouf Oualhadj, and Régine Laleau. Deciding the Synthesis Problem for Hybrid Games Through Bisimulation. In Kazuhiro Ogata, Dominique Mery, Meng Sun, and Shaoying Liu, editors, *Formal Methods and Software Engineering*, pages 181–198, Singapore, 2024. Springer Nature Singapore. ISBN 978-981-96-0617-7. doi: `10.1007/978-981-96-0617-7_11`. 1.3, 6.11

[42] Mengge Du, Yuntian Chen, Zhongzheng Wang, Longfeng Nie, and Dongxiao Zhang. Large language models for automatic equation discovery of nonlinear dynamics. *Physics of Fluids*, 36(9), 2024. 8.3

[43] Jony Eckert, Ícaro Teodoro, Luis da Silva Teixeira, Thiago Martins, Paulo Kurka, and Auteliano Santos. A fast simulation approach to assess draft gear loads for heavy haul trains during braking. *Mechanics Based Design of Structures and Machines*, 2021. doi:

10.1080/15397734.2021.1875233. 3.2

[44] Didier Essamé and Daniel Dollé. B in large-scale projects: The canarsie line cbtc experience. pages 252–254, 12 2006. ISBN 978-3-540-68760-3. doi: 10.1007/11955757_21. 3.2

[45] Alessandro Fantechi. Connected or autonomous trains? In *RSSRail*, pages 3–19, 2019. doi: 10.1007/978-3-030-18744-6_1. 3.2

[46] Yao Feng, Jun Zhu, André Platzer, and Jonathan Laurent. Adaptive Shielding via Parametric Safety Proofs. *Proc. ACM Program. Lang.*, 9(OOPSLA1), 2025. doi: 10.1145/3720450. 1, 1.3

[47] Alessio Ferrari, Maurice H. ter Beek, Franco Mazzanti, D. Basile, Alessandro Fantechi, Stefania Gnesi, Andrea Piattino, and Daniele Trentini. Survey on formal methods and tools in railways: The astrail approach. In *RSSRail*, 2019. 3.2

[48] J. Fisac, A. Akametalu, M. Zeilinger, S. Kaynama, J. Gillula, and C. Tomlin. A general safety framework for learning-based control in uncertain robotic systems. *Ieee Transactions on Automatic Control*, 64:2737–2752, 2019. doi: 10.1109/tac.2018.2876389. 1.3, 5.5

[49] Robert W. Floyd. Assigning meanings to programs. In Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin, editors, *Program Verification - Fundamental Issues in Computer Science*, volume 14 of *Studies in Cognitive Systems*, pages 65–81. Springer Netherlands, 1993. doi: 10.1007/978-94-011-1793-7_4. URL https://doi.org/10.1007/978-94-011-1793-7_4. 1

[50] Nathan Fulton and André Platzer. Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018. ISBN 978-1-57735-800-8. 1, 1.3, 9.1

[51] Nathan Fulton and André Platzer. Verifiably safe off-model reinforcement learning. In Tomáš Vojnar and Lijun Zhang, editors, *TACAS*, pages 413–430, Cham, 2019. Springer International Publishing. doi: 10.1007/978-3-030-17462-0_28. 10

[52] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In *CADE*, pages 527–538, 2015. doi: 10.1007/978-3-319-21401-6_36. 3.1, 4.1, 4.2, 5.4, 8.2

[53] Nathan Fulton, Stefan Mitsch, Brandon Bohrer, and André Platzer. Bellerophon: Tactical theorem proving for hybrid systems. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *ITP*, volume 10499 of *LNCS*, pages 207–224. Springer, 2017. ISBN 978-3-319-66106-3. doi: 10.1007/978-3-319-66107-0_14. 4.4.1, 5.1, A.1, A.3.3, 13

[54] Carlos E. García, David M. Prett, and Manfred Morari. Model predictive control: Theory and practice—a survey. *Automatica*, 25(3):335–348, 1989. ISSN 0005-1098. doi: 10.1016/0005-1098(89)90002-2. 4

[55] Luca Geretti, Julien Alexandre Dit Sandretto, Matthias Althoff, Luis Benet, Pieter Collins, Marcelo Forets, Stefan Mitsch, Ismail Patel, Maximilian Perschl, Christian Schilling, and Joris Tillet. Arch-comp25 category report: Continuous and hybrid systems with nonlinear dynamics. In Goran Frehse and Matthias Althoff, editors, *Proceedings of 12th Int. Workshop on Applied Verification for Continuous and Hybrid Systems*, volume 108 of *EPiC Series in Computing*, pages 39–70. EasyChair, 2025. doi: `10.29007/7br2`. 4.5.1, C.2

[56] Susan L. Gerhart, Dan Craigen, and Ted Ralston. Case study: Paris metro signaling system. *IEEE Software*, 11:32–28, 1994. 3.2

[57] Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. Compositional game theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 472–481, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355834. doi: `10.1145/3209108.3209165`. 6.11

[58] Khalil Ghorbal and André Platzer. Characterizing algebraic invariants by differential radical invariants. In Erika Ábrahám and Klaus Havelund, editors, *TACAS*, volume 8413 of *LNCS*, pages 279–294. Springer, 2014. ISBN 978-3-642-54861-1. doi: `10.1007/978-3-642-54862-8_19`. 4.5.1

[59] Khalil Ghorbal, Andrew Sogokon, and André Platzer. Invariance of conjunctions of polynomial equalities for algebraic differential equations. In Markus Müller-Olm and Helmut Seidl, editors, *SAS*, volume 8723 of *LNCS*, pages 151–167. Springer, 2014. ISBN 978-3-319-10935-0. doi: `10.1007/978-3-319-10936-7_10`. 4.1

[60] Bijoy K. Ghosh. Controlled invariant and feedback controlled invariant subspaces in the design of a generalized dynamical system. In *1985 24th IEEE Conference on Decision and Control*, pages 872–873, 1985. doi: `10.1109/CDC.1985.268620`. 5.3.1

[61] Ichiro Hasuo, Clovis Eberhart, James Haydon, Jérémy Dubut, Rose Bohrer, Tsutomu Kobayashi, Sasinee Pruekprasert, Xiao-Yi Zhang, Erik André Pallas, Akihisa Yamada, Kohei Suenaga, Fuyuki Ishikawa, Kenji Kamijo, Yoshiyuki Shinya, and Takamasa Suetomi. Goal-aware rss for complex scenarios via program logic. *IEEE Transactions on Intelligent Vehicles*, 8(4):3040–3072, 2023. doi: `10.1109/TIV.2022.3169762`. 1.3, 6.11

[62] William Walter Hay. *Railroad engineering*. Wiley, New York, 2nd ed. edition, 1982. ISBN 0471364002. 3.3, 3.6.1, 4

[63] Thomas A. Henzinger, Benjamin Horowitz, and Rupak Majumdar. Rectangular hybrid games. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR'99 Concurrency Theory*, pages 320–335, Berlin, Heidelberg, 1999. Springer. ISBN 978-3-540-48320-5. doi: `10.1007/3-540-48320-9_23`. 1.3, 6.11, 8.1

[64] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10): 576–580, 1969. doi: `10.1145/363235.363259`. URL `https://doi.org/10.1145/363235.363259`. 1

[65] Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. LLMs for mathematical modeling: Towards bridging the gap between natural and mathematical languages. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association*

for Computational Linguistics: NAACL 2025, pages 2678–2710, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-7. URL `https://aclanthology.org/2025.findings-naacl.146/`. 8.3

[66] I. E. Irodov. *Problems in General Physics*. Mir Publishers, 1988. 8, 8.1, 8.1, 8.4

[67] Aditi Kabra. Cleo: Command Line EnvelOpe Verification Tool. 2026. doi: `10.1184/R1/32320254`. 7.4.1

[68] Aditi Kabra, Stefan Mitsch, and Andre Platzer. Verified Train Controllers for the Federal Railroad Administration Train Kinematics Model: Balancing Competing Brake and Track Forces (Models and Proofs). 8 2022. doi: `10.1184/R1/19542610.v1`. URL `https://kilthub.cmu.edu/articles/software/Verified_Train_Controllers_for_the_Federal_Railroad_Administration_Train_Kinematics_Model_Balancing_Competing_Brake_and_Track_Forces_Models_and_Proofs_/19542610`. 1.3

[69] Aditi Kabra, Stefan Mitsch, and André Platzer. Verified train controllers for the federal railroad administration train kinematics model: Balancing competing brake and track forces. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4409–4420, 2022. doi: `10.1109/TCAD.2022.3197690`. 1.2, 3, 9.1.1, 9.2.3

[70] Aditi Kabra, Jonathan Laurent, Stefan Mitsch, and André Platzer. CESAR: Control envelope synthesis via angelic refinements. In Laura Kovacs and Bernd Finkbeiner, editors, *TACAS*, LNCS. Springer, 2024. 1.2, 1.3, 6.7, 6.7, 6.7.1, 6.10, 6.10, 6.3b, 9.1.2, 9.1.4, B.4.1, B.4.4

[71] Aditi Kabra, Jonathan Laurent, Stefan Mitsch, and Andre Platzer. Control Envelope Synthesis via Angelic Refinements (CESAR): Artifact, 1 2024. URL `https://figshare.com/articles/software/Control_Envelope_Synthesis_via_Angelic_Refinements_CESAR_Artifact/24922896`. 5.4

[72] Aditi Kabra, Jonathan Laurent, Sagar Bharadwaj, Ruben Martins, Stefan Mitsch, and André Platzer. Can large language models autoformalize kinematics? In Ahmed Irfan and Daniela Kaufmann, editors, *FMCAD*, pages 78–83. TU Wien Academic Press, 2025. doi: `10.34727/2025/isbn.978-3-85448-084-6_13`. 1.3, 9.2.2

[73] Aditi Kabra, Jonathan Laurent, Sagar Bharadwaj, Ruben Martins, Stefan Mitsch, and Andre Platzer. Artifact for Can Large Language Models Autoformalize Kinematics? 2025. `https://kilthub.cmu.edu/articles/software/Artifact_for_Can_Large_Language_Models_Autoformalize_Kinematics_/28934195`. DOI: 10.1184/R1/28934195. 8.1, 8.2, 8.2, 8.4

[74] Aditi Kabra, Jonathan Laurent, Ruben Martins, Stefan Mitsch, and Andre Platzer. Artifact for LLM-Powered Automatic Theorem Proving and Synthesis for Hybrid Systems and Games. 2026. doi: `10.1184/R1/32248389`. 4.1, 7.3

[75] Aditi Kabra, Jonathan Laurent, Ruben Martins, Stefan Mitsch, and André Platzer. Llm-powered automatic theorem proving and synthesis for hybrid systems and game, 2026. 4.2

[76] Aditi Kabra, Jonathan Laurent, Stefan Mitsch, and André Platzer. Hybrid game control envelope synthesis. *Proc. ACM Program. Lang.*, 10(OOPSLA1):850–876, 2026. doi: `10.1145/3798230`. 1.3, 9.1.2

[77] Aditi Kabra, Jonathan Laurent, Stefan Mitsch, and Andre Platzer. Artifact for Hybrid Game Control Envelope Synthesis. 2 2026. doi: `10.1184/R1/30310993`. 6.12

[78] Eduard Kamburjan and Reiner Hähnle. Deductive verification of railway operations. In *RSSRail*, pages 131–147, 2017. ISBN 978-3-319-68499-4. 3.2

[79] Shyam Lal Karra, Kim Guldstrand Larsen, Florian Lorber, and Jiří Srba. Safe and time-optimal control for railway games. In *RSSRail*, pages 106–122, 2019. ISBN 978-3-030-18744-6. 3.2

[80] Donald E. Knuth. *The Art of Computer Programming*. Addison Wesley Longman Publishing Co., Inc., USA, 1997. ISBN 0201896834. 5.4

[81] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975. ISSN 0004-3702. doi: `10.1016/0004-3702(75)90019-3`. 9

[82] Mykel J Kochenderfer, Jessica E Holland, and James P Chryssanthacopoulos. Next generation airborne collision avoidance system. *Lincoln Laboratory Journal*, 19(1):17–33, 2012. 1.3, 5.5

[83] Yanni Kouskoulas, David W. Renshaw, André Platzer, and Peter Kazanzides. Certifying the safe design of a virtual fixture control algorithm for a surgical robot. In Calin Belta and Franjo Ivancic, editors, *Hybrid Systems: Computation and Control, HSCC, Philadelphia, PA, USA*, pages 263–272. ACM, 2013. doi: `10.1145/2461328.2461369`. 6.3a, 6.10, B.4.1, 19

[84] Atanu Kundu, Sarthak Das, and Rajarshi Ray. Sat-reach: A bounded model checker for affine hybrid systems. *ACM Trans. Embed. Comput. Syst.*, 22(2), January 2023. ISSN 1539-9087. doi: `10.1145/3567425`. 4.1

[85] Martin Lara, Rosario López, Iván Pérez, and Juan F. San-Juan. Exploring the long-term dynamics of perturbed keplerian motion in high degree potential fields. *Communications in Nonlinear Science and Numerical Simulation*, 82:105053, 2020. ISSN 1007-5704. doi: `https://doi.org/10.1016/j.cnsns.2019.105053`. URL `https://www.sciencedirect.com/science/article/pii/S1007570419303727`. 5.4

[86] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023. doi: `10.1145/3586037`. URL `https://doi.org/10.1145/3586037`. 4.2

[87] Jonathan Laurent and André Platzer. Oracular programming: A modular foundation for building llm-enabled software, 2025. URL `https://arxiv.org/abs/2502.05310`. 4.4.1, 8.2

[88] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intel-*

ligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi: `10.1007/978-3-642-17511-4_20`. URL `https://doi.org/10.1007/978-3-642-17511-4_20`. 4.2

[89] Zenan Li, Yifan Wu, Zhaoyu Li, Xinming Wei, Fan Yang, Xian Zhang, and Xiaoxing Ma. Autoformalize mathematical statements by symbolic equivalence and semantic consistency. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 53598–53625. Curran Associates, Inc., 2024. URL `https://proceedings.neurips.cc/paper_files/paper/2024/file/6034a661584af6c28fd97a6f23e56c0a-Paper-Conference.pdf`. 8, 8.3

[90] Samuel J. Ling, Jeff Sanny, and William Moebs. *University Physics: Volume 1*. OpenStax, Houston, TX, September 19 2016. ISBN 978-1506698175. URL `https://openstax.org/books/university-physics-volume-1`. 8, 8.1, 8.4

[91] Siyuan Liu, Ashutosh Trivedi, Xiang Yin, and Majid Zamani. Secure-by-construction synthesis of cyber-physical systems. *Annual Reviews in Control*, 53:30–50, 2022. ISSN 1367-5788. doi: `https://doi.org/10.1016/j.arcontrol.2022.03.004`. 1.3, 5.5

[92] Sarah M. Loos and André Platzer. Safe intersections: At the crossing of hybrid systems and verification. In Kyongsu Yi, editor, *ITSC*, pages 1181–1186, 2011. doi: `10.1109/ITSC.2011.6083138`. 1.2

[93] Sarah M. Loos and André Platzer. Differential refinement logic. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *LICS*, pages 505–514. ACM, 2016. doi: `10.1145/2933575.2934555`. 3.1

[94] Sarah M. Loos, André Platzer, and Ligia Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In Michael Butler and Wolfram Schulte, editors, *FM*, volume 6664 of *LNCS*, pages 42–56. Springer, 2011. doi: `10.1007/978-3-642-21437-0_6`. 6.3a, 6.10, B.4.1

[95] Konstantinos Mamouras. Synthesis of strategies and the hoare logic of angelic nondeterminism. In Andrew Pitts, editor, *Foundations of Software Science and Computation Structures*, pages 25–40, Berlin, Heidelberg, 2015. Springer. ISBN 978-3-662-46678-0. doi: `10.1007/978-3-662-46678-0_2`. 1.3, 6.11

[96] MIT OpenCourseWare. 8.01SC Classical Mechanics Problem Set 1. `https://ocw.mit.edu/courses/8-01sc-classical-mechanics-fall-2016/resources/mit8_01f16_pset1_new/`, 2016. Fall 2016, undergraduate level. 8, 8.1, 8.4

[97] Ian M. Mitchell and Jeremy A. Templeton. A toolbox of hamilton-jacobi solvers for analysis of nondeterministic continuous and hybrid systems. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*, pages 480–494, Berlin, Heidelberg, 2005. Springer. ISBN 978-3-540-31954-2. doi: `10.1007/978-3-540-31954-2_31`. 1.3, 6.11

[98] Stefan Mitsch and André Platzer. ModelPlex: Verified runtime validation of verified cyber-physical system models. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *RV*, volume 8734 of *LNCS*, pages 199–214. Springer, 2014. doi: `10.1007/978-3-319-11164-3_17`. 1, 1.3

[99] Stefan Mitsch and André Platzer. Modelplex: verified runtime validation of verified cyber-physical system models. *Formal Methods Syst. Des.*, 49(1-2):33–74, 2016. doi: `10.1007/s10703-016-0241-z`. 3.1, 3.8, 6.9, 8, 9.2.4

[100] Stefan Mitsch, Sarah M. Loos, and André Platzer. Towards formal verification of freeway traffic control. In Chenyang Lu, editor, *ICCPS*, pages 171–180. IEEE, 2012. doi: `10.1109/ICCPS.2012.25`. 1.2

[101] Stefan Mitsch, Marco Gario, Christof J. Budnik, Michael Golm, and André Platzer. Formal verification of train control with air pressure brakes. In *RSSRail*, pages 173–191, 2017. doi: `10.1007/978-3-319-68499-4_12`. 1.2, 3.1, 3.2, 3.5.1, C.2

[102] Stefan Mitsch, Huanhuan Sheng, Bohua Zhan, Shuling Wang, Simon Foster, and Jonathan Julian Huerta Y Munive. Arch-comp23 category report: Hybrid systems theorem proving. In Goran Frehse and Matthias Althoff, editors, *Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH23)*, volume 96 of *EPiC Series in Computing*, pages 170–188. EasyChair, 2023. doi: `10.29007/57g4`. URL `/publications/paper/TtcQ`. 4.2

[103] Stefan Mitsch, Ismail Patel, Hari Hara Sudhan Kannan, Xiangyu Jin, Bohua Zhan, and Shuling Wang. Arch-comp25 category report: Hybrid systems theorem proving. In Goran Frehse and Matthias Althoff, editors, *Proceedings of 12th Int. Workshop on Applied Verification for Continuous and Hybrid Systems*, volume 108 of *EPiC Series in Computing*, pages 152–168. EasyChair, 2025. doi: `10.29007/pd1w`. URL `/publications/paper/ZlZZ`. 4.2

[104] Thomas Moor and Jennifer M. Davoren. Robust controller synthesis for hybrid systems using modal logic. In Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli, editors, *HSCC*, volume 2034 of *LNCS*, pages 433–446. Springer, 2001. 6.1

[105] Carroll Morgan. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., GBR, 1994. ISBN 0131232746. 1.3, 6.11

[106] Andreas Müller, Stefan Mitsch, and André Platzer. Verified traffic networks: Component-based verification of cyber-physical flow systems. In *ITSC*, pages 757–764, 2015. doi: `10.1109/ITSC.2015.128`. 1.2

[107] Daniel Neider and Rajarshi Roy. *What Is Formal Verification Without Specifications? A Survey on Mining LTL Specifications*, pages 109–125. Springer Nature Switzerland, Cham, 2025. ISBN 978-3-031-75778-5. doi: `10.1007/978-3-031-75778-5_6`. URL `https://doi.org/10.1007/978-3-031-75778-5_6`. 8.3

[108] A. Nerode and A. Yakhnis. Modelling hybrid systems as games. In *Decision and Control, 1992., Proceedings of the 31st IEEE Conference on*, pages 2947–2952 vol.3, 1992. doi: `10.1109/CDC.1992.371272`. 1.3, 5.5, 6.1, 9.1.2

[109] Bruno Oliveira, Gustavo Carvalho, Mohammad Reza Mousavi, and Augusto Sampaio. Simulation of hybrid systems from natural-language requirements. In *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, pages 1320–1325, 2017. doi: `10.1109/COASE.2017.8256284`. 8.3

[110] Jiayi Pan, Glen Chou, and Dmitry Berenson. Data-efficient learning of natural language to linear temporal logic translators for robot task specification. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11554–11561, 2023. doi: `10.1109/ICRA48891.2023.10161125`. 8, 8.3

[111] Rohit Parikh. Propositional game logic. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 195–200, 1983. doi: `10.1109/SFCS.1983.47`. 6.11

[112] Jainta Paul, Stefan Mitsch, and Luis Garcia. Hytwin: Hybrid program semantics for digital twin-based security interventions in industrial control systems. In *NASA Formal Methods: 17th International Symposium, NFM 2025, Williamsburg, VA, USA, June 11–13, 2025, Proceedings*, page 294–312, Berlin, Heidelberg, 2025. Springer-Verlag. ISBN 978-3-031-93705-7. doi: `10.1007/978-3-031-93706-4_17`. 1.3

[113] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010. doi: `10.29007/36DT`. 4.2

[114] Marc Pauly. A modal logic for coalitional power in games. *Journal of Logic and Computation*, 12(1):149–166, 02 2002. ISSN 0955-792X. doi: `10.1093/logcom/12.1.149`. URL `https://doi.org/10.1093/logcom/12.1.149`. 6.11

[115] Christian Pek and Matthias Althoff. Fail-safe motion planning for online verification of autonomous vehicles using convex optimization. *IEEE Transactions on Robotics*, 37(3): 798–814, 2020. 5.3.5

[116] André Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2): 143–189, 2008. ISSN 0168-7433. doi: `10.1007/s10817-008-9103-8`. 1.2, 2.1, 3.1, 3.2

[117] André Platzer. *Differential Dynamic Logics: Automated Theorem Proving for Hybrid Systems*. PhD thesis, Department of Computing Science, University of Oldenburg, 2008. 2, 2.3, 2.2

[118] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010. doi: `10.1007/978-3-642-14509-4`. 2.1

[119] André Platzer. Stochastic differential dynamic logic for stochastic hybrid programs. In *CADE*, volume 6803 of *LNCS*. Springer, 2011. doi: `10.1007/978-3-642-22438-6_34`. 9.2.1

[120] André Platzer. Logics of dynamical systems. In *LICS*, pages 13–24. IEEE, 2012. ISBN

978-1-4673-2263-8. doi: `10.1109/LICS.2012.13`. 4.1

[121] André Platzer. Differential game logic. *ACM Trans. Comput. Log.*, 17(1):1:1–1:51, 2015. ISSN 1529-3785. doi: `10.1145/2817824`. 1.1, 1.2, 2, 2.2, 2.3, 2.3, 4.2, 6.1, 6.2, 1, 6.4.2, 6.4.2, 3, 6.7, 6.11, 8, 8.1, 9.1.2, 2, B.6.1, B.6.2

[122] André Platzer. A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.*, 59(2):219–265, 2017. doi: `10.1007/s10817-016-9385-1`. 2.1, 2.1, 3.1, 3.1, 3.2, 3.4, 3.4, 3.7

[123] André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, Cham, 2018. ISBN 978-3-319-63587-3. doi: `10.1007/978-3-319-63588-0`. 2.1, 2.1, 2.2, 2.2, 1, 5.1

[124] André Platzer. Intersymbolic AI: Interlinking Symbolic AI and Subsymbolic AI. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA 2024*, volume 15222 of *LNCS*, pages 162–180. Springer, 2024. doi: `10.1007/978-3-031-75387-9_11`. 9.2.1

[125] André Platzer and Edmund M. Clarke. Computing differential invariants of hybrid systems as fixedpoints. *Form. Methods Syst. Des.*, 35(1):98–120, 2009. doi: `10.1007/s10703-009-0079-8`. 6.7, 6.7.1

[126] André Platzer and Jan-David Quesel. European train control system: A case study in formal verification. In *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, pages 246–265, 2009. doi: `10.1007/978-3-642-10373-5_13`. 1.2, 3.1, 3.2, 3.5.1, 3.5.2, 7, 5.2, 5.3.6, 5.1, 5.5, A.1, A.3.1, B.4.1

[127] André Platzer and Yong Kiam Tan. Differential equation axiomatization: The impressive power of differential ghosts. In Anuj Dawar and Erich Grädel, editors, *LICS*, pages 819–828, New York, 2018. ACM. ISBN 978-1-4503-5583-4. doi: `10.1145/3209108.3209147`. 1

[128] André Platzer and Yong Kiam Tan. Differential equation invariance axiomatization. *Journal of the ACM (JACM)*, 67(1):1–66, 2020. 1.3, 1, 5.3.4

[129] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *ArXiv*, abs/2009.03393, 2020. 4.1, 4.2

[130] Stephen Prajna and Ali Jadbabaie. Safety verification of hybrid systems using barrier certificates. In Rajeev Alur and George J. Pappas, editors, *Hybrid Systems: Computation and Control*, pages 477–492, Berlin, Heidelberg, 2004. Springer. ISBN 978-3-540-24743-2. doi: `10.1007/978-3-540-24743-2_32`. 1.3, 6.11

[131] Marian Qian and Stefan Mitsch. Reward shaping from hybrid systems models in reinforcement learning. In Kristin Yvonne Rozier and Swarat Chaudhuri, editors, *NFM*, volume 13903 of *LNCS*. Springer, 2023. 1, 9.1

[132] Jianxing Qin, Alexander Du, Danfeng Zhang, Matthew Lentz, and Danyang Zhuo. Can large language models verify system software? a case study using fscq as a benchmark. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, HotOS ’25,

page 34–41, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714757. doi: `10.1145/3713082.3730382`. 4.1, 4.2

[133] Hadi Ravanbakhsh and Sriram Sankaranarayanan. Robust controller synthesis of switched systems using counterexample guided framework. In *2016 International Conference on Embedded Software, EMSOFT 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*, pages 8:1–8:10, 2016. doi: `10.1145/2968478.2968485`. 1.3, 5.5

[134] Rajarshi Ray, Amit Gurung, Binayak Das, Ezio Bartocci, Sergiy Bogomolov, and Radu Grosu. Xspeed: Accelerating reachability analysis on multi-core processors. In Nir Piterman, editor, *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, volume 9434 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2015. doi: `10.1007/978-3-319-26287-1_1`. 4.1

[135] Jacques Richalet, André Rault, JL Testud, and J Papon. Model predictive heuristic control. *Automatica*, 14(5):413–428, 1978. doi: `10.1016/0005-1098(78)90001-8`. 4

[136] Daniel Richardson. Some undecidable problems involving elementary functions of a real variable. *J. Symb. Log.*, 33(4):514–520, 1968. 3.5.1

[137] Kristin Yvonne Rozier. Specification: The biggest bottleneck in formal methods and autonomy. In Sandrine Blazy and Marsha Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, pages 8–26, Cham, 2016. Springer International Publishing. ISBN 978-3-319-48869-1. 8

[138] D. Seto, B. Krogh, L. Sha, and A. Chutinan. The simplex architecture for safe online control system upgrades. In *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*, volume 6, pages 3504–3508 vol.6, 1998. doi: `10.1109/ACC.1998.703255`. 1.3

[139] Claude E. Shannon. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950. doi: `10.1080/14786445008521796`. 1.3, 6.11

[140] Huanhuan Sheng, Alexander Bentkamp, and Bohua Zhan. Hhlpy: Practical verification of hybrid systems using hoare logic. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings*, volume 14000 of *Lecture Notes in Computer Science*, pages 160–178. Springer, 2023. doi: `10.1007/978-3-031-27481-7_11`. URL `https://doi.org/10.1007/978-3-031-27481-7_11`. 4.1, 4.2

[141] J. Tanner Slagel, Mariano M. Moscato, Lauren M. White, César A. Muñoz, Swee Balachandran, and Aaron Dutle. Embedding differential dynamic logic in PVS. In Temur Kutsia, Daniel Ventura, David Monniaux, and José F. Morales, editors, *Proceedings 18th International Workshop on Logical and Semantic Frameworks, with Applications and 10th Workshop on Horn Clauses for Verification and Synthesis, LSFA/HCVS 2023, and 10th Workshop on Horn Clauses for Verification and SynthesisRome, Italy & Paris, France, 1-2 July, 2023 & 23rd April 2023*, volume 402 of *EPTCS*, pages 43–62, 2023. doi: `10.4204/EPTCS.402.7`. URL `https://doi.org/10.4204/EPTCS.402.7`.

4.2

[142] Andrew Sogokon, Stefan Mitsch, Yong Kiam Tan, Katherine Cordwell, and André Platzer. Pegasus: Sound continuous invariant generation. *Form. Methods Syst. Des.*, 58(1):5–41, 2022. ISSN 0925-9856. doi: `10.1007/s10703-020-00355-z`. Special issue for selected papers from FM'19. 4.1, 4.4.2, 5.3.4, 5.3.7, 6.7, 6.7.1, 6.10, 8.5, A.1

[143] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013. doi: `10.1007/s10009-012-0249-7`. 1.3, 5.5

[144] Eduardo D. Sontag. A Lyapunov-like characterization of asymptotic controllability. *SIAM Journal on Control and Optimization*, 21(3):462–471, 1983. doi: `10.1137/0321028`. 1.3, 6.11

[145] R. S. Sutton and A. Barto. Reinforcement learning: an introduction. *IEEE Transactions on Neural Networks*, 9:1054–1054, 1998. doi: `10.1109/tnn.1998.712192`. 1.3, 6.1, 6.3, 6.11

[146] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988. doi: `10.1007/BF00115009`. 6.3, 6.11

[147] C. Szepesvári and M. Littman. A unified analysis of value-function-based reinforcement-learning algorithms. *Neural Computation*, 11:2017–2060, 1999. doi: `10.1162/089976699300016070`. 6.3, 6.11

[148] Paulo Tabuada. *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, Berlin, 2009. doi: `10.1007/978-1-4419-0224-5`. 1.3, 5.5

[149] Yong Kiam Tan and André Platzer. Switched systems as hybrid programs$**$this research was sponsored by the afosr under grant number fa9550-16-1-0288. the first author was also supported by a $*$star, singapore. *IFAC-PapersOnLine*, 54(5):247–252, 2021. ISSN 2405-8963. doi: `https://doi.org/10.1016/j.ifacol.2021.08.506`. URL `https://www.sciencedirect.com/science/article/pii/S2405896321012817`. 7th IFAC Conference on Analysis and Design of Hybrid Systems ADHS 2021. 8.1

[150] Alfred Tarski. A decision method for elementary algebra and geometry. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 24–84, Vienna, 1998. Springer Vienna. ISBN 978-3-7091-9459-1. 5.3.7, 6.3

[151] Ícaro P. Teodoro, Jony J. Eckert, Pedro F. Lopes, Thiago S. Martins, and Auteliano A. Santos. Parallel simulation of railway pneumatic brake using openMP. *International Journal of Rail Transportation*, 8(2):180–194, 04 2020. doi: `10.1080/23248378.2019.1660239`. 3.2

[152] Samuel Teuber, Stefan Mitsch, and André Platzer. Provably safe neural network controllers via differential dynamic logic. In *NeurIPS*. Curran Associates, Inc., 2024. doi: `10.5555/3737916.3737967`. 1, 9.1

[153] Claire J. Tomlin, John Lygeros, and Shankar Sastry. A game theoretic approach to controller design for hybrid systems. *Proc. IEEE*, 88(7):949–970, 2000. doi: `10.1109/5.`

`871303`. 1.3, 5.5, 6.1, 8.1

[154] Alan C. van Hulst, Michel A. Reniers, and Wan J. Fokkink. Maximally permissive controlled system synthesis for modal logic. In Giuseppe F. Italiano, Tiziana Margaria-Steffen, Jaroslav Pokorný, Jean-Jacques Quisquater, and Roger Wattenhofer, editors, *SOFSEM 2015: Theory and Practice of Computer Science*, pages 230–241, Berlin, Heidelberg, 2015. Springer. ISBN 978-3-662-46078-8. doi: `10.1007/978-3-662-46078-8_19`. 1.3, 6.11

[155] Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. In *NeurIPS Workshop*, 2025. URL `https://arxiv.org/abs/2509.22819`. 4.1, 4.2

[156] John von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100: 295–320, 1928. doi: `10.1007/BF01448847`. 1.3, 6.11

[157] Linh Hong Vu, Anne E. Haxthausen, and Jan Peleska. Formal modelling and verification of interlocking systems featuring sequential release. *Science of Computer Programming*, 133:91–115, 2017. Formal Techniques for Safety-Critical Systems. 3.2

[158] Martine Wahl. *Survey of railway embedded network solutions - Towards the use of Industrial Ethernet technologies. In: Les Collections de l'Inrets, Synthése S61, 104 pages*. 2010. 3.2

[159] Shuling Wang, Naijun Zhan, and Liang Zou. An improved HHL prover: An interactive theorem prover for hybrid systems. In Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, volume 9407 of *Lecture Notes in Computer Science*, pages 382–399. Springer, 2015. doi: `10.1007/978-3-319-25423-4_25`. 4.1, 4.2

[160] Volker Weispfenning. The complexity of linear problems in fields. *J. Symb. Comput.*, 5 (1-2):3–27, 1988. 3.5.1, 5.3.7

[161] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: `10.1145/3434304`. URL `https://doi.org/10.1145/3434304`. 5.4

[162] Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088. 8, 8.3

[163] Jonathan Julián Huerta y Munive, Simon Foster, Mario Gleirscher, Georg Struth, Christian Pardillo Laursen, and Thomas Hickman. Isavodes: Interactive verification of cyber-physical systems at scale. *J. Autom. Reason.*, 68(4):21, 2024. doi: `10.1007/S10817-024-09709-2`. URL `https://doi.org/10.1007/s10817-024-09709-2`. 4.1, 4.2

[164] Shuo Yang, Xiang Yin, Shaoyuan Li, and Majid Zamani. Secure-by-construction optimal

path planning for linear temporal logic tasks. In *2020 59th IEEE Conference on Decision and Control (CDC)*, pages 4460–4466, 2020. doi: `10.1109/CDC42340.2020.9304153`. 1.3, 5.5

[165] Chaochen Zhou, Ji Wang, and Anders P. Ravn. A formal description of hybrid systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA*, volume 1066 of *Lecture Notes in Computer Science*, pages 511–530. Springer, 1995. doi: `10.1007/BFB0020972`. 4.1

[166] Shufang Zhu and Giuseppe De Giacomo. Synthesis of maximally permissive strategies for ltlf specifications. In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 2783–2789. International Joint Conferences on Artificial Intelligence Organization, 7 2022. doi: `10.24963/ijcai.2022/386`. URL `https://doi.org/10.24963/ijcai.2022/386`. 1.3, 6.11

[167] Liang Zou, Jidong Lv, Shuling Wang, Naijun Zhan, Tao Tang, Lei Yuan, and Yu Liu. Verifying chinese train control system under a combined scenario by theorem proving. In *Verified Software: Theories, Tools, Experiments*, pages 262–280, 2014. 3.1, 3.2