Controller Verification and Design with Logical Analysis Support

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Nikos Aréchiga

B.S., Electrical Engineering with Minor in Mathematics, New Mexico Tech

Carnegie Mellon University
Pittsburgh, PA

May 2015

Keywords: cyber-physical systems, model-based design, formal verification, control design, automated reasoning

dedicated to my family,
for teaching me that knowledge is the greatest wealth

Abstract

Modern computer-controlled systems deployed for safety-critical applications are growing increasingly large and complex. Industry professionals submit their designs to rigorous testing procedures to detect possible errors and re-design the system as necessary. Nonetheless, design errors can go undetected and appear in the final product. In safety-critical systems, these errors may cause severe financial and even human losses. As a result, the modern engineering development process needs to address safety specifications as well as performance specifications.

This dissertation proposes the use of control envelopes, which are abstractions on the input-output relation of a controller. Control envelopes can be used to verify safety of proposed controllers. Since the control envelope does not depend on any specific controller implementation, it can be reused throughout the system development cycle. As a result, safety specifications can be checked with the control envelope by a static check on the input-output of the controller. Second, control envelopes constitute a reusable specification. Initial effort devoted to computing a good control pays for itself throughout the rest of the development process in terms of flexibility and reusability.

We describe a tool called Perseus to automatically check when a controller satisfies a control envelope. We illustrate our approach on control design case studies for autonomous driving scenarios intended to reduce accidents at traffic intersections.

Our case studies make use of the theorem prover KeYmaera to verify plants controlled by control envelopes. KeYmaera uses a powerful representation language called differential dynamic logic, which supports symbolic parameters and can handle nonlinear dynamics without resorting to approximation techniques that incur errors. However, KeYmaera (and theorem proving approaches in general) suffer from a lack of automation, and often require specialized knowledge to operate. We propose the addition of a forward invariant cut proof rule to KeYmaera's reasoning calculus, which allows one to leverage designer insights into proofs of safety of a closed-loop system. We describe the tool Manticore, which aids the search for forward invariants. We illustrate our approach on a case study of a benchmark fuel control system.

Acknowledgments

I thank my advisor Bruce Krogh for his support throughout these years. He has consistently gone beyond the expected role of an advisor by iterating on technical papers and presentations many times, providing detailed feedback.

I acknowledge the other memebers of my committee. James Kapinski was also one of my mentors during my internship at Toyota. André Platzer has been a helpful sounding board for ideas. Scott Smolka was involved with CMACS, and provided helpful remarks and interesting questions after my presentations with the group.

I thank Sicun Gao and Soonho Kong for their help with all dReal-related issues. Sarah Loos for the productive discussions near the early days of the control envelopes work, as well as Stefan Mitsch, Jan Quesel, and André Platzer for their help in learning KeYmaera.

My friends and colleagues in Porter Hall made the years as a Ph. D student feel like a breeze. My academic siblings, Akshay Rajhans, Ajinkya Bhave, Luca Parolini, Matthias Althoff, and Jim Weimer for the early mentoring they gave me. Thanks to Aurora Schmidt, June Zhang, Kyle Anderson, Subro Das, Liangyan Gui, Stefanos Baros, Milos Cvetkovic, Sanja Cvijic, Joel Harley, Evgeny Toropov, Aliaksei Sandrihayla, Rohan Chabukswar, Kyri Baker, Joya Deri, Jonathan Donadee, Jhi-Young Joo, Sergio Pequito, Stephen Kruzick, Matthew Baron, Andrew Hsu, Nipun Popli, Javad Mohammadi, and Anit Sahu. I thank Claire Bauerle, Carol Patterson, Samantha Goldstein, and Elaine Lawrence for helping me keep the administrative side of my Ph. D running smoothly.

I thank my mentors at Toyota Model-Based Development, Jyotirmoy Deshmukh, Xiaoqing Jin and James Kapinski, who earns a double acknowledgement for also being on my committee. Also my managers, Hisahiro Ito, Koichi Ueda, and Ken Butts. I learned a great deal about the craft of scientific research from them. I also thank Jyotirmoy for letting me drive his spare car during my internship. I gained invaluable first-hand experience with the issues of nonautonomous driving in Los Angeles (including the time I lightly collided with the vintage car of an elderly lady—again, I'm very sorry about that Mrs. Tomomi).

During my time at Toyota-MBD, I had the chance to interact with other researchers, who gave me a broader view of the field during lively lunch discussions. Thanks to Oded Maler, Paulo Tabuada, Georgios Fainekos, Sriram Sankaranarayanan, and Thao Dang. I thank the other fellow interns for their camaraderie, Aditya Zutshi, Tommasso Dreossi, Ayca Balkan, Alex Girard, and Karen Sugano. I thank the staff at Toyota who were patient with me when I fell behind on administrative paperwork, Cheryl Jacobs, Wayne Wade, Denise Walker, Karen Yannetti, and Rie Iida.

I also want to thank my parents and my siblings, for the encouragement and support. Last but not least, Kelin Zhao—thank you for being you.

This work was supported by National Science Foundation under grants CNS 1035800-NSF and CCF-0926181.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Existing control design techniques, built on the theory of differential equations, deal mainly with problems like stability, rate of convergence to equilibrium, magnitude of oscillations, and overshoot when the reference changes. Generally, the goal of the conventional control design procedure is to bring the system state close to a desired operating point and resist disturbances, staying always near the desired reference. More complex problems, such as moving a robot arm to perform an action, can be addressed in this framework by designing first a stabilizer to move and maintain the system around a desired reference, and then designing a trajectory generator to choose a sequence of references for the stabilizer to follow. The invention of inexpensive computers spawned theory of discrete-time control, which has only been fully developed for linear systems, and which continues to treat stability as its chief concern.

Nonetheless, modern engineered systems are of staggering complexity, and often must satisfy diverse correctness requirements such as avoiding unsafe conditions. Industry professionals submit their designs to rigorous testing procedures to detect possible errors and re-design the sys-

tem as necessary. Nonetheless, design errors can go undetected and appear in the final product. In safety-critical systems, these errors may cause severe financial and even human losses. Conversely, industry professionals are interested in providing formal certifications that their systems operate correctly to avert costly litigation.

The complexity of engineered systems means that a human cannot expect to be able to keep track of all of the system details required to prove system correctness. The scale and difficulty of the task demands machine assistance to aid and automate the process of reasoning about system designs. Fortunately, a great deal of progress in mathematics and computer science from the last century can step in to provide the required assistance.

This dissertation considers the problem of designing a controller to satisfy safety as well as performance specifications. Existing work in formal methods for hybrid systems has focused either on verification of control systems designed by traditional methods or on direct synthesis of correct controllers. In the pure verification approach, safety constraints are not taken into account at design time, and in the pure synthesis approach traditional control design techniques are discarded. This work presents a framework that can be used as an intermediate between the two extremes. We introduce *control envelopes*, which are abstractions on the input–output relation of a class of safe controllers. Control envelopes are intended to work in synergy with traditional design techniques by bringing safety considerations into the design phase without discarding the wealth of techniques that have been developed for attaining other performance requirements.

To develop our technique, we draw on ideas that have been developed in the field of logic over the course of the past century. We describe these techniques within their historical context to give

the reader a clear picture of the motivation and trajectory of these techniques. In particular, we wish to provide insight into the motivation and context for the development of quantifier elimination, a method of evaluating logical formulas, which is often unfamiliar to a control engineering audience. Also, we wish to emphasize that the use of automated reasoning tools in mathematics has served to supplement the human mind and allow attaining results that had resisted all previous attempts. Similarly, we believe that automatic reasoning techniques in control engineering will lead to an explosion of innovation in the cyberphysical domain by simplifying the design process and automating the transformation from high-level concept to low-level implementation.

1.1 The development of logic and automatic reasoning

At the dawn of the twentieth century, mathematics found itself deep in an existential crisis. The foundations of mathematics were being assailed by the discovery of paradoxes. One such example is Russell's Paradox, which shows that set theory, as formulated by Georg Cantor, contains inconsistencies. In response to this crisis, the great mathematician David Hilbert proposed a program to reformulate mathematics and set it on sound footing [51].

The main goal of Hilbert's program was to construct a precise, formal language, free of the ambiguity and confusion of natural speech,in which all of mathematical reasoning could be expressed. This language should have clearly defined rules for constructing and manipulating its statements. The language should be *consistent*, in the sense that it should not be possible to use the language to derive a contradiction. The language should also be *complete*, in that every true mathematical statement must be provable within the language. Also, the fact that

this language is consistent and complete should be provable by simple methods that cannot be reasonably doubted. Finally, the language should be *decidable*, in the sense that there should be an algorithm to determine whether any given statement of the language is true or false.

If this holy grail of mathematics, consisting of a formal language and a decision procedure, could be attained, the entire endeavor of future mathematicians could be devoted to improving the efficiency of the decision algorithm. After the decision algorithm was fully optimized, all mathematical questions could be handled efficiently and precisely.

Initial results were encouraging. Between 1910 and 1913, Alfred North Whitehead and Bertrand Russell published the *Principia Mathematica* [114]. The Principia attempts to construct a system of symbolic logic to contain all of mathematics. In its final form, the Principia covered the core of set theory, integer arithmetic, and rational and real number theory. However, no proof was given of the consistency or completeness of theory, and no algorithm was given to prove or disprove proposed statements.

In 1929, Presburger published a formalization of the first-order theory of natural numbers with addition and equality [98]. This logical system contains all formulas that quantify variables over natural numbers, and in which only the equality and addition symbols appear, beside variable names, quantifiers, and logical connectives such as "and" and "or". Presburger proved that this system was consistent, complete, and decidable. This positive result electrified the mathematical community, and provided further evidence that Hilbert's program would likely meet with success. Presburger's decision algorithm was the first *quantifier elimination* routine, so named because logical formulas with quantifiers are reduced to formulas without quantifiers. We will have more to say about quantifier elimination in Section 2.2.1. Presburger did not give a runtime

bound for his procedure, but Fischer and Rabin would show in 1974 that the worst-case run-time complexity of this algorithm has a lower bound of $2^{2^{cn}}$, where c is a constant and n is the length of the logical formula [36]. This ominous result presaged later results on the prohibitive complexity of other quantifier elimination procedures.

In 1930, Skolem proved that the first-order theory of the naturals with multiplication and equality was also consistent, complete, and decidable [104]. However, a quantifier elimination procedure would not be provided for Skolem arithmetic (as this theory has been named) until 1981 [23].

With the accumulating momentum of positive results, in September of 1930 Hilbert gave a now famous speech to the Society of German Scientists and Physicians, in which he declared that there are no limits to human understanding with the celebrated phrase "We must know—we will know!". [1] In November of that same year, Kurt Gödel submitted for review a paper titled *On Formally Indecidable Proposiitons of Principia Mathematica and Related Systems* [44].

This paper contained the result we now know as Gödel's First Incompleteness Theorem. The first theorem states that any consistent, effectively generated formal theory that extends the first-order theory of the natural numbers with *both* addition and multiplication is incomplete, in the sense there is a true statement that theory cannot prove. A side effect of this theorem is that theory of natural numbers is undecidable. Since there are true statements that do not have proofs, there can be no algorithm that writes a proof for every true statement.

The term *effectively generated* means that there should be a systematic way of listing the axioms of theory without listing formulas that are not axioms. In this way, one can semi-decide

[1] A recording of this speech can be found at http://math.sfsu.edu/smith/Documents/HilbertRadio/HilbertRadio.mp3

whether or not a given formula is an axiom by listing the axioms. If the set of axioms is finite, the procedure will terminate and know if the given formula is an axiom or not. If the set of axioms is infinite, the procedure will terminate if the formula is an axiom and appears on the list, and otherwise fail to terminate. If we suspend the requirement that theory must be effectively generated, we can construct a theory of the naturals that does not suffer from incompleteness, in the sense that every true statement has a proof and every false statement has a refutation in the theory. However, if the theory is not effectively generated, there is now no way to even semi-decide whether a given formula is an axiom, so deciding whether a given formula is true or not is equally hopeless.

Hilbert's aim of finding an algorithm to decide arbitrary mathematical queries had failed. However, the axiomatic approach to mathematics continued. Also, efforts continued to provide decision procedures for restricted theories.

Starting in 1935, a group of French mathematicians writing under the pseudonym Nicolas Bourbaki began publishing a re-formulation of core mathematics topics, beginning from axioms and giving complete, detailed proofs[73]. The latest instalment was published in 2012.

In 1951 Tarski demonstrated a decision procedure for first-order formulas over *real* numbers with addition, multiplication, equalities, and inequalities (i.e., polynomial equations and inequations) [108]. In 1954, Seidenberg published a version of the algorithm that was so improved that today we refer to quantifier elimination over real numbers as Tarski-Seidenberg quantifier elimination [102].

The latter half of the century saw the rise of automatic reasoning. One of the earliest experiments in this area was the appearance of the famous logical programming language Prolog

[26]. Since then, the field has exploded with a plethora of powerful theorem provers, such as the Theorem Proving System (TPS) [77], Isabelle [79], Coq [14], PVS [82], and many, many others.

Although none of these systems serves as a universal decision procedure for all of mathematics, automated tools have gained traction in proving difficult, long-standing theorems, which has contributed increasing their acceptance in the mathematical community and appreciation of their value.

The first use of automated reasoning in the proof of a major theorem was in the proof of the four color theorem [7][8].

The next important milestone happened in 2002, when Hales submitted a proof of the Kepler conjecture that relied heavily on machine-checked components [47]. However, his paper was not published in the *Annals of Mathematics* until 2005 [48], during which time a group of 12 reviewers reviewed the proof. Over this period of four years, the reviewers did not feel confident that they fully understood all of the details of the proof [106][53]. To address the difficulties in verifying the proof, Hales announced the launch of the Flyspeck project to produce a fully machine-checkable proof of the Kepler Conjecture [49]. The project was completed, with a fully machine-checkable proof, in January of 2015 [46].

The Flyspeck project is significant because it is the first instance in which it has been necessary to check a proof so large and complex that human experts have not been up to the task—necessitating the use of automatic reasoning. Indeed, automatic reasoning tools have the potential to serve as an extension of the human brain, allowing mathematicians to prove increasingly complex theorems with less and less effort. Over time, this will provide a net increase in productive output, in much the same way that compilers for programming languages have increased the

value of programmer work by allowing the programmer to work at a higher level of abstraction, and automating the details at the lower levels.

1.2 Formal methods: applied logic

In the field of computer science, *formal methods* have been developed to check correctness of computer programs for different kinds of specifications. One notable technique is *model checking*, in which a model of the program is searched exhaustively to determine whether or not the desired specification is true [25][99]. A second notable technique is *abstract interpretation* [27]. In this technique, the program can be analyzed by stepping through each command and keeping track of whether the result belongs to certain *abstract domains* (e.g., being positive, negative, or zero) instead of the exact value of the computation. At the end of the execution, it is checked whether the program variables belong to sets that satisfy the desired correctness requirements. A third technique is to construct a deductive proof of program correctness in a specially designed logic. Notable examples include Hoare logic [52] and dynamic logic [50].

In the case of control systems, the presence of the physical plant implies that a suitable verification technique must have the ability to reason about continuous systems. If the control scheme is discrete-time, the overall system is hybrid.

In the spirit of model checking, techniques have been developed to verify properties of continuous and hybrid systems by computing an approximation of the reachable set. To evaluate safety properties, an overapproximation of the reachable set is computed, and if this overapproximation is contained in the safe set, the system is concluded to be safe. To evaluate reachability of

a target, an underapproximation of the reachable set is computed, and if this underapproximation reaches the target set, it is concluded that the system reaches the target. These techniques work well for linear systems, but incur approximation errors, and struggle with nonlinear systems.

Deductive techniques for continuous and hybrid system verification have also been proposed. For sufficiently simple systems, correctness may be inferred by checking certain sufficient conditions, such as those provided by a barrier certificate [96]. For more complex systems, a full proof of system correctness is required. *Differential dynamic logic* (d$\mathcal{L}$) is an extension of dynamic logic which allows modeling and reasoning about programs that include differential equations as possible computation steps, using a construct called a *hybrid program*[86]. In this way, the language and proof system can model and reason about discrete, continuous, and hybrid systems. The semi-interactive theorem prover KeYmaera supports reasoning in d$\mathcal{L}$ [94]. Proofs generated with KeYmaera are fully machine-checkable, which eliminates the scope for human error in judgments of correctness. The deductive nature of d$\mathcal{L}$ allows reasoning about symbolic parameters, complex hybrid dynamics, and nonlinear differential equations without approximation-related artifacts. The main drawback of this deductive approach, however, is lack of automation. Human intervention is required to guide the theorem prover in many cases. Our control envelopes reduce the number of times that manual human intervention is required, since once a control envelope is verified, it can be reused to verify multiple controllers automatically.

A direct application of verification techniques to the correct controller design problem is often framed as first designing a controller that satisfies performance requirements using standard control design methods and then attempting to verify correctness of the closed-loop system. Since verification techniques cannot handle the full complexity of large, detailed models, the

closed-loop system is abstracted, and a verification procedure is applied to see if the abstraction of the closed-loop system satisfies the required specifications. If the verification procedure fails to prove that the closed-loop system behaves correctly, then either the closed-loop abstraction is refined or the controller is redesigned until the verification procedure succeeds. This design-then-verify workflow is inefficient because it does not incorporate all of the system specifications at design time. Furthermore, the system design changes frequently during the control design process, so that several prototypes must be tested and evaluated.

On the other hand, several techniques exist for direct design of correct control systems. Optimal control theory, for example, provides open-loop control strategies that avoid given unsafe sets while reaching for given target sets. Open-loop strategies, however, are not used in real-world control scenarios due to their lack of robustness. Closed-loop control schemes based on optimal control, such as model-based control, do not provide formal guarantees.

Overall, correct-by-construction synthesis techniques prescribe a *specific* control law, whereas our control envelopes characterize a broad class of correct controllers. In this way, control designers have greater freedom in their choice of control design techniques, and the control envelope abstraction can be used and reused to validate different iterations of controllers tuned in different ways throughout the product design lifecycle.

In this work, we propose a design process that allows incorporating sufficient conditions for safety into the design process. We introduce *control envelopes*, which are abstractions of the input-output relation of the controller, and present a control design workflow in which a control envelope is verified in closed-loop with a plant model before the controller is designed. The nondeterministic closed-loop system resulting from controlling the plant model with the

envelope is modeled as a *hybrid program* and verified in the d$\mathcal{L}$ calculus using the theorem prover KeYmaera. The verified control envelope can then be used to provide constraints for the design of a controller that will guarantee safety. The parametric nature of our approach affords robustness to changes in system design, to reduce the amount of effort required for subsequent redesigns, as well as providing robustness to variations in plant parameters from nominal values.

This work still relies on verification of closed-loop models. When verifying a closed-loop model, important designer insights—such as local invariants that arise from stability properties of the system—are not exploited by typical verification techniques. We propose a new proof rule for the d$\mathcal{L}$ calculus called a *forward invariant cut* to leverage local invariance properties within a proof of system safety.

1.3 Summary of contributions

In summary, this work makes the following contributions.

1. We use parametrized control envelopes as reusable abstractions to convert the closed-loop safety specifications into constraints on the input-output properties of the controller, so that safety considerations can be incorporated directly into the controller design process. The parametric nature of our approach affords robustness to parameter uncertainties and design changes.
2. We propose a technique called *refinement checking by parts* for advanced use of parametric control envelopes which reduces the conservativeness of the verification technique.
3. We illustrate the use of control envelopes in case studies involving control of vehicles in a

Cooperative Intersection Collision Avoidance System.

4. We add a *forward invariant cut* proof rule to the d$\mathcal{L}$ calculus, which allows one to leverage knowledge of local invariants in safety proofs with the theorem prover KeYmaera.

5. We illustrate the use of forward invariant cuts to aid in the verification of the recovery routine of an engine fuel control model.

1.4 Overview

Chapter 2 gives an overview of background ideas we will draw on in the remainder of the dissertation. Chapter 3 describes the problems statement and presents control envelopes, verification of control envelopes in KeYmaera, and the technique of refinement for controller verification. In the two chapters that follow, we discuss usage of control envelopes for controller verification. Chapter 4 describes an automatic procedure for controller verification with control envelopes. Chapter 5 describes the technique of refinement checking by parts, which allows one to use parametric control envelopes in a way that reduces their conservativeness. Chapter 6 describes the software tool Perseus, which automates refinement checking and refinement checking by parts. Chapter 7 presents two case studies regarding cooperative intersection collision avoidance systems. Chapter 8 switches gears, and describes the forward invariant cut, a proof rule which he have added to the d$\mathcal{L}$ calculus to leverage knowledge of local invariants in a safety proof. We describe an automotive fuel control case study and our tool, the Manticore preprocessor. Chapter 9 presents our conclusions and proposes directions for future work.

Chapter 2

Preliminaries

This section gives an overview of concepts and techniques that will be used to develop the contributions of the dissertation. We draw on material from logic and automated reasoning as well as the theory of differential equations and simulation-driven techniques to infer invariance properties.

2.1 Differential dynamic logic and KeYmaera

We model nondeterministically controlled systems as hybrid programs in differential dynamic logic (d$\mathcal{L}$) [87], an extension of dynamic logic [50] that is supported by the theorem prover KeYmaera [94]. Dynamic logic provides symbolic constructs called *programs* to reason about change. The truth value of a logical formula may change as the program evolves, in contrast with classical logic, in which the value of a logical formula is immutable. Differential dynamic logic extends dynamic logic to reason about programs that include differential equations, providing a natural framework for modeling and analysis of hybrid systems.

A hybrid program is given by the grammar shown below.

$$\alpha, \beta ::= x := \theta \mid x := * \mid x_1' = \theta_1, \ldots, x_n' = \theta_n \& H \mid ?H \mid \alpha \cup \beta \mid \alpha; \beta \mid \alpha^*$$

where α and β are hybrid programs, θ is a term, x is a variable, each x_i is a variable, and H is a formula of first order logic. A term consists of constants, variables, and functions applied to variables and constants. The program $x := \theta$ means that the variable x is assigned to the value of θ. The program $x := *$ is a special case of the assignment program, in which the variable x is assigned an arbitrary real value. In the system of differential equations $x_1' = \theta_1, \ldots, x_n' = \theta_n \& H$, each x_i is a variable, each θ_i is a term, each θ_i is a term, and the hybrid program means that each x_i evolves according to derivative θ_i for any duration of time, as long as the logical formula H is true. The test $?H$ behaves as a *skip* if the logical formula H is true in the current state and as an *abort* if it is false. The nondeterministic choice $\alpha \cup \beta$ means that either α or β may be executed. Sequential composition $\alpha; \beta$ means that first α is executed, and then β. Nondeterministic repetition α^* means that program α is executed an arbitrary (zero or more) number of times.

Differential dynamic logic annotates modalities with hybrid programs. The grammar of d$\mathcal{L}$ is given by

$$\phi, \psi ::= \theta_1 = \theta_2 \mid \theta_1 \geq \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid [\alpha]\phi \mid \langle\alpha\rangle\, \phi$$

where ϕ and ψ are formulas of d$\mathcal{L}$, θ_1 and θ_2 are terms, and α is a hybrid program. $\theta_1 \geq \theta_2$ is the logical formula that is true if the value of the term θ_1 is greater than or equal than that of θ_2, and the logical operators $\neg$ (negation), $\wedge$ (and), and $\vee$ (or) are interpreted as usual. The formula with the box modality $[\alpha]\phi$ is true if the formula ϕ is true after every execution of α, and the formula

with the diamond modality $\langle\alpha\rangle\,\phi$ means that ϕ is true after some execution of α. See [87], [89] for details.

2.2 Logic and logic solvers

We give a brief discussion of the terminology we use surrounding logical formulas and logic solvers. Our discussion of logic formulas draws on material from [55].

A *closed* formula is a formula in which all variables are bound by quantifiers. A closed formula must be either true or false. An *open* is a formula in which some variables are not quantified, and a *quantifier-free* formula is a formula that contains no quantifiers. The truth value of open formulas depends on the *valuation* assigned to its free variables, whereas a closed formula must be true or false, since none of its variables are free. To emphasize this point, consider the following logical formulas.

1. $\forall x.\ x > 0$ The truth value of this formula does not depend on the value of x, since it is a statement about all x. Its truth value is `false`, since it does not hold when $x = 0$.

2. $\exists x.\ x^2 = y$ The truth value of this formula depends on the valuation of the free variable y. In particular, if $y \geq 0$, the formula is `true`, and for all other valuations, the formula is `false`.

Two logical formulas ϕ_1 and ϕ_2 are said to be *logically equivalent* if

1. ϕ_1 and ϕ_2 are both closed and have value `true`, or

2. ϕ_1 and ϕ_2 are both closed and have value `false`, or

3. every valuation of free variables that makes ϕ_1 true also makes ϕ_2 true.

We use the term *logic solver* to mean a procedure to manipulate logical expressions. In particular, we are concerned with the following two basic operations.

1. Given a closed formula, we would like to know if it is true or false. We call this operation *validity checking*.
2. Given an open formula, we would like to know a valuation of variables that makes the formula true. We call this operation *instance finding*.

Unfortunately, neither of the logic solvers that we describe is able to fully solve the instance finding problem. However, the work in this dissertation only uses instance finding to guide heuristics, and not in any step that is critical for soundness, as we will point out in the relevant sections.

2.2.1 Quantifier elimination

The following discussion draws on material from [85]. Tarski-Seidenberg quantifier elimination pertains to the first-order theory of real closed fields, which consists of formulas with the following characteristics.

1. Allowed arithmetic function are addition and multiplication.
2. Allowed arithmetic relations are order relations, i.e. $\leq, <, =, \geq, >, \neq$.
3. All standard logical operators are permitted, i.e. $\wedge, \vee, \neg, \rightarrow$; conjunction, disjunction, negation, and implication, respectively.
4. All quantified variables must take their values from the set of real numbers, e.g. "For all real numbers such that...", "There exists a real number such that...". This means that

quantifiers may *not* be over functions or sets, e.g. "For all functions such that...", "There exists a set such that...".

In summary, the first-order theory of real closed fields consists of formulas that consist of logical combinations of polynomial equalities and inequalities, and all quantifiers refer to real numbers.

Definition 1 (Quantifier elimination)**.** *A theory is said to admit quantifier elimination if for every (possibly open) quantified formula ϕ there is another, quantifier-free formula $\hat{\phi}$ such that the variables of $\hat{\phi}$ are exactly the free variables of ϕ, and $\hat{\phi}$ is logically equivalent to ϕ.*

If a theory admits quantifier elimination, the theory is decidable if:

1. for every quantified formula, its quantifier-free equivalent can be computed, i.e., there is an algorithm to compute quantifier-free formulas, and
2. the arithmetic of ground formulas (i.e., formulas in which specific valuations have been given to the free variables) is decidable.

As discussed in the introduction, several decision techniques based on quantifier elimination procedures were developed for different theories throughout the early twentieth century. Our interest in this dissertation is with Tarski-Seidenberg quantifier elimination.

Theorem 1 (Tarski-Seidenberg [108], [102])**.** *The first order theory of real closed fields admits quantifier elimination.*

The worst-case runtime of Tarski-Seidenberg quantifier elimination has a lower bound that is doubly exponential in the number of quantifier alternations [30], and in practice is very slow for all but the simplest applications. In our computational examples, we use quantifier elimination as implemented by Mathematica. Quantifier elimination is implemented by the `Resolve` command, but we use `Reduce`, which additionally performs convenient simplifications on the

quantifier-free formula.

In summary, if the formula consists of polynomial arithmetic, we can use quantifier elimination to solve the validity checking problem for closed formulas, and also to obtain constraints on valuations that satisfy open formulas. However, in the tools we develop, we will use Mathematica's `FindInstance` command to handle the instance finding problem.

2.2.2 SMT solver: dReal

dReal [38] is a Satisfiability Modulo Theories (SMT) solver with a twist. Many variations of the SMT problem exist, but we are concerned with the bounded, quantifier-free SMT problem. The bounded, quantifier-free SMT problem is to take a set of quantifier-free logical formulas with arithmetic operators from a predetermined class,

$$\phi_1(x_1, \ldots, x_s) \ , \ \ldots \ , \ \phi_k(x_1, \ldots, x_s) \tag{2.1}$$

along with bounds on the variables

$$-\ell_1 \leq x_1 \leq u_1 \tag{2.2}$$

$$\vdots \tag{2.3}$$

$$-\ell_s \leq x_s \leq u_s \tag{2.4}$$

and produce a valuation that satisfies all of the formulas and is contained within the given bounds, or conclude that no such valuation exists. A general, algorithmic solution to this problem is as hard as quantifier elimination over the given formulas and bounds. In other words, the problem is solvable for the case when the arithmetic is restricted to polynomials (by Tarski-Seidenberg), and no general algorithm exists for other classes of arithmetic. SMT solvers, however, usually

do not adopt the paradigm of providing a general algorithm, but instead provide fast heuristics to attack specific kinds of problems.

dReal begins by altering the problem formulation. Instead of solving the conventional problem, dReal solves the bounded, quantifier-free δ-SMT problem. Before we define this, we define the δ-weakening of a logical formula.

Definition 2. *For a logical formula ϕ, the δ-weakening ϕ^δ for some small parameter $\delta > 0$ is defined recursively on the structure of ϕ as follows.*

1. *If ϕ is of the form $f(x_1, \ldots, x_s) = 0$, the δ-weakening ϕ^δ is $|f(x_1, \ldots, x_s)| \leq \delta$.*
2. *If ϕ is of the form $f(x_1, \ldots, x_s) < 0$, the δ-weakening ϕ^δ is $f(x_1, \ldots, x_s) < \delta$.*
3. *If ϕ is of the form $\phi_1 \vee \phi_2$, the δ-weakening ϕ^δ is $\phi_1^\delta \vee \phi_2^\delta$.*
4. *If ϕ is of the form $\phi_1 \wedge \phi_2$, the δ-weakening ϕ^δ is $\phi_1^\delta \wedge \phi_2^\delta$*

Note that other inequalities can be rewritten in terms of the cases we have defined above and handled appropriately, e.g. $f(x_1, \ldots, x_s) \leq 0$ is equivalent to $f(x_1, \ldots, x_s) < 0 \vee f(x_1, \ldots, x_s) = 0$, $f(x_1, \ldots, x_s) \neq 0$ is equivalent to $f(x_1, \ldots, x_s) < 0 \vee f(x_1, \ldots, x_s) > 0$, etc.

Definition 3 (Bounded, quantifier-free δ-SMT problem)**.** *Let $\delta > 0$ be given. The bounded, quantifier-free δ-SMT problem is to take a collection of quantifier-free logical formulas $\phi_1, \ldots, \phi_k$ along with bounds on their variables $-\ell_1 \leq x_1 \leq u_1, \ldots, -\ell_s \leq x_s \leq u_s$ and produce a valuation that satisfies the δ-weakened formulas $\phi_1^\delta, \ldots, \phi_k^\delta$, or to say if no such valuation exists.*

In [37], an algorithm is given to decide the bounded quantifier-free δ-SMT problem over the class of Type-2 computable functions, which are functions that can be approximated to arbitrary precision by a Turing machine, and includes polynomials as well as transcendentals such as trigonometrics, exponentials, and logarithms. The existence of this algorithm proves that the

problem is decidable. Furthermore, it is proved that the given algorithm is NP-complete, which is a drastic improvement on the doubly-exponential bound of Tarski-Seidenberg quantifier elimination.

It may seem that considering the δ-weakened problem is of limited use when we are interested in exact valuations. However, the case of interest is *not* when the weakened formulas are satisfiable, but when they are unsatisfiable. In this dissertation, we will use the bounded quantifier-free δ-SMT problem as follows. Let

$$\forall x_1 \in I_1. \ \ldots . \forall x_s \in I_s. \, \phi(x_1, \ldots, x_s) \tag{2.5}$$

be a closed, purely universally quantified logical formula with arithmetic drawn from Type-2 computable functions, and let $I_1, \ldots, I_s$ be bounded intervals. This formula is true if and only if there does not exist a valuation of variables $x_1, \ldots, x_s$ within the bounded intervals $I_1, \ldots, I_s$ such that

$$\neg\phi(x_1, \ldots, x_s). \tag{2.6}$$

In particular, such a valuation does not exist if the δ-weakening $\phi^\delta(x_1, \ldots, x_s)$ has no satisfying valuation. This can be determined using dReal. If a valuation is found, nothing can be concluded, since a valuation that satisfies the weakened formula may not satisfy the original formula, but if no valuation is found for the weakened formula, none exists for the original. As a result, we have a fast method to check validity for formulas that are true *robustly*, in the sense that not only their negation is unsatisfiable, but also the δ-weakening of the negation.

In summary, dReal can be used to solve the validity checking problem for logical formulas that contain only universal quantifiers, and in which the variables take their values from bounded

sets. However, dReal cannot be used alone to solve the instance finding problem. In our applications, we will often use dReal to propose instances as part of a heuristic procedure in which it is not important if the valuation satisfies the formula exactly. In the cases when exact formula satisfaction is required, we double-check the result afterwards by substituting the proposed instance into the original formula and checking if it evaluates to true.

2.2.3 Comparison of logical solvers

Compared with quantifier elimination, use of dReal has the following advantages.

1. Theoretical bounds on computational complexity are much better, and on practical examples, the runtime can be orders of magnitude better.
2. Arithmetic is not restricted to polynomials, and includes many functions of practical interest, such as trigonometrics and exponentials.

On the other hand, dReal has the following disadvantages with quantifier elimination.

1. dReal cannot validate formulas with alternating quantifiers, or even with existential quantifiers at all, whereas quantifier elimination can handle arbitrary quantifiers.
2. dReal is limited to checking formulas over bounded sets of possible valuations, whereas quantifier elimination can handle bounded as well as unbounded sets.

In practice, the two procedures complement each other, and throughout this dissertation we sometimes use one and sometimes the other.

2.2.4 dReal-enhanced KeYmaera

For the work in this dissertation, we have constructed a version of KeYmaera that supports dReal as a choice for a back-end solver. In this modified version of KeYmaera, the user may select use of either dReal or Mathematica's quantifier elimination for validity checking. Instance finding, however, is always done with Mathematica. The source code for this modified version of KeYmaera is freely available at [9].

2.3 Invariants for differential equations

Informally, a forward invariant is a set that traps any behaviors that enter. Forward invariants are important in the context of verification, because a system can be proved safe by finding a forward invariant that contains the initial set of the system and excludes the unsafe set. In this way, all behaviors will be trapped from the moment they begin, and remain in the safe region.

A *continuous behavior* is a continuous function $x : \mathbb{R}_{\geq 0} \to \mathbb{R}^n$.

Definition 4 (Forward invariant)**.** *A* forward invariant *for a set of behaviors $\mathcal{B}$ is a set $F \subseteq \mathbb{R}^n$ such that for any $x \in \mathcal{B}$, if $x(t_0) \in F$ for some $t_0 \geq 0$, then $x(t) \in F$ for $t \geq t_0$.*

Def. 4 is not *effective*, in the sense that it does not give a procedure to determine whether a given set is forward invariant, much less any information about how to find a forward invariant in the first place. We review three well studied classes of forward invariants: Lyapunov function sublevelsets, differential invariants, and barrier certificates. For these classes, invariance may be proved by proving a formula of first order logic. This formula, however, may be undecidable if it does not belong to a decidable fragment of first order logic.

Some of the earliest work on forward invariants is a result of Lyapunov's study of the problem of stability [71][72].

Definition 5 (Lyapunov function [62])**.** *Suppose the continuous system $\dot{x} = f(x)$ has an equilibrium at the origin, i.e. $f(0) = 0$. Let $D \subseteq \mathbb{R}^n$ be a domain such that $0 \in D$. A continuously differentiable function $V : D \to \mathbb{R}$ is said to be a* Lyapunov function *if $V(0) = 0$ and $V(x) > 0$ for all $x \in D \setminus \{0\}$, and if $\frac{dV}{dx} f(x) \leq 0$ in D.*

Existence of a Lyapunov function implies that the system is stable. Although stability is a different problem from forward invariant search, the sublevelsets of a Lyapunov function, i.e. sets of the form $\{x \mid V(x) \leq c\}$ for $c \geq 0$, constitute forward invariants.

For the case of stable linear systems, quadratic Lyapunov functions always exist and can be easily computed. There has been much work on computation of Lyapunov functions using techniques such as linear matrix inequalities and sum of squares semidefinite programming for stable systems with piecewise linear[57], polynomial [84], and nonpolynomial [83] dynamics. In [60] we describe a simulation-driven approach to searching for Lyapunov functions and barrier certificates, which are described in 2.3.

In the same year as Lyapunov's first paper on stability, Lie published his first paper on *differential invariants*[67]. The general formulation of differential invariants is given in terms of group action. Although it is important in theory of differential equations and continuous transformation groups, in this work we restrict our attention to a formulation of differential invariants in the context of differential dynamic logic[89]. An account of the relationship between this formulation and Lie's original work can be found in [91].

Definition 6 (Differential invariant)**.** *Consider a (vector) differential equation $\dot{x} = f(x)$. Let F*

be a logical formula of the form

$$\bigwedge_{i=1}^{n} a_i(x) \sim_i b_i(x)$$

where each $\sim_i$ *is one of* $\{=, \geq, \leq, >, <\}$*, and* $a_i(x)$ *and* $b_i(x)$ *are arithmetic expressions that mention* x*. The* syntactic derivative *of* F *is the logical formula*

$$\bigwedge_{i=1}^{n} \frac{\partial a_i(x)}{\partial x} \cdot f \sim_i \frac{\partial b_i(x)}{\partial x} \cdot f$$

F *is a* differential invariant *if its syntactic derivative is true for all* $x \in \mathbb{R}^n$.

It can be shown from the above definition that for a differential invariant F, if F is true at a state $x \in \mathbb{R}^n$, then it will be true at any state reachable by the differential equations. Hence, the set of states at which F is true constitutes a forward invariant. In an abuse of terminology, we will refer to the set at which F is true as simply F, and say that a differential invariant is a forward invariant.

In the definition above, we assumed the top level operator was a conjunction, but disjunctions of logical formulas may also be used. The syntactic derivative of a disjunction, however, is a conjunction, for reasons of soundness discussed in[89]. Differential invariants are the class of forward invariant natively supported in theorem prover KeYmaera.

In many cases, e.g. globally stable systems, Lyapunov sublevelsets are differential invariants and vice versa. Differential invariants, however, exist for systems that are not stable in the sense of Lyapunov—hence, not all differential invariants are Lyapunov sublevelsets. Conversely, not all Lyapunov sublevelsets are differential invariants—an easy example is the time-reversed van der Pol oscillator, in which Lyapunov sublevelsets can be fitted around the origin since the system is locally stable; however, these sublevel sets are not differential invariants, since the syntactic

derivative of a logical formula characterizing those sublevelsets will be false outside of the (local) region of attraction.

A technique to search for differential invariants using parametrized templates in a fixedpoint loop is described in [93]. An algorithm to compute algebraic invariants for systems with algebraic differential equations is presented in [41].

More recently, barrier certificates have been proposed as another class of forward invariants in the context of safety verification[95].

Definition 7 (Barrier certificate)**.** *Consider two sets, D and U. A* barrier certificate *is a continuous function $B : \mathbb{R}^n \to \mathbb{R}$ such that $B(x) > 0$ for all $x \in U$, $B(x) \leq 0$ for all $x \in D$, and whenever $x = 0$, $\frac{dB}{dx} f(x) < 0$.*

It is proved in [95] that the zero level set of a barrier certificate is a forward invariant. Semidefinite programming techniques have been proposed in [95] and [97] to search for Lyapunov functions and barrier certificates.

Barrier certificates are not supported directly in the d$\mathcal{L}$ calculus, but we will need them in our examples. We introduce the following proof rule to support barrier certificates.

$$\frac{I \to (B(x) \leq 0) \qquad (B(x) = 0) \to \frac{\partial B}{\partial x} \cdot f(x) < 0 \qquad (B(x) \leq 0) \to S}{I \to [\{x' = f(x)\}]S} \tag{2.7}$$

The first premise says that the initial set is contained in the zero sublevelset of the barrier function. The second premise says that on the zero level set, the derivative of the barrier function along system dynamics is strictly negative. The third premise says that the zero sublevelset of the barrier function is contained in the safe set. Since these conditions encode the definition of a barrier certificate, we can infer that from the initial state, the continuous system given by $x' = f(x)$ will always remain in the safe set S.

2.4 Simulation-driven techniques for invariant discovery

The foundational work for the simulation-driven search of forward invariants for differential equations is the work of Topcu et al. on the estimation of the region of attraction using simulations and sum-of-squares programming [110]. The problem statement of interest in [110] is computing a sum-of-squares Lyapunov function for a system, along with the largest sublevelset size that fits in the region of attraction. This leads to a bilinear optimization problem. However, using simulations, the problem can be transformed into a linear optimization problem that produces candidate Lyapunov functions. Checking the Lyapunov function is performed by affine sum-of-squares optimization problems. The main limitation of this work is that it is restricted to the sorts of dynamical systems that can be handled with sum-of-squares techniques—mostly polynomials, though special enhancements exist for other classes of functions.

In [59], Kapinski and Deshmukh propose an extension to this method in which a global optimizer is used to improve the candidate Lyapunov functions by searching for initial conditions that falsify the candidate. This technique no longer requires sum-of-squares optimization, and hence can accommodate a larger class of dynamical systems. Furthermore, validation of the candidate is by oversampling the region of interest and showing from a Lipschitz argument that no counterexample to the Lipschitz conditions can exist. As long as one chooses the correct parameter template, this method is applicable to any dynamical system with a Lyapunov function that fits the template and such that it and its Lie derivative are Lipshitz continuous. This is especially useful in cases when an explicit representation of system dynamics is not available, e.g. in engineering environments where the only representation of the system of interest is a Simulink model, or test data from a physical prototype. However, an impractical number of

samples is needed to attain the required density conditions.

In [60], Kapinski et al. propose a different extension to [110] which assumes the system dynamics are available. Again, a global optimizer is used to improve the candidate after each iteration, but a logic solver is used to validate the Lyapunov candidate. The logic solvers used were dReal and Mathematica. In the cases in which dReal was used, the class of dynamical systems that can be handled by the method is effectively extended to Type-2 computable functions. This technique can also be used to produce barrier functions if the logic solver queries are used to check barrier conditions instead of Lyapunov conditions.

We provide a brief summary of the mechanics of [60].

1. First, choose an affine parametric template for the desired Lyapunov (or barrier) function. For example, one may choose $V(x) = z^T P z - \gamma$, where P and γ are the parameters, and z is a vector of monomials in x. Here, we use the term "monomial" loosely, to mean a term that is composed by addition with the other terms of the template. However, the internal structure of z may contain any nonlinear function that the logic solver of interest can handle, such as trigonometric functions. In general, the template for V is not polynomial in x.

2. Next, select a number of sample points from a region of interest, $x_1^i, x_2^i, \ldots, x_k^i$ and run simulations from each of them. Suppose the final point of each initial condition is $x_1^f, x_2^f, \ldots, x_k^f$.

3. Since the Lyapunov function should be positive at each point and decrease along every system trajectory, we can use the simulation traces to produce necessary conditions that are affine in the parameters P, γ so that V is a Lyapunov function. Let z_j^i and z_j^f be the monomial vector z evaluated at the jth initial and final condition, respectively. From the

fact that the Lyapunov function must be positive at every point, we obtain the following.

$$(z_1^i)^T P(z_1^i) - \gamma \geq 0 \tag{2.8}$$

$$\vdots \tag{2.9}$$

$$(z_k^i)^T P(z_k^i) - \gamma \geq 0 \tag{2.10}$$

$$(z_1^f)^T P(z_1^f) - \gamma \geq 0 \tag{2.11}$$

$$\vdots \tag{2.12}$$

$$(z_k^f)^T P(z_k^f) - \gamma \geq 0 \tag{2.13}$$

$$\tag{2.14}$$

From the fact that the Lyapunov function must decrease along all system trajectories, we obtain the following.

$$(z_1^f)^T P(z_1^f) - (z_1^i)^T P(z_1^i) \leq 0 \tag{2.15}$$

$$\vdots \tag{2.16}$$

$$(z_k^f)^T P(z_k^f) - (z_k^i)^T P(z_k^i) \leq 0 \tag{2.17}$$

$$\tag{2.18}$$

4. We can solve a feasibility linear program to obtain candidate constraints. Then, an optimization procedure can be used to search for a trajectory that violates the Lyapunov conditions. Finally, the candidate is validated with a logic solver.

This technique has been validated on a number of interesting examples, including a benchmark engine fuel control model, which is the basis for the case study in Chapter 8.

We will also mention that Balkan et al. have extended this line of work to the simulation-

driven search of contraction metrics, which are useful when reasoning about incremental stability [13].

Chapter 3

Control Envelopes

3.1 Introduction

Fig. 3.1 illustrates the conventional approach to using verification tools to design a controller with safety specifications. We use the term *safety specification* to mean any hard constraints that the system behaviors must respect. First, the controller is designed to meet the performance specifications; second, the closed-loop system is abstracted; and third, a verification procedure is used to attempt to verify that the closed-loop abstraction satisfies the safety specifications. The verification procedure may fail if either the controller is unsafe or the closed-loop abstraction is too coarse. If it is believed that the abstraction is too conservative, the counterexample may be used to refine it. Otherwise, a new controller is designed.

Fig. 3.2 illustrates the proposed approach. First, a control envelope is designed to address the safety specifications; second, a verification procedure is used to certify that the envelope keeps the system state within a forward invariant set that is contained in the safe set; and third a

controller is designed—leveraging traditional design schemes—so that it behaves as a refinement of the control envelope over the safe forward invariant. In this way, the safety of the resulting closed-loop system is guaranteed. Furthermore, if the performance specifications change, the control envelope can be reused in new designs. Alternatively, the control envelope can be used to certify proposed controller designs, without the need to compute a closed-loop abstraction for each controller with the plant.

One drawback of the proposed approach is that performance specifications may be difficult to attain as a refinement of a single control envelope. To address this, we propose the method of *refinement checking by parts*. In this scheme, a collection of control envelopes are designed and verified, obtaining a forward invariant for each envelope. The controller may be certified by showing that at any state in the union of the forward invariants, the controller refines one of the envelopes associated to a forward invariant that contains the state. In this way, regions of state space in which refinement checking is difficult can be addressed with specially fitted control envelopes.

3.2 Related work

3.2.1 Synthesis of safe controllers

This section reviews existing work to address design of controllers that are safe from the outset, instead of designing first and verifying later.

Some existing work in safe controller synthesis focuses on constructing discrete abstractions of the plant, using bisimulation functions [107] and more recently, approximate bisimulation

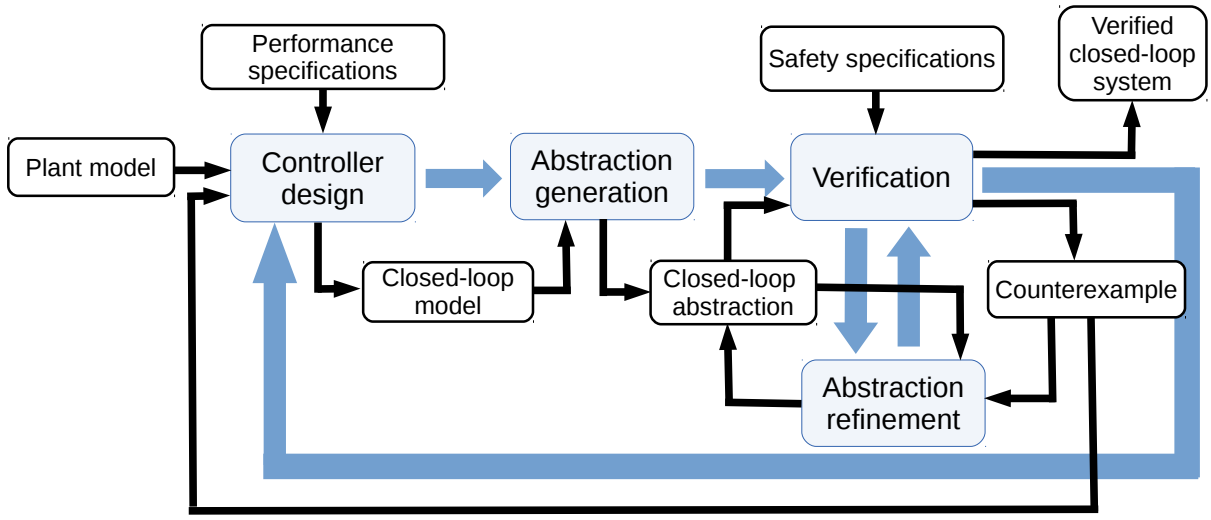


Figure 3.1: Conventional approach: design according to performance specifications, abstract the closed-loop system and apply formal method to verify safety.

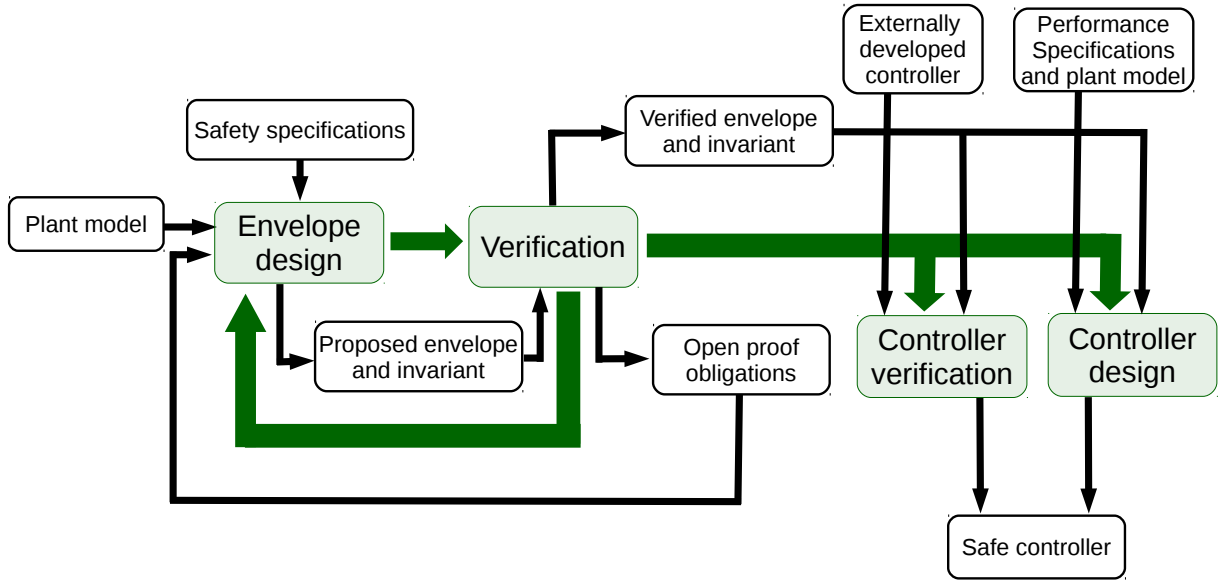


Figure 3.2: Proposed approach: design control envelope according to safety specifications. Proceed to controller design or verification after the control envelope is verified.

functions [42] [43]. Approximate bisimulations are better suited for real-world systems with noise and other imperfections that make exact bisimulations overly restrictive. Once a discrete

abstraction has been constructed, a controller can be efficiently computed using techniques from, e.g., supervisory control or game theory[11].

Ding and Tomlin[34] present a method for synthesis of finite horizon controllers in the presence of a malicious disturbance that has complete knowledge of the control input. The problem is formulated as a differential game and is solved using reachable set computation.

Wolff et al. [115] propose a method using optimization techniques to synthesize controllers that satisfy given linear temporal logic specifications. To avoid generating a discrete abstraction, the specifications are encoded as mixed-integer linear constraints on the state variables. To handle infinite executions, the closed-loop behaviors are required to be periodic.

Our approach does not use discrete abstractions, as is common with bisimulation approaches. Instead, we rely on the modeling capabilities of d$\mathcal{L}$ to directly represent continuous dynamics.

The key distinction of our approach with respect to others, however, is that we are not simply searching for a single safe control policy. Instead, our control envelopes represent an abstract class of control policies. The key benefit is that the control envelope can be reused throughout the controller development process, both to verify proposed designs and to synthesize parameters for a controller template. Our scheme is flexible, and accommodates different techniques that may be used to address performance specifications.

3.2.2 Controlled invariants

A *controlled invariant* is a set N_C of the state space such that at every state in N_C there exists a control input that will keep the system state within N_C for all future time.

Rungger et al. [101] propose a technique to compute the largest controlled invariants for

discrete-time controlled linear systems. [113] Kerrigan and Maciejowski [61] present a similar technique for nonlinear discrete-time systems, for use with model predictive control. The idea is that the model predictive controller should always aim to leave the system within the same invariant set, so that future runs of the controller will be feasible. Indeed, most of the early work concerning controlled invariants focuses on feasibility of model predictive controllers [75] [32]. Since this procedure is computationally expensive, numerous procedures have been proposed to compute approximations quickly [116][15].

Controlled invariants are analogous to the invariants in our envelope–invariant pairs. If a controlled invariant were supplemented with the set of control inputs at each state that preserve the invariant, the combination would constitute an envelope–invariant pair. However, the existing work in controlled invariants is nonconstructive in the sense that it does not concern itself with explicitly characterizing the set of relevant control inputs. The thrust is simply to ensure that some control exists. The controlled invariant is identified simply to ensure that a later synthesis routine will be feasible, typically a technique with an optimal control flavor.

Furthermore, the most valuable contributions of this dissertation appear downstream from the computation of a controlled invariant. In particular, we present techniques to make the best use of an envelope–invariant pair that is overly conservative due to the difficulties of computing optimal envelope–invariant pairs. Our parametric framework provides flexibility, and we also provide mechanical procedures to handle the added complexity introduced by the use of parameters. Further, the technique of refinement checking by parts allows us to give a good fit to control laws that cannot be verified by any single parametrization.

3.2.3 Refinement reasoning

Dijkstra's weakest precondition calculus [33] is one of the earliest instances of nondeterministic programs as abstractions. However, Dijkstra's work mainly focused on guards, i.e. preconditions that are required so that a certain set of computations can be carried out and satisfy a correctness condition.

The refinement calculus [12] uses a high level description language for program specifications, and provides a collection of proof rules to check that successive refinements satisfy the desired specifications, until an executable program is attained.

The Rodin tool allows reasoning about refinement between models in Event-B[2]. Notions of refinement have also been discussed in [78] in the context of using KeYmaera proofs for models with varying levels of detail.

Our use of parametric templates for control synthesis is similar to program sketching [105], in which a human provides a program with holes. An SMT solver is used to fill the holes so that a correctness property of the program holds.

Program synthesis techniques rely on heuristics and human insights. In our case, a wealth of heuristics and insights is provided by control theory, along with a rich collection of templates for different purposes.

3.3 Problem formulation

The main object of study is the set of behaviors of a closed-loop system, which is a structure that consists of a plant and a controller.

We model the plant by the parametrized differential equation $\dot{x} = f_p(x, u)$, with $f_p : \mathbb{R}^n \times \mathbb{R}^r \to \mathbb{R}^n$. The state vector is $x \in \mathbb{R}^n$, the control input is $u \in \mathbb{R}^r$, and p is a parameter vector that takes its value from a set $P \subseteq \mathbb{R}^m$. We also assume that a set of initial plant states $X_p^0 \subseteq \mathbb{R}^n$ has been specified. A safety specification is given as a set X_{safe} of safe states.

A *sampling sequence* is a divergent and nondecreasing sequence (t_k), $k = 0, 1, \ldots$, with $t_0 = 0$. We model sampling uncertainty as nondeterminism. We assume that the controller may sample the state according to any sequence from a parametrized class. This formalism allows us to represent many common sampling models, as shown in the following examples.

Example 1 (Sampling with jitter)**.** Let the parameter vector s represent the nominal period T and the largest deviation Δ_T, as $s = (T, \Delta_T) \in S \subset \mathbb{R}^2$, where $S = \{(T, \Delta_T) \mid T > 0,\ \Delta_T > 0\}$. For each s in S, let J_s be the class of sampling sequences (t_j) such that for each j in $\mathbb{N}_0$, t_j belongs to the interval $[jT - \Delta_T, jT + \Delta_T]$. Then J_s represents sampling sequences with nominal period $T = 1s$ and jitter $\Delta_T = 0.5s$. □

Example 2 (Sampling with bounded network delay)**.** We use the parameter vector to represent the upper and lower bounds on network delay, as $s = (\underline{\tau}, \overline{\tau}) \in S \subset \mathbb{R}^2$, where $S = \{(\underline{\tau}, \overline{\tau}) \mid \overline{\tau} > \underline{\tau} \geq 0\}$. For each s in S, let J_s be the class of sampling sequences (t_j) such that for each $j \in \mathbb{N}_0$, $\underline{\tau} \leq t_{j+1} - t_j \leq \overline{\tau}$. Then J_s represents the class of sampling sequences with sampling step of length at least $\underline{\tau}$ and at most $\overline{\tau}$. □

As before, we will represent a control law as a function $g : \mathbb{R}^n \to \mathbb{R}^r$. We assume the controller has no internal state.

These components interact as a closed-loop system to produce behaviors as follows.

Definition 8 (Behaviors of a closed-loop system)**.** *Consider a plant model $\dot{x} = f_p(x, u)$ with*

parameters $p \in P$, a class of sampling sequences J_s with parameters $s \in S$, and a control law $g(x)$. These three components form a closed-loop system. A function $x : [0, \infty) \to \mathbb{R}^n$ is a behavior *of this closed-loop system if there is some $p \in P$ and some $s \in S$ such that $x(0) \in X_p^0$ and there is a sampling sequence (t_j) in J_s such that for each index $j = 0, 1, 2, \ldots$, $x(t)$ satisfies the controlled differential equation $\dot{x} = f_p(x, g(x(t_j^-)))$ with boundary condition $x(t_j) = x(t_j^-)$ over the time interval $[t_j, t_{j+1}]$.*

We consider the problems of controller verification and synthesis for safety and reachability.

Definition 9 (Robust safety verification)**.** *Given a closed-loop system composed of a plant model $\dot{x} = f_p(x, u)$ with parameters $p \in P$, a class of sampling sequences J_s with timing parameters $s \in S$ and a control law $g(x)$, the* robust safety verification problem *is to decide whether for all $p \in P$ and all $s \in S$, every behavioR $x(t)$ is contained in X_{safe} for $t \geq 0$.*

Definition 10 (Robust safety controller synthesis)**.** *Given a closed-loop system composed of a plant model $\dot{x} = f_p(x, u)$ with parameters $p \in P$, a class of sampling sequences J_s with timing parameters $s \in S$ and a parametrized controller template $g_c(x)$ with parameters $c \in C \subseteq \mathbb{R}^v$, the* robustly safe controller synthesis problem *is to choose $c \in C$ such that for all $p \in P$ and all $s \in S$, every behavior $x(t)$ of the closed-loop system is contained in X_{safe} for $t \geq 0$.*

Example 3. We will use a simplified version of a stop sign assist system as a running example. A car is approaching an intersection and must stop without violating the stop sign. In this simplified version, we consider first-order dynamics in which the state variable is the position of the car, d. The control variable is some input u such that the velocity of the car is given by

$$v = \begin{cases} pu & if \quad u \geq 0 \\ 0 & if \quad u < 0 \end{cases}, \tag{3.1}$$

where p is a parameter that varies between the cars of interest and is known to be within the interval $0.9 \leq p \leq 1.1$. This interval is the set of plant parameter values P. The velocity is defined in this way so that the velocity cannot become negative even if the control input is negative.

The parametrized dynamics of the system are given by the differential equation $\dot{d} = pu$ We assume that the control scheme is discrete-time, and that the controller samples the position of the car at least every τ seconds, with $\tau = 0.1s$, so that the set of sampling parameters S is a singleton. The car starts $20m$ behind the intersection, so that the initial set is the interval $-21 \leq d \leq -19$.The safety specification is that the car must never exceed the stop sign, located at position $d = 0$. In other words, we require $d \leq 0$ as a safety condition.

For the robust safety verification problem, we propose the control law

$$u = -d. \tag{3.2}$$

A sample behavior is shown in Figure 3.3. The start of the intersection is shown by the red line at $d = 0$, and the end of the intersection is shown by the red line at $d = 7$. □

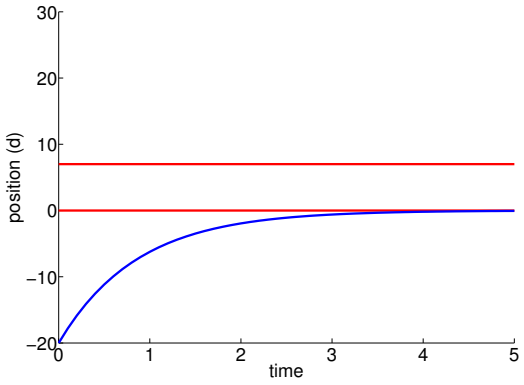


Figure 3.3: A sample behavior of the simple stop sign assist example

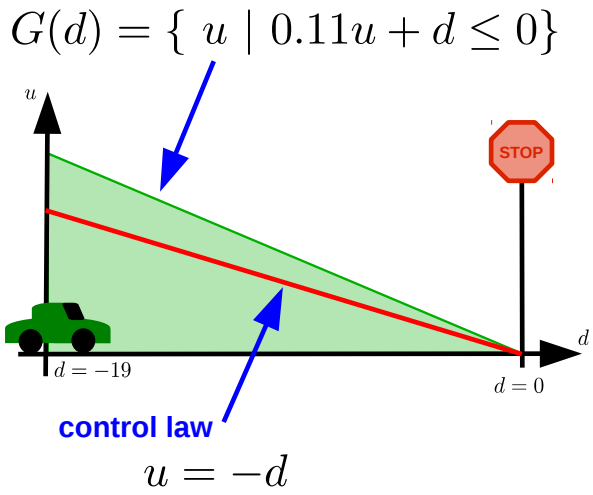


Figure 3.4: Illustration of refinement: control envelope contains control law

Figure 3.4 shows a plot of the control envelope and the control law as a function of the distance of the car to the stop sign. The control law is completely contained in the control envelope.

3.4 Control envelopes for design and verification

We stated the verification and controller design problems above. Next, we introduce control envelopes.

3.4.1 Control envelopes

A control envelope is an abstraction of the input–output relation of the controller.

Definition 11. *A* control envelope $G(x) : \mathbb{R}^n \to 2^{\mathbb{R}^r}$ *is a function that assigns to each state a set of control inputs.*

We next describe the behaviors of a closed-loop structure resulting from the interconnection

of a control envelope and plant.

Definition 12 (Behaviors of an envelope-controlled system, discrete time)**.** *Let $\dot{x} = f_p(x, u)$ be a plant model with parameters $p \in P$, J_s be a class of sampling sequences with $s \in S$, and $G(x)$ be a control envelope. These three components form a discrete-time envelope-controlled system. A function $x : [0, \infty) \to \mathbb{R}^n$ is a* behavior *of this system if there is $p \in P$ and $s \in S$ such that $x(0) \in X_p^0$ and there is a sampling sequence (t_j) in J_s such that for each index $j = 0, 1, 2, \ldots$; the function $x(t)$ satisfies the controlled differential equation $\dot{x} = f_p(x, u)$, for some $u \in G(x(t_j^-))$, over the time interval $[t_j, t_{j+1}]$.*

Definition 13. *Given a set $D \subseteq \mathbb{R}^n$, we say that a control law $g : \mathbb{R}^n \to \mathbb{R}^r$* refines *a control envelope $G : \mathbb{R}^n \to 2^{\mathbb{R}^r}$ over D if for every $x \in D$, $g(x) \in G(x)$.*

Note that refinement of a control envelope provides a stepping stone to safety verification. Suppose that a control law refines a control envelope over a set D. Further suppose that $X_p^0 \subseteq D$. Then each behavior of the closed-loop system can be matched by a behavior of the envelope-controlled system until it leaves the set D. If we can additionally establish that no behavior ever leaves D, i.e. by showing that D is a forward invariant, then the set of behaviors of the closed-loop system will be a subset of the behaviors of the closed-loop system.

Example 4. In our running example, we can compute the largest possible control envelope by inspection. At each car position, the control envelope may choose any input such that the car has not exceeded the stop sign at the next sampling time. This must hold for all possible plant and sampling parameters. In this case, $p = 1.1$ can easily be seen to be the worst case, in the sense that any control input that keeps the system safe with this parameter value will keep the system

safe for all $p \in P$. The control envelope is then

$$G(d) = \{\ u \mid (1.1)(0.1)u + d \leq 0\ \}. \tag{3.3}$$

The safety condition itself is an invariant,

$$N = \{d \mid d \leq 0\ \} \tag{3.4}$$

□

3.4.2 Parametric envelope invariant pairs

In cases of practical interest, the optimal control envelope cannot be found directly by inspection. Instead, a conservative underapproximation of the optimal control envelope may be readily computable.

Underapproximations, however, can often be made more flexible by replacing numeric values with symbolic parameters. The (now parametric) control envelope may be valid for some range of parameter values around the original numeric value. Of course, a verification procedure is required to ensure that this is the case.

Example 5. Consider again the simplified stop sign assist, and suppose that the following (underapproximating) control envelope has been given.

$$G(x) = \{u \mid ((0.1)(5) + d \leq 0\ \wedge\ u \leq 5) \vee (u \leq 0)\} \tag{3.5}$$

The above control envelope allows the car to move forward with $u \leq 5$ if it is possible to choose this input without violating the stop sign before the next sample time. Otherwise, the car must

remain stopped. This control envelope produces the invariant

$$N(x) = \{\ d \mid d \leq -(0.11)5\}. \tag{3.6}$$

However, this control envelope is very rigid. A plot is shown in 3.6. The horizontal axis is car position, and the vertical axis is allowed input. Over the invariant, the allowed control inputs are represented by the blue square. This envelope–invariant pair will not be able to verify the correctness of the controller we have proposed. Furthermore, it will not allow us to verify the correctness of any controller that would allow the car to get closer to the intersection than $0.55m$.

We can buy flexibility by replacing the number 5 in the envelope–invariant pair with a parameter, e, obtaining the following parametric class of envelope–invariant pairs.

$$G_e(x) = \{u \mid ((0.1)(e) + d \leq 0\ \wedge\ u \leq e) \vee (u \leq 0)\} \tag{3.7}$$

$$N_e(x) = \{\ d \mid d \leq -(0.11)e\}. \tag{3.8}$$

Now, by changing the value of e, we can vary the shape of the envelope–invariant pair. By choosing a small value of e, we can verify a controller that gets closer to the intersection. However, the trade-off is that small values of e will restrict the largest permissible value of u. The penalty for allowing the car to get closer to the intersection is that the car must move slower throughout. Figure 3.5 shows how the parametric envelopes are related to the optimal control envelope.

We will leave the discussion of this matter at this point for now, but in Chapter 5 we will discuss refinement by parts, which is a technique to have our cake, and eat it too. With refinement

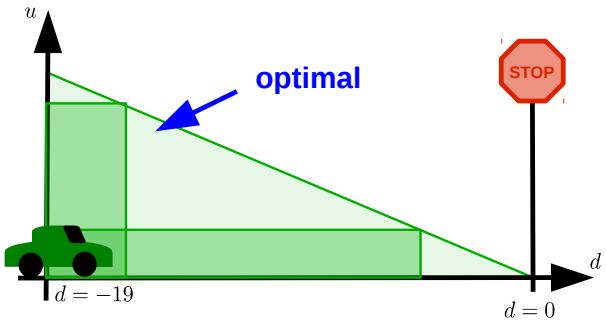


Figure 3.5: The parametric envelopes are underapproximations of the optimal control envelope

by parts, we will be able to take underapproximating, parametric envelopes, and use multiple parametrizations. For the simplified stop sign assist, we will be able to verify a controller that moves quickly *and* gets close to the intersection.

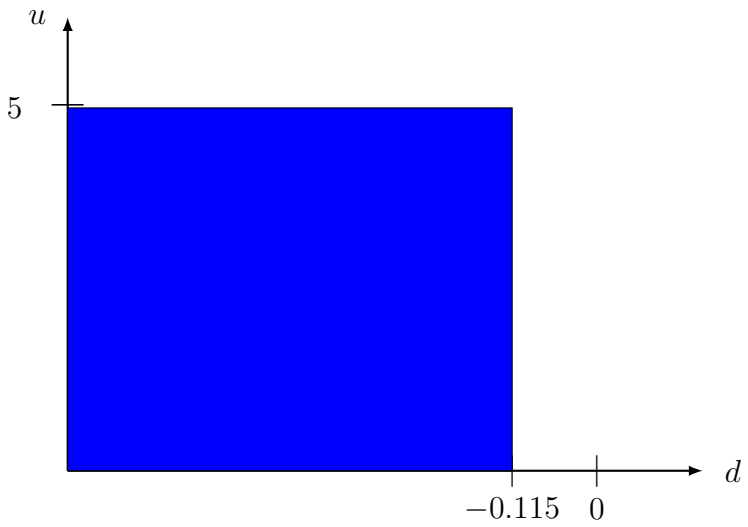


Figure 3.6: Control envelope for a fixed choice of e.

□

Hereafter, we will talk about parametric envelope invariant pairs, and use the notation $\{(G_e, N_e)\}_{e \in E}$, where e is a parameter vector and E is a set of possible values of e.

The task of producing parametric envelope–invariant pairs (and even non-parametric enve-

lope invariant pairs) is, at present, more of an art than a science, and requires a great deal of human insight and manual labor. However, as discussed in the chapter on future work, we have some ideas to make inroads into the problem.

3.4.3 Verifying parametric envelope–invariant pairs with KeYmaera

The goal of the verification procedure is to show that a certain set is a forward invariant of the envelope-controlled system.

The theorem prover is used to verify a nondeterministically controlled closed-loop system with plant and sampling parameters $\mathbf{P}$ and $\mathbf{S}$ such that $P \subseteq \mathbf{P}$ and $S \subseteq \mathbf{S}$, so that all possible plants and sampling situations of interest are covered. The designer provides candidates for the envelope–invariant pair. The parametrized class of envelopes is given as $\{G_e\}_{e \in \mathbf{E}_{p,s}}$, where $\mathbf{E}_{p,s}$ is a set of parameter values for the envelope, which may depend on the specific plant and timing parameters. The goal of each envelope is to keep the plant within the corresponding element of a parametrized class of safe forward invariants $\{N_e\}_{e \in \mathbf{E}_{p,s}}$, i.e. $N_e \in X_{\text{safe}}$ for each e.

We use the theorem prover KeYmaera to verify an envelope-controlled system. KeYmaera's support for symbolic parameters are useful to address our concerns of parametric robustness. Given a plant model and a class of sampling sequences, the first step is to propose a parametrized class of control envelopes along with candidate forward invariants. The second step is to construct a hybrid program that represents the envelope-controlled system, and to embed it in a d$\mathcal{L}$ formula that states that the candidate forward invariant is indeed a forward invariant for the envelope-controlled system. The third and final step is to use KeYmaera to verify that the d$\mathcal{L}$ formula we have constructed is true.

The next task is to construct a hybrid program that represents the envelope-controlled system, and verify that each control envelope G_e maintains forward invariance of its corresponding N_e. The resulting d$\mathcal{L}$ formula tends to have a predictable structure, which only varies with the class of sampling sequences under consideration. We show two examples, one for each of the class of sampling sequences we have discussed above, (a) sampling with jitter, and (b) sampling with bounded network delay.

The case of sampling with jitter is shown in Fig. 3.7a, and the case of bounded network delay is shown in Fig. 3.7b. Line 3.1 (respectively Line 3.13) shows the overall structure of the logical formula we wish to prove, which is the same for both cases. The top level logical operator is an implication, which requires that whenever the logical formula `init` is satisfied, we require that always after arbitrarily many runs of $(ctrl \cup plant)$, the formula `inv` should hold. In this way, `init` serves as a specification of initial conditions.

The structure of the overall hybrid program is that on each run it nondeterministically elects to run either the control program or the plant program, though as we will see, the control envelope program may only run when the class sampling sequences under consideration allow a sampling operation, and the plant may only run until the sampling sequences enforce a sampling operation.

The formula `init` requires that the plant parameters be chosen from the set of interest, $p \in P$ `inv` on Line 3.2 (Line 3.14, respectively). Line 3.3 (respectively Line 3.15) chooses values for the parameters, as we chose them in Ex. 1 and Ex. 2. Line 3.4 (respectively Line 3.16) chooses a permissible envelope parameter vector, and requires that the system state x start in the candidate forward invariant N_e. Line 3.5 (respectively Line 3.17) requires that the initial controller value should respect the control envelope. Line 3.6 (respectively Line 3.6) sets the initial time to zero.

Note that line 3.6 has an additional variable j, whose meaning will shortly be explained.

In the case of sampling with jitter, the control envelope may sample at any time $t \in [jT - \Delta_T, jT + \Delta_T]$. To enforce the left side of this interval, line 3.7 checks whether $t \geq jT - \Delta_T$, where j has been initialized to zero. If this check succeeds, the program may choose any input in the control envelope at the given state 3.7. Then the variable j is incremented (line 3.9), so that this check can be performed again for the next sample. If the check fails, the control program is not allowed to run. To enforce the right side of the timing interval, the plant program on line 3.10 may execute the plant dynamics with the addition of the time dynamics as long as $t \leq jT + \Delta_T$. If this does not hold, the plant must stop executing. The overall result is an envelope-controlled system, using sampling with jitter.

In the case of sampling with bounded network delay, the control envelope may sample at the earliest $\underline{\tau}$ time has elapsed since the previous sample, and must sample at the latest when $\overline{\tau}$ time has elapsed since the previous sample. To enforce the lower bound on sampling time, line 3.19 checks whether $\underline{\tau}$ time has elapsed. If so, the control program chooses any input from the control envelope (line 3.20), and then resets the timer (line 3.21). If the test fails, the control program is not allowed to run. To enforce the upper bound on sampling, the plant program on line 3.10 may execute the plant dynamics in conjunction with the timer dynamics as long as $t \leq \overline{\tau}$, at which time the plant program must stop executing to allow the control program to execute.

In both cases, if the logical formula is found to be true, it is verified that for each $p \in P$, every $s \in S$, and all $e \in E$, the set N_e is a forward invariant for the envelope-controlled system under the control envelope G_e.

$$\texttt{sample} \equiv \texttt{init} \rightarrow [(ctrl \cup plant)^*]\texttt{inv} \quad (3.1)$$

$$\texttt{init} \equiv p \in \mathbf{P} \quad (3.2)$$

$$\wedge\ T = 1 \wedge\ \Delta_T = 0.5 \quad (3.3)$$

$$\wedge\ e \in \mathbf{E}_{p,s} \wedge\ x \in N_e \quad (3.4)$$

$$\wedge\ u \in G_e(x) \quad (3.5)$$

$$\wedge\ t = 0 \wedge\ j = 0 \quad (3.6)$$

$$ctrl \equiv ?t \geq jT - \Delta_T; \quad (3.7)$$

$$u := *;\ ?u \in G_e(x); \quad (3.8)$$

$$j := j + 1 \quad (3.9)$$

$$dyn \equiv \{x' = f_p(x,u), t' = 1 \quad (3.10)$$

$$\&\ t \leq jT + \Delta_T\} \quad (3.11)$$

$$\texttt{inv} \equiv x \in N_e \quad (3.12)$$

(a) Sample program using sampling with jitter

$$\texttt{sample} \equiv \texttt{init} \rightarrow [(ctrl \cup plant)^*]\texttt{inv} \quad (3.13)$$

$$\texttt{init} \equiv p \in P \quad (3.14)$$

$$\wedge\ \underline{\tau} = 0.1 \wedge\ \overline{\tau} = 2 \quad (3.15)$$

$$\wedge\ e \in E \wedge\ x \in N_e \quad (3.16)$$

$$\wedge\ u \in G_e(x) \quad (3.17)$$

$$\wedge\ t = 0 \quad (3.18)$$

$$ctrl \equiv ?t \geq \underline{\tau}; \quad (3.19)$$

$$u := *;\ ?u \in G_e(x); \quad (3.20)$$

$$t := 0 \quad (3.21)$$

$$dyn \equiv \{x' = f_p(x,u), t' = 1 \quad (3.22)$$

$$\&\ t \leq \overline{\tau}\} \quad (3.23)$$

$$\texttt{inv} \equiv x \in N_e \quad (3.24)$$

(b) Sample program using sampling with bounded network delay

Figure 3.7: Sample hybrid programs for control envelope verification under different sampling schemes

3.5 Using control envelopes

In this section, we assume that a parametrized class of control envelope have been proven to have an invariance property . This means that there is an invariant corresponding to each control

envelope parametrization.

In this section, we present the key results that allow us to use control envelopes to translate safety and reachability requirements into first-order logic constraints on the input-output relation of the controller. In later chapters, we describe specific computational procedures to verify and synthesize controllers.

We consider the subset of plant parameters $P \subseteq \mathbf{P}$ that corresponds to the physical plants we wish to verify. Similarly, we are interested in a subset of the verified timing parameters $S \subseteq \mathbf{S}$. The following lemma says how to obtain envelope parameters that will provide invariants robustly, independent of the specific choice of $p \in P$ and $s \in S$.

Lemma 1. *Suppose it has been verified that the class of envelope invariant pairs* $\{(G_e, N_e)\}_{e \in \mathbf{E}_{p,s}}$ *is such that for fixed* $p \in \mathbf{P}$ *and* $s \in \mathbf{S}$*, for any* $e \in \mathbf{E}_{p,s}$*, the system nondeterministically controlled with envelope* G_e *has invariant* N_e*. Suppose sets of parameters of interest* $P \subseteq \mathbf{P}$ *and* $S \subseteq \mathbf{S}$ *are given. Define*

$$E = \bigcap_{\substack{p \in P \\ s \in S}} \mathbf{E}_{p,s}. \tag{3.25}$$

Then for all $e \in E$*, the nondeterministically controlled system controlled by* G_e *has invariant* N_e *for all* $p \in P$ *and all* $s \in S$*.*

The envelopes parametrized by $e \in E$ will keep the plant safe *robustly*, i.e., without regard for the specific value of P, and will maintain the plant within the class of forward invariants $\{N_e\}_{e \in E}$, independent of the specific choice of the parameter vectors $p \in P$ and $s \in S$.

3.5.1 Controller verification and design with control envelopes

The next proposition sets the framework for verification and design of safe controllers using control envelopes.

Proposition 1 (Safety refinement)**.** *Let G be a control envelope be a control envelope that, when in closed-loop with a given plant model, causes the set $N \subseteq \mathbb{R}^n$ to be invariant. Then, a control law $u = g(x)$ in closed-loop with this plant will be safe if the three conditions below hold.*

P1: $\forall p \in P.\ X_p^0 \subseteq N$

P2: $\forall x \in N.\ g(x) \in G(x)$

P3: $N \subseteq X_{safe}$

Proof. Choose any $p \in P$, $s \in S$. To prove that N is an invariant for the system, assume for a contradiction that there is a behavior $x(t)$ that is not contained in N. Then there is a finite $k \geq 0$ such that $x(k) \notin N$.

Let (t_j) be the sampling sequence associated to the behavior $x(t)$. The intervals $[t_j, t_{j+1}]$ for $j = 0, 1, 2, \ldots$ cover the semiaxis $t \geq 0$. We first show that for any j, if $x(t_j) \in N$, then for all $t \in [t_j, t_{j+1}]$, $x(t) \in N$. Over the interval $[t_j, t_{j+1}]$, $x(t) = \xi_j(t - t_j)$, such that $\dot{\xi}_j(t) = f_p(\xi_j(t), g(x(t_j^-)))$ and $\xi_j(t_j) = x(t_j^-)$ and $\xi_j(t) \in X_p^0$ for all $t \in [t_j, t_{j+1}]$. We have that $g(x(t_j^-)) \in G(x(t_j^-))$, so it follows that $\xi_j(t) \in N$ over $[t_j, t_{j+1}]$ by the definition of behaviors of the nondeterministically controlled system.

By induction on j, k is not in any of the $[t_j, t_{j+1}]$ for any $t_j \leq k$. For the base case $j = 0$, we have from the assumptions of the theorem of the theorem that $x(0) \in N$, so it follows from our previous reasoning that $x \in N$ for all $t \in [t_0, t_1]$, so $k \notin [t_0, t_1]$. Now assume that $x(t) \in N$

for all $t \in [t_{j-1}, t_j]$. Then in particular $x(t_j) \in N$, so it follows by the argument above that $x(t) \in N$ for all $t \in [t_j, t_{j+1}]$. So $k \notin [t_j, t_{j+1}]$ for any $t_j \leq k$. It follows that if k exists, it must be contained in $[t_j, t_{j+1}]$ such that $t_j > k$, but that is a contradiction and the proposition follows. □

Formula $P1$ requires that the invariant N contain the initial set X_p^0, formula $P2$ requires that the control law be a refinement of the control envelope over the invariant, and formula $P3$ requires that the invariant be contained in the safe set X_{safe} . If these three conditions hold, the controller is proven safe.

> Formula $P2$ of Proposition 1 can be thought of as a special case of the box-generalization rule of d$\mathcal{L}$ ([87], rule (G1)). By formula $P2$ the control law as a logical formula relating u to the state x implies the control envelope as in $P2$. Then any property that holds after the hybrid program that chooses an arbitrary control from the control envelope also holds after the hybrid program that assigns a control input according to the control law. In this sense, the control envelope *generalizes* the control law.

For the synthesis problem, if $\{g_c\}_{c \in C}$ is a parametrized class of control laws, the first two logical formulas above are used in conjunction with $\forall x.\ x \in N \rightarrow g_c(x) \in G_e(x)$, where the controller parameter vector c is left unspecified and unquantified. The logical formula that results from quantifier elimination will contain constraints on the control parameters only. Then, parameter values can be chosen that satisfy these constraints according to some optimality criterion. In our examples, we partially specify some control parameters—such as gains—according to a traditional design scheme (LQR), and solve for constraints on the remaining parameters—typically setpoints and switching thresholds.

For the parametric case, given a parametric class of envelope–invariant pairs $\{G_e, N_e\}_{e \in E}$, safety checking reduces to finding an $e \in E$ such that

$$(\forall p \in P.\ X_p^0 \subseteq N_e) \ \wedge \ (\forall x \in N_{\cdot}\ g(x) \in G_e(x)) \ \wedge \ (N_e \subseteq X_{\text{safe}}) \tag{3.26}$$

In 4.1 we will discuss how to handle the parametric refinement case.

Example 6. In the non-parametric version of our running example, the invariant $d \leq 0$ covers the initial set, so the first condition of Proposition 1 is met.

The second condition holds because the control law refines the control envelope, which we can check by substituting the expression for the control law into the characteristic of the control envelope over the invariant $d \leq 0$;

$$(1.1)(-d)(0.1) + d = 0.89d \leq 0. \tag{3.27}$$

The invariant is the same as the safe set, so the third condition is met. □

3.6 Summary

In this chapter, we introduced control envelopes, and described how to verify that a control envelope renders a set invariant or attractant. Control envelopes provide an abstraction on the input-output relation of a controller such that a given safety or reachability property is satisfied. Control envelopes can be used, for example, as part of a control development workflow in which several candidate controllers are proposed and then checked for correctness. As we will see, checking safety and reachability with the help of control envelopes can be automated, so these checks can be incorporated into a design process in the same way that automatic testing is performed on incremental builds of software systems.

The basic mechanism that allows us to perform controller verification can also be used to handle synthesis problems directly by searching for appropriate parameter values in a parametrized template. This idea will be developed further in the chapter on controller synthesis.

Chapter 4

Controller Verification by Refinement Checking

We have described control envelopes, set-valued functions of the system state that delimit choices of control input that are guaranteed to be safe. We have discussed how the need for flexibility in the face of control envelopes that are overly restrictive leads naturally to the idea of refinement by parts to verify the controller with different control envelopes over different regions of state space. We have also discussed how control envelopes can be verified to control the system state through various means; using barrier and Lyapunov function arguments in the case of continuous systems, as well as encoding the problem as a hybrid program, so that theorem prover KeYmaera can be used. KeYmaera is especially well suited to the case of parametric control envelopes, since its logical framework naturally supports reasoning with symbolic parameters.

In this chapter, we consider the problem of using a control envelope to verify a candidate controller. We call this procedure refinement checking, since it reduces to checking that the con-

troller is a refinement of the control envelope. This reflects, for example, the scenario in which a control envelope has been produced by a verification team, and a separate team is tasked with the design of the controller. This is analogous to the way that unit tests are used to automatically check the integrity of software builds—but with the added benefit that our control envelopes provide formal guarantees, whereas unit tests do not. Since control envelopes are not tied to a specific controller implementation, they can be reused throughout the development lifecycle to verify different designs.

Another important scenario for this type of check is compliance with regulatory specifications and industrial standards. If industrial and regulatory specifications are given as control envelopes, compliance teams within each company can formally verify that their designs comply with the required specifications.

We first describe the parameter-free case, in which a controller is verified by checking that it refines a control envelope over its associated invariant. For small problems, a quantifier elimination suffices. For larger problems we recommend using a fast SMT solver such as dReal. We then move on to the parametric case, in which a parametric class of control envelopes is given with their associated invariants. The task in this case is to search over the possible parametrizations of the control envelope, and select one such that the controller refines the control envelope over its associated invariant. Finally, we conclude with a summary of our findings.

4.1 Refinement checking with a single envelope–invariant pair

Let a control envelope $G : X \to 2^U$, an associated invariant $N \subseteq X$, and a control law $g : X \to U$ be given. The safety verification problem is reduced to checking the validity of the first-order logical formulas

1. $\forall p \in P.\ X_p^0 \subseteq N$, and
2. $\forall x \in N\ .\ g(x) \in G(x)$,
3. $N \subseteq X_{\text{safe}}$.

To check these formulas, we have two options.

1. We can use quantifier elimination. If the formulas reduce to true, the control law is safe. The drawback of this approach, as discussed in 2.2.1, are the doubly exponential complexity of quantifier elimination, and the limitation that all arithmetic expressions must be polynomials.
2. Another possibility is to use a fast SMT solver for nonlinear arithmetic to check the satisfiability of the formulas
   (a) $p \in P\ \wedge\ x \in X_p^0\ \wedge\ x \notin N$,
   (b) $x \in N\ \wedge\ g(x) \notin G(x)$, and
   (c) $x \in N\ \wedge\ x \notin X_{\text{safe}}$

   If all three formulas are unsatisfiable, then the envelope–invariant pair vouches for the safety of the proposed control law. The strength of this approach is that in many cases it may be much faster than quantifier elimination. This is because SMT solvers often incorporate heuristics to address specific kinds of arithmetic. If one chooses an SMT solver

with heuristics that match the arithmetic in the logical expressions, excellent performance can be achieved.

The drawback is that the speedup gained by heuristics often comes at the cost of completeness guarantees, and may fail to validate a refinement query. This introduces an additional layer of conservativeness, since the control envelope itself may be conservative by excluding controls that are safe.

Example 7. We can automatically check that the control law of (3.2) is safe as follows. We use the control envelope of formula (3.3) and the invariant of formula (3.4) to construct the following logical formulas.

1. $\forall p.\ (0.9 \leq p \leq 1.1) \rightarrow \forall d.\ (-21 \leq d \leq -19) \rightarrow d \leq 0.$
2. $\forall d.\ (d \leq 0)\ \wedge\ (u = -d) \rightarrow (0.11u + d \leq 0)$
3. $\forall d.\ (d \leq 0) \rightarrow (d \leq 0)$

We use the Mathematica `Reduce` command to apply quantifier elimination to the above formulas, and they all reduce to true. □

Example 8. Now we consider a variation of the controller, one that makes the car move too fast for safety.

$$u = -10d \tag{4.1}$$

Since we have only changed the control law, only the second formula needs to be checked again. Using Mathematica's `Reduce` command to apply quantifier elimination yields falsehood. We can use the Mathematica `FindInstance` command to find a state at which the controller does

not refine the control envelope, and we obtain

$$d = -10. \tag{4.2}$$

At this state, $u = 100$ and the control envelope is

$$G(-10) = \{u \mid 0.11u - 10 \leq 0\}. \tag{4.3}$$

By substituting $u = 100$ into the characteristic of the control envelope, we see that $11 - 10 \not\leq 0$, so the control is not contained in the envelope at this point. □

4.2 Refinement checking with parametric envelope–invariant pairs

If we have verified that a parametrized class of envelope–invariant pairs $\{G_e, N_e\}_{e \in E}$ is safe, and a control law $g : X \to U$ is given, the safety verification problem is reduced to searching for a value of the envelope parameter such that the refinement condition holds. This is equivalent to checking validity of the logical formula

$$\exists e \in E.\, \forall x \in N_e.\, g(x) \in G_e(x). \tag{4.4}$$

For all but the simplest instances, quantifier elimination will be too slow to be useful, as discussed in 2.2.1. Instead, we describe a methodology inspired by simulation-driven search of Lyapunov and Barrier functions that we have discussed in the preliminaries. [1] The goal is to find a value of the envelope e such that the refinement conditions hold.

[1] This methodology is based on a personal conversation with Sicun Gao.

The goal is then to find $e \in E$ such that

$$\forall x \in N_e.\ g(x) \in G_e(x). \tag{4.5}$$

To start the procedure, we use instance-finding to choose an arbitrary value of $e \in E$. Then we check whether e makes formula (4.5) valid. If so, we simply need to double-check that $e \in E$ (recall that instance-finding is unreliable), and we are done. Otherwise, we choose sample values for x, as $\{x_1, \ldots, x_k\} \subseteq \mathbb{R}^n$. Then we use these values to generate constraints on values of e of interest. In general these constraints will be in nonlinear real arithmetic, but in specific cases they will be linear constraints. In the nonlinear case, we need to search for a candidate e^* using an instance finding query. However, if the set of envelope parameters of interest can be expressed by linear equalities and inequalities, then the search for appropriate values of E can be performed more efficiently by linear programming. The final step is to attempt to validate the envelope parameter vector e^*, and if it cannot be validated, add the resulting counterexample to the set of values of x and re-start the procedure. We first present the general case, and then the case that can be searched with the help of linear programming.

This is akin to finding parameter values for a Lyapunov function template. Indeed, the parametric envelope–invariant pair can be thought of as a template for non-parametric envelope–invariant pair.

A Lyapunov function must decrease along all system trajectories. In the work on simulation-driven search of Lyapunov functions, the idea is to construct several short sample trajectories. Any Lyapunov function must decrease along these trajectories. As a result, we can use these traces to make an educated guess for possible values of the parameters. Then the guess can be checked with a logic solver. If a counterexample is found in the form of a state at which the

Lyapunov function does not have a negative Lie derivative, we can add a simulation trace that includes this state and repeat the process.

Similarly, in the parametric refinement case, we are choosing a sampling of states, similar to choosing system traces. We use these states to make an educated guess about appropriate values of the envelope–invariant parameters, and check with a logic solver. If we find a state at which refinement does not hold, we add it to the set of samples and make a new guess.

4.2.1 Parametric refinement checking, general case

Let a parameter $\delta > 0$ be given.

1. Choose any value of $e^* \in E$. Check if (4.5) is valid when $e = e^*$. If so, ensure that $e^* \in E$ and terminate. If $e^* \notin E$, try a different value of e^*. If (4.5) is not valid when $e = e^*$, continue.

2. Choose a set of initial sample values for the variable x, $\{x_1, \ldots, x_k\} \subset \mathbb{R}^n$, separated by a distance of at least δ that are counterexamples of (4.5). We use these sample values to generate the constraint

$$(x_1 \in N_e \rightarrow g(x_1) \in G_e(x_1)) \ \wedge \ \ldots \ \wedge \ (x_k \in N_e \rightarrow g(x_k) \in G_e(x_k)) \tag{4.6}$$

   which is a constraint on e that must hold for any e that satisfies the formula (4.5).

3. The next step is to use a logic solver to find a value of e that satisfies the constraint (4.6). If no such value exists then the refinement formula (4.4) is false, and the control envelope cannot verify this controller. Otherwise, let $e = e^*$ be the value that satisfies constraint (4.6). We double-check that $e^* \in E$ by substituting e^* into the characteristic of E, and if it

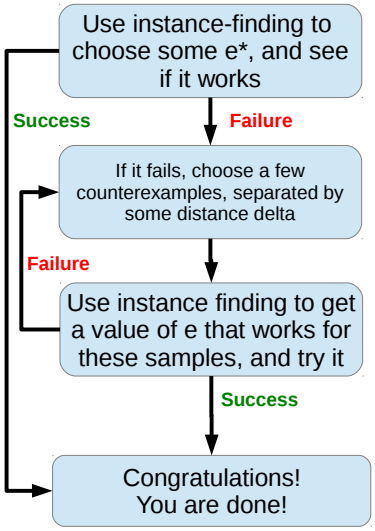


Figure 4.1: Procedure for parametric refinement checking

is not successful, we choose a different value of e. This parameter value is our candidate envelope parameter for verification, and the next step is to validate.

4. Next we use a logic solver to check validity of (4.5) with the new value of $e = e^*$. On the other hand, if x^* is a counterexample, then we go back to Step 1, adding the sample $x_{k+1} = x^*$.

Figure 4.1 shows a flowchart of this procedure.

This procedure is only guaranteed to terminate if the union of all of the invariants

$$N = \bigcup_{e \in E} N_e \tag{4.7}$$

is a bounded set. This is because the procedure will eventually run out of counterexamples separated by a distances δ that it can propose to continue guiding the search for e.

The procedure is not guaranteed to succeed every time that there is a value of e that provides refinement. However, if the procedure declares success, it will provide a value of e that serves as a witness for the refinement conditions. The procedure is sound, but it is incomplete, in the

sense that it may fail to verify safe control laws.

Note that the correctness of of the instance finding procedure is not critical to soundness. Instance finding is used to propose candidates of $e^* \in E$, and we always double check to make sure that they are contained in E. The procedure does not declare correct refinement unless this is checked. Instance finding is also used to find the samples x_i which generate the constraints that guide the search of e^*. Since the samples x_i are only used as a search heuristic, it is not critical that these values fully satisfy the constraints that generate them.

We will defer treatment of the simple stop sign assist example with parametric refinement until we present our tool Perseus in Chapter 6.

4.2.2 Parametric refinement checking, quasi-affine case

In this section, we consider a special case that allows us to search the envelope parameter space more efficiently using linear programming. The special case we consider is *quasi-affine* refinement checks.

Definition 14 (Quasi-affine refinement check)**.** *Let $N = \cup_{e \in E} N_e$. The refinement check*

$$\exists e \in E.\, \forall x \in N_e.\, g(x) \in G_e(x) \tag{4.8}$$

is said to be quasi-affine if for any fixed value $x \in N$, the set of values $e \in E$ that satisfies the constraint

$$\begin{aligned} &\forall p \in P.\, X_0^p \subseteq N_e \\ \wedge\ &\forall x \in N_e.\, g(x) \in N_e(x) \\ \wedge\ &N_e \subseteq X_{\textit{safe}} \end{aligned} \tag{4.9}$$

is a polyhedron. In this case, the constraint (4.9) can be written in the form

$$A_{ineq}(x)e \leq b_{ineq}(x) \ \wedge \ A_{eq}(x) = b_{eq}(x) \tag{4.10}$$

where $A_{ineq}(x)$, $A_{eq}(x)$, b_{ineq}, $b_{eq}(x)$ are two matrices and two vectors (respectively) that depend on x alone, possibly in a nonlinear way.

The new procedure is as follows.

Choose a $\delta > 0$.

1. As before, choose any value of $e^* \in E$. Check if (4.5) is valid when $e = e^*$. If so, ensure that $e^* \in E$ and terminate. If $e^* \notin E$, try a different value of e^*. If (4.5) is not valid when $e = e^*$, continue.

2. Choose a set of counterexamples to formula (4.5) when $e = e^*$, $\{x_1, \ldots, x_k\}$ separated by a distance of at least $\delta > 0$, and construct the logical formula

$$(x_1 \in N_e \rightarrow g(x_1) \in G_e(x_1)) \ \wedge \ \ldots \ \wedge \ (x_k \in N_e \rightarrow g(x_k) \in G_e(x_k)) \tag{4.11}$$

3. Let e_i be the ith component of the parameter vector e. For each i, solve the following optimization problems.

$$\begin{array}{ll} \min\limits_{e} \ \ e_i & \max\limits_{e} \ \ e_i \\ s.t. \ A_{ineq}(x)e \leq b_{ineq}(x) & s.t. \ A_{ineq}(x)e \leq b_{ineq}(x) \\ \quad A_{eq}(x)e = b_{eq} & \quad A_{eq}(x)e = b_{eq}(x) \end{array}$$

Since the refinement check is quasi-affine, all of the optimization problems are linear programs in the decision variable e. These optimization problems yield polyhedral bounds on

the value of e_i that proves the formula valid. If any of the linear programs are not feasible, we can use the SMT solver as before to double-check that no satisfying instance of e exists, and terminate appropriately.

At each iteration of this procedure, the box gets smaller and smaller, and help us converge on the desired value of e.

If the linear programs succeed, we choose a point that has each of its components within the bounds computed by the linear program, say $e = e^*$. Check that $e^* \in E$ to ensure that numerical error of the optimization procedure did not introduce unsoundness.

4. As before, we attempt to check that (4.5) when $e = e^*$ is a valid formula with the help of a logic solver. If not, and x_{k+1} is a counterexample, we add $x_{k+1} = x^*$ to the set of samples and repeat the procedure.

In this version of the procedure, we exploit the quasi-affine property of the logical formula to perform a faster exploration of the space. Searching the envelope parameter space with linear programming, which takes polynomial time, whereas dReal queries are NP-complete. In the general procedure, each iteration chooses e to deal with the sample values of x. In the second method, each iteration creates a smaller and smaller polyhedron, contracting at each iteration.

This section has described how to automatically check the refinement condition when the control envelope is parametric, drawing its parameters from a known set. The procedure is guaranteed to terminate when the set of envelope parameters is bounded. Furthermore, if the logical query satisfies a quasi-linearity property, the parameter space can be explored more efficiently by using linear programming to provide bounds on the sought parameter value, bounds which tighten at each iteration.

4.3 Summary

In this chapter, we described how control envelopes can be used to automatically verify specific controller designs. This is useful, for example, to check that multiple controller designs throughout the development lifecycle satisfy the safety requirement, or to provide formal verification that controllers produced by different companies satisfy regulatory specifications or industry standards.

We first considered the case in which a single, fully specified control envelope is used to verify a candidate controller. This simple case reduces to a quantifier elimination or SMT query.

Second, we considered the case in which a parametric collection of control envelopes is given. In this case, the task is to search for an appropriate parametrization. We described an automated search heuristic to find a parametrization of the envelope–invariant pair that satisfies the refinement conditions. Also, we described a more specialized case, in which a quasi-affinity condition is exploited to simplify the search procedure.

We hope that these techniques will be refined to improve speed and scalability, bringing automatic safety checking within the scope of industrial applicability.

Chapter 5

Refinement Checking by Parts

In this section, we present the technique of refinement checking by parts. In many cases, it may be difficult (or even mathematically impossible) to characterize the full set of control envelopes that keep a system safe. In general, control envelopes provide an underapproximation of the set of safe control values. As a result, controllers that are safe may be discarded as unsafe. Frequently, this conservativeness will conflict with performance requirements, and in some cases, it may be impossible to attain a minimal acceptable performance bound. We illustrate this issue on our running example below.

Example 9. In general, the maximal control envelope is difficult to compute, and we must settle for conservative underapproximations. This means that there will be safe controllers that cannot be verified by the mechanism we have described. As an example, suppose the following

parametric class of control envelopes is given, as in Example 5.

$$G_e(d) = \{u \mid (u = 0) \tag{5.1}$$

$$\vee\ (0 \leq u \leq e\ \wedge\ 0.11e + d \leq 0)\ \} \tag{5.2}$$

Recall that the car dynamics are such that the car will not drive backwards even if u is negative, so $u \leq 0$ produces a zero velocity.

For the purposes of this example, we will take the following larger invariant, which does not depend on the parameter e. However, we will Write it with the subscript N_e for consistency of notation, but of course $N_{e_1} = N_{e_2}$ for any two parameters e_1, e_2.

$$N_e = \{\ d \mid d \leq 0\ \} \tag{5.3}$$

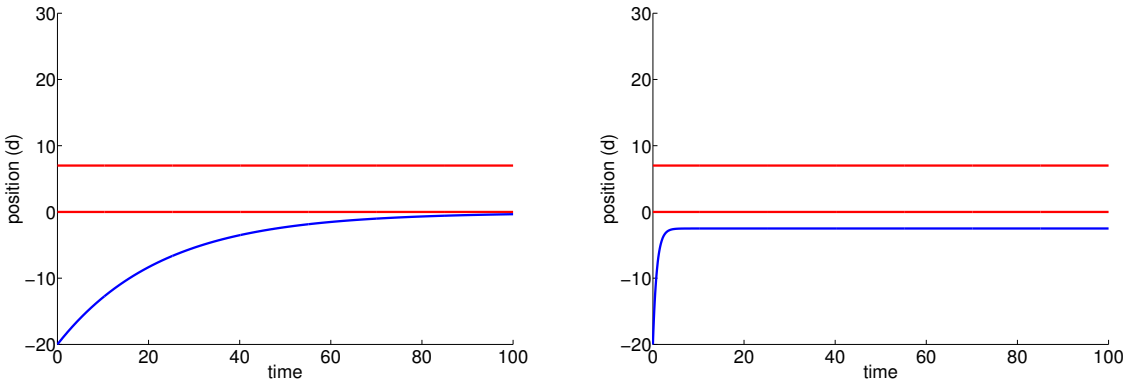
Since the characteristic of the envelope is a disjunction, the envelope is the union of two sets. The first disjunct ($u \leq 0$) allows the car to always choose to remain stopped. The second disjunct specifies that the car may choose u between 0 and e always that e is a safe choice ($(0 \leq u \leq e\ \wedge\ 0.11e + d \leq 0)$). This envelope is conservative because, for a given choice of e, it may be safe to move forward with velocity less than e, but not with e itself. In that case, the envelope requires that the car remained stopped.

The controller we have considered in equation (3.27) does not refine this control envelope, because at every point in the interval $[-0.11e, 0)$, the control law is nonzero, but the envelope demands a value of zero. This can be easily resolved by re-designing the controller to have a nonzero setpoint, d_{set}. Then the refinement check will succeed when $-0.11e = d_{set}$. However, there is an inconvenient trade-off in the value of the parameter e.

1. If the design goal is to choose the setpoint close to the intersection, the choice of e that

satisfies the refinement condition is small. This in turn restricts the velocity that the car can take, resulting in a very slow approach towards the intersection. Figure 5.1a shows a sample behavior for a controller designed with a setpoint of $0.1m$.

2. If the design goal is to control the car so that it approaches the intersection quickly, the corresponding value of e that satisfies the refinement condition is large, resulting in a set-point that is far from the intersection. Figure 5.1b shows a sample behavior for a controller that starts with a velocity of $21m/s(\approx 44mph)$ and quickly brings the car to a stop, but at a distance of about $2.4m$ from the intersection.



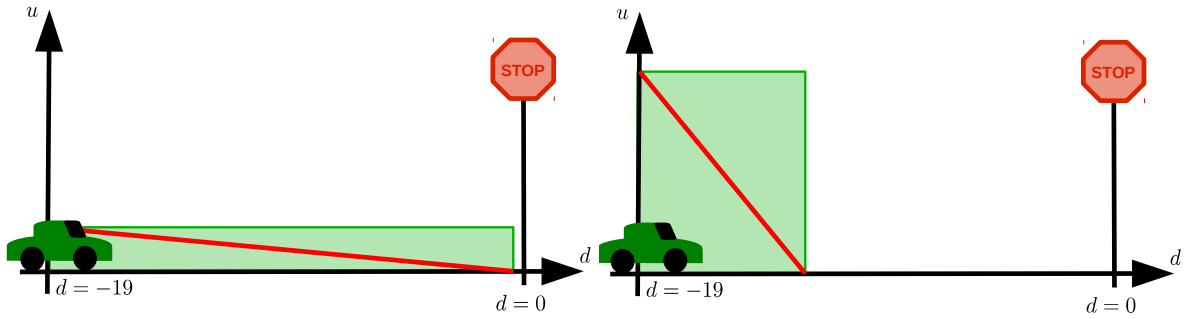
(a) Sample behavior of the slow controller

(b) Sample behavior of the far controller

Figure 5.1: Simulations of slow and far controller

We can see the corresponding illustrations in terms of the controller and control envelope in figure 5.2

We would like to have a scheme that allows us to work with an envelope that allows the car to move fast when it is far from the intersection, and to use another envelope near the intersection so that the car can get near the intersection. □



(a) Refinement illustration of the slow controller (b) Refinement illustration of the far controller

Figure 5.2: Refinement of slow and far controller

In such a scenario, one option would be to design a new control envelope. However, designing control envelopes is costly, and we prefer to do it only once in the development process. In the example, the controller needed to meet the two performance requirements of stopping near the intersection and arriving there quickly. Although controllers could be designed (and verified) for one requirement or the other, it was not possible to verify a controller that met both. What we would like is a method to combine more than one parametrization in a sound way. In this chapter, we describe the principle of refinement checking by parts and discuss an automatic procedure for it.

5.1 Refinement checking by parts

Prop. 1 may be overly restrictive if the control envelope is too conservative, since at a given point of the forward invariant the control envelope might not contain all of the safe control inputs. Any controller that tries to use safe control inputs that are not in the envelope cannot be verified, even though it may be safe. As we will see in the examples, it is often the case that a control envelope

that allows a larger set of control inputs may be associated to a smaller forward invariant. This smaller forward invariant may exclude a state to which the controller is required to drive the plant. Hence, there is a conflict if the larger envelope is associated to a forward invariant that is too small, and the larger forward invariant is associated to an envelope that is too large. Prop. 2 states the conditions that allow composing control envelopes to ameliorate the effects of this conservativeness.

Proposition 2 (Safety refinement checking by parts)**.** *Let $\{G_1, \ldots, G_k\}$ be a collection of control envelopes, and suppose that each control envelope, in closed-loop with the plant of interest, preserves the invariants $\{N_1, \ldots, N_k\}$, respectively. The system with control law $u = g(x)$ has invariant N, given by*

$$N = \cup_{i=1}^{k} N_i \tag{5.4}$$

and is safe if the following logical formulas are valid.

1. $\forall p \in P.\ X_p^0 \subseteq N$
2. $\forall x \in N.\ \bigvee_{i=1}^{k} (x \in N_i \ \wedge \ g(x) \in G_i(x))$
3. $N \subseteq X_{safe}$

Proof. Fix any $p \in P$ and $s \in S$. To prove that N is an invariant for the deterministically controlled system, assume for a contradiction that there is a behavior such that for some finite $k \geq 0$, $x(k) \notin N$.

Let (t_j) be the sampling sequence associated to $x(t)$. Note that the family of intervals $[t_j, t_{j+1}]$ covers the semiaxis $t \geq 0$. Note that for any j, if $x(t_j) \in N$, then $x(t) \in N$ for all $t \in [t_j, t_{j+1}]$. To show this, let $i \in I(x(t_j))$, so that $x(t_j) \in N_i$. Over the interval $[t_j, t_{j+1}]$, $x(t) = \xi_j(t - t_j)$, such that $\dot{\xi}_j = f_p(\xi_j(t), g(x(t_j^-)))$ and $\xi_j(t_j) = x(t_j^-)$ and $\xi_j(t) \in X_{\text{safe}}$ for all $t \in [t_j, t_{j+1}]$.

Since $g(x) \in G_i(x)$, it follows that $\xi_j(t) \in N_i$ over $[t_j, t_{j+1}]$ by the definition of behaviors of the nondeterministically controlled system.

By induction on j, k is not in any of the intervals $[t_j, t_{j+1}]$ for any $t_j \leq k$. For the base case $j = 0$, we have from $X_p^0 \subseteq N$ that $x(0) \in N$, so it follows from our previous argument that $x(t) \in N$ for all $t \in [t_0, t_1]$. Now assume for the induction hypothesis that $x(t) \in N$ for all $t \in [t_{j-1}, t_j]$. Then, in particular $x(t_j) \in N$, and it follows from the reasoning above that $x(t) \in N$ for all $t \in [t_j, t_{j+1}]$. Then if k exists, it must be contained in some $[t_j, t_{j+1}]$ such that $k > t_j$, but that is a contradiction and the proposition follows. □

An alternate way to think of Proposition 2 is as follows. Suppose one wishes to verify safety of the system controlled by the control law. This can be accomplished by applying a cut corresponding to each part that one is using. In a sense, one is providing cuts from envelope–invariant pairs until the full action of the controller and the corresponding reaction of the plant is accounted for and noted to be safe. A similar idea is at work in [93], which applies differential cuts incrementally until safety can be proven for a looping hybrid program.

Example 10. A controller that both moves the car quickly and leaves it near the intersection is given by

$$u = -(d + 0.1). \tag{5.5}$$

Next we use refinement by parts to verify this controller. This can be attained by using the four parametrizations $e_1 = 23$, $e_2 = 15$, $e_3 = 5$, $e_4 = 0.9$. These effectively provide four rectangles that provide cover for the controller. A sample trajectory for this controller is shown in Figure 5.3.

Figure 5.4 shows the relationship of the control envelopes to the control law.

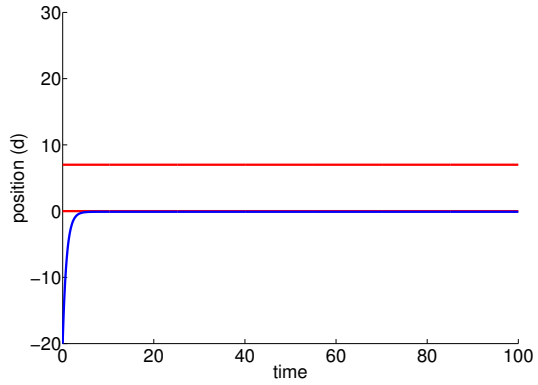


Figure 5.3: Controller verified using refinement by parts

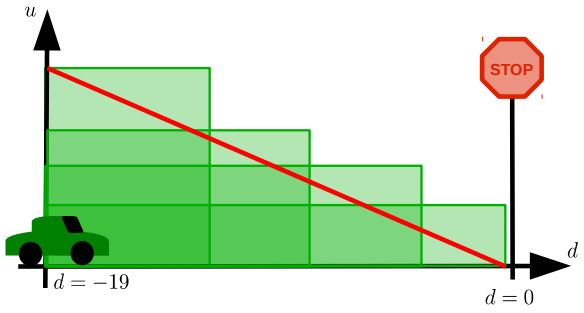


Figure 5.4: Control law and multiple control envelopes

□

5.2 Automatic refinement checking by parts

Given a parametrized class of control envelope–invariant pairs $\{G_e, N_e\}_{e \in E}$ and a control law $g : X \to U$, the safety verification problem is reduced to searching for a set of values of the envelope parameters $\eta = (e_1, \ldots, e_k)$ that satisfy the following requirements. Let

$$N_\eta = \bigcup_{e_i \in \eta} N_{e_i}, \tag{5.6}$$

and let

$$E^k = \underbrace{E \times \cdots \times E}_{k}. \tag{5.7}$$

Then, for each number of parameters that we wish to use in the refinement checking process, we can form a logical formula

$$\exists \eta \in E^k. \, \forall x \in N_\eta. \, \bigvee_{i=1}^{k} (x \in N_{e_i} \ \wedge \ g(x) \in G_{e_i}) \tag{5.8}$$

This logical formula is an exists-forall formula that can be handled as the previous case. Then, the procedure to check refinement by parts is as follows.

1. Start with $k = 1$, to attempt refinement checking with a single control envelope. Then $\eta \in E$ is a single parameter vector.

2. Construct the logical formula

$$\exists \eta \in E^k. \, \forall x \in N_\eta. \, \bigvee_{i=1}^{k} (x \in N_{e_i} \ \wedge \ g(x) \in G_{e_i}(x)), \tag{5.9}$$

   and check its validity using a logic solver. If it succeeds, terminate.

3. If the query does not succeed, increment k. If $k = 2$, then $\eta \in E \times E$, if $k = 3$, then $\eta \in E \times E \times E$, and so on.

Figure 5.5 shows a flowchart of this procedure.

The procedure can be made to terminate by placing an upper bound on the number of parts that we wish to try, and if the underlying refinement checking procedures terminate.

This provides a mechanical heuristic to search for a set of envelope–invariant pairs that satisfy the conditions of refinement by parts. It is not an algorithm, in the sense that it is not guaranteed to terminate with a refinement by parts whenever one exists, but it is sound in the sense that if it returns with a result, the result is guaranteed to be correct.

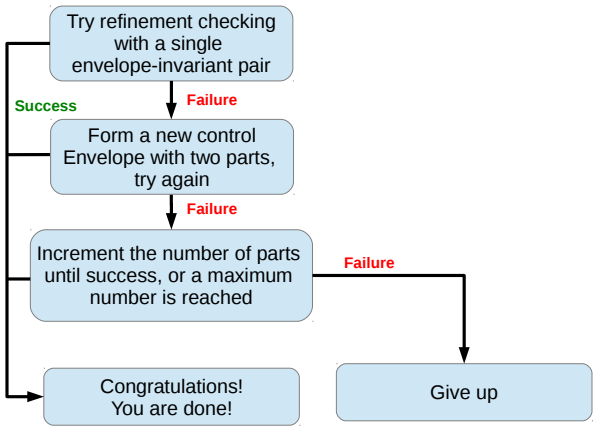


Figure 5.5: Refinement checking by parts

5.3 Conclusions

A control envelope is usually an underapproximation of the full set of safe control actions, because the full set is usually difficult and sometimes impossible to compute. As a result, controllers that are perfectly safe will be discarded by the verification process. In some cases, it may not even be possible to verify controllers that meet minimal performance criteria. This chapter described the technique of refinement by parts, in which different parametrizations of a control envelope are used in different regions of the state space. This is useful when the multiple parametrizations provide a better approximation of the true set of safe control inputs. In our running example, this can be seen as analogous to approximating a triangular region by squares that fit inside it. We also described a procedure to perform refinement checking by parts automatically.

Chapter 6

Tool support: Verification with Perseus

We have developed a tool named Perseus to automate the parametric refinement checking process. A diagram of Perseus is shown in figure 6.1. Perseus consists mainly of a user interface block, which takes commands and displays results, and a refinement verifier block.

The user interface block takes commands from the user, and dispatches requests to the refinement verifier. The refinement verifier receives a problem instance and a request for the desired operation and forms the logical formulas that need to be analyzed. Then, these formulas are dispatched to a logic solver, and the results are received and reported back to the user.

Perseus makes use of two blocks of the proteus logic library, which was also developed over the course of this research. Proteus provides a parser, interfaces to external tools, and rudimentary arithmetic and logic manipulation capabilities.

Perseus uses the parser module proteus when loading problem files, as well as the logic solver interface component, which provides a unified interface to both Mathematica and dReal. The logic solver interface was designed with extensibility in mind, and can be easily be extended

to accomodate other logic solvers in the future.

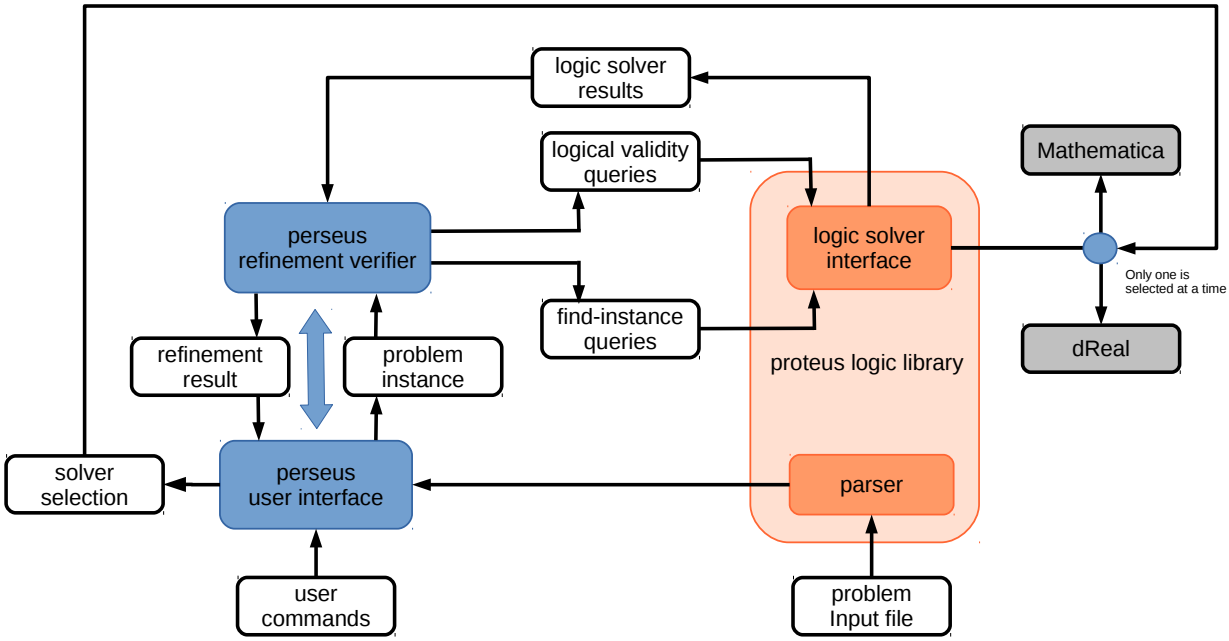


Figure 6.1: Diagram of Perseus

6.1 The Perseus user interface

Perseus uses a command-line interface. The prompt supports the following commands.

1. `load <filename>`: Loads a problem file. The input file format is discussed in section 6.2.

2. `print`: Displays the information of a problem, if one has been loaded.

3. `auto-refine`: Attempts to automatically find a parameter for the envelope–invariant pair such that the refinement conditions hold.

4. `auto-parts`: Attempts to automatically search for a set of parameters such that the refinement by parts conditions hold.

5. `propose-refine`: Allows the user to propose a valuation of the envelope–invariant parameter such that the refinement conditions hold, and then checks it.
6. `propose-parts`: Allows the user to propose a set of valuations such that the refinement by parts conditions hold, and checks.
7. `clear`: Clears working memory.
8. `version`: Prints version information.
9. `set-solver`: Allows the user to set the logical solver to use. Currently supported choices are Mathematica and dReal.
10. `help`: Displays a list of available commands.
11. `exit`: Terminates the program.

6.2 Perseus input file format

A sample input file for the slow controller of the simplified stop sign assist is shown in figure 6.2.

The `statevariables` block contains a comma-separated list of the state variables of the system. The `initialset` block contains a logical formula that characterizes the set of all possible initial states, for all possible plant parameter valuations,

$$X^0 = \bigcup_{p \in P} X_p^0. \tag{6.1}$$

The `safeset` block contains a logical formula that characterizes the set of safe states, X_{safe}. The `eiparameterset` block contains a logical formula that characterizes the set of envelope–

```
\statevariables{
        d
}

\initialset{
        -20 <= d & d <= -19
}

\safeset{
        -30 <= d & d <= 0
}

\eiparameterset{
        e >= 0 & e <= 1000
}
\envelope{
        ((0.11*e + d <= 0) & ( u <= e )) | ( u <= 0 )
}

\invariant\{
        -21 <= d & d <= 0
}

\controllaw{
        (?(d >= -0.01); u := 0)
        ++(?(d < -0.01); u := -0.04*(d + 0.1) )
}
```

Figure 6.2: Simplified stop sign assist, slow controller

invariant pairs E. The block `envelope` contains a logical formula over the state variables and the control input. At each state, the control envelope consists of the input that satisfy the logical formula. The block `invariant` contains a logical formula that characterizes the invariant set N_e. The `controllaw` block describes the control law as a purely discrete program, with only guards and assignments, in the language of d$\mathcal{L}$. This merits some further discussion.

In general, suppose we are given the following switching controller.

$$u = \begin{cases} u_1 & \text{if} \quad x \in S_1 \\ \vdots & \\ u_k & \text{if} \quad x \in S_k \end{cases} \tag{6.2}$$

In the language of d$\mathcal{L}$, and for the purposes of a Perseus input file, we will model the control law as the following discrete program.

$$?(x \in S_1); u := u_1 \tag{6.3}$$

$$\cup?(x \in S_2); u := u_2 \tag{6.4}$$

$$\vdots \tag{6.5}$$

$$\cup?(x \in S_k); u := u_k \tag{6.6}$$

$$\tag{6.7}$$

The above program nondeterministically checks for membership in the sets S_i, and executes the control assignment for the check that succeeds. The sets must be disjoint, because otherwise our control law would be a relation and not a true function. Note that the non-ascii symbol $\cup$ of d$\mathcal{L}$ is represented in the input file by the notation ++.

The discrete program in the `controllaw` block of 6.2 corresponds to the following control law.

$$u = \begin{cases} 0 & \text{if} \quad d \geq -0.01 \\ -0.04(d + 0.01) & \text{if} \quad d < -0.01 \end{cases} \tag{6.8}$$

These components specify a problem instance for refinement checking or refinement checking by parts.

6.3 The Perseus refinement verifier

The refinement verifier supports four core operations.

1. Automatic refinement checking.
2. Automatic refinement checking by parts.
3. Checking refinement with a user-specified envelope–invariant parameter.
4. Checking refinement with a set of user-specified envelope–invariant parameters.

The last two simply involve substituting parameter values that the user provides into the refinement formulas, so we will not discuss them further.

6.3.1 Parametric refinement checking with Perseus

First, Perseus constructs the logical formulas for parametric refinement checking. We show the corresponding logical formulas for our running example of the simple stop sign assist slow controller.

1. Check that the invariant is initialized, $X^0 \subseteq N_e$. In our running example, this is

$$(-20 \leq d \leq -19) \rightarrow (-21 \leq d \leq 0). \tag{6.9}$$

2. Check that the invariant is safe, $N_e \subseteq X_{\text{safe}}$. In our running example, this is

$$(-21 \leq d \leq 0) \rightarrow (-30 \leq d \leq 0). \tag{6.10}$$

3. Check that the control law refines the control envelope over the invariant, $\forall x \in N_e. \rightarrow g(x) \in G_e(x)$. This item merits further discussion.

If the control law has no switching, i.e. the control law is a simple assignment such as $u = g(x)$, where g has no switching, then the corresponding d$\mathcal{L}$ program is the concrete assignment program $u := g(x)$. Suppose our control envelope is given in the form

$$G_e(x) = \{\ u \mid \phi(x, u, e)\ \} \tag{6.11}$$

where $\phi(x, u, e)$ is a logical formula. This means that for a choice of the parameter e, the allowed control values at x are those that satisfy $\phi(x, u, e)$.

Then the refinement formula can be expressed as

$$\forall x.\ x \in N_e \rightarrow \phi(x, g(x), e). \tag{6.12}$$

In general (and in our running example), the control law will be a switching control law, of the following form.

$$u = \begin{cases} g_1(x) & \text{if} \quad x \in S_1 \\ \vdots & \\ g_k(x) & \text{if} \quad x \in S_k \end{cases} \tag{6.13}$$

In this case, the refinement formula will take the form

$$\bigvee_{i=1}^{k} (x \in N_i\ \wedge\ \phi(x, g_i(x), e). \tag{6.14}$$

In our running example, the resulting refinement formula is

$$(d \geq -0.01\ \wedge\ ((0.11e + d \leq 0) \wedge (0 \leq e)) \vee (0 \leq 0)) \tag{6.15}$$

$$\vee\ (d < -0.01\ \wedge\ ((0.11e + d \leq 0) \tag{6.16}$$

$$\wedge\ (-0.04(d + 0.1) \leq e)) \vee (-0.04(d + 0.1) \leq 0)). \tag{6.17}$$

This formula is obtained by forming a clause for each of the two regions of the controller. Equation (6.15) corresponds to the region $d \geq -0.01$. The expression for this region is logically and-ed with the expression for the control envelope with the corresponding control law substituted for u. In this case, $u = 0$. The second clause on lines (6.16 and 6.17) correspond to the region $d < -0.01$, and this condition is logically and-ed with the expression of the control envelope in which $u = -0.04(d+0.1)$ has been substituted, since that is the control action over this region.

Now that the required logical formulas have been constructed, Perseus carries out the procedure described in Chapter 4.

1. First, Perseus selects a value of e that satisfies

$$e \geq 0 \ \wedge \ e \leq 1000. \tag{6.18}$$

and obtains

$$\{e \rightarrow 2.9802322387695316E-5\} \tag{6.19}$$

2. Next, Perseus constructs the logical formulas for invariant initialization, invariant safety, and refinement checking. Perseus substitutes the value of e and checks their validity. The refinement condition fails, so Perseus extracts a counterexample,

$$\{d \rightarrow -20.999960137834655\} \tag{6.20}$$

3. Perseus takes the logical formula from lines (6.15—6.17), and forms the logical and with

the characterization of E, yielding the following constraint.

$$e \geq 0 \ \wedge \ e \leq 1000 \tag{6.21}$$

$$(d \geq -0.01 \ \wedge \ ((0.11e + (-20.999960137834655) \leq 0) \wedge (0 \leq e)) \vee (0 \leq 0)) \tag{6.22}$$

$$\vee \ ((-20.999960137834655) < -0.01 \tag{6.23}$$

$$\wedge \ ((0.11e + (-20.999960137834655) \leq 0) \tag{6.24}$$

$$\wedge \ (-0.04((-20.999960137834655) + 0.1) \leq e)) \tag{6.25}$$

$$\vee (-0.04((-20.999960137834655) + 0.1) \leq 0)). \tag{6.26}$$

4. Perseus searches for a value of e that satisfies the formula on lines (6.21—6.26), and obtains

$$\{e \to 0.8360437223836272\}. \tag{6.27}$$

5. Perseus substitutes the value from (6.27) into the formulas from lines (6.15)—(6.17) and checks their validity. Since the invariant does not depend on e, the checks of invariant initialization and safety do not change and we do not repeat them here. The refinement query is

$$(d \geq -0.01 \ \wedge \ ((0.11(0.8360437223836272) + d \leq 0) \tag{6.28}$$

$$\wedge (0 \leq (0.8360437223836272))) \vee (0 \leq 0)) \tag{6.29}$$

$$\vee \ (d < -0.01 \ \wedge \ ((0.11(0.8360437223836272) + d \leq 0) \tag{6.30}$$

$$\wedge \ (-0.04(d + 0.1) \leq (0.8360437223836272))) \tag{6.31}$$

$$\vee (-0.04(d + 0.1) \leq 0)). \tag{6.32}$$

6. The formula from lines (6.28—6.32) is valid, so Perseus terminates and reports success.

6.3.2 Refinement checking by parts with Perseus

Perseus also supports refinement by parts. We omit the detailed formula constructions, since they are similar to those of the previous example, and because thirteen iterations are required, which is too unwieldy for an in-text example.

The Perseus input file is shown in 6.3. Perseus is able to verify refinement with three parts.

1. $\{e \rightarrow 2.1990297860024226\}$
2. $\{e \rightarrow 20.90000092429955\}$
3. $\{e \rightarrow 0.14191314306946679\}$

It can be seen that the second parameter is meant to be valid when the car is far from the intersection, and the third parameter when the car is very close. The first parameter in the list is used to provide refinement at an intermediate position.

6.4 Logic solver interfaces

Two logic solvers are currently interfaced into Perseus; dReal and Mathematica. Perseus only assumes that the code wrappers for this interface implement the following two methods.

1. `checkValidity`: Takes a logical formula and returns whether it is valid or not.
2. `findInstance`: Takes a logical formula and returns a valuation of variables that satisfy the formula.

We clarify that the dReal and Mathematica interfaces are *not* used at the same time, during the same verification attempt. However, if verification fails with one solver, the user may swap the solver using the `set-solver` command and try again.

```
\statevariables{
        d
}

\initialset{
        -20 <= d & d <= -19
}

\safeset{
        -30 <= d & d <= 0
}

\eiparameterset{
        (e >= 0 ) & ( e <= 1000 )
}

\envelope{
        ((0.11*e + d <= 0) & ( u <= e )) | ( u <= 0 )
}

\invariant{
        -21<= d & d <= 0
}

\controllaw{
        u := -1*(d + 0.1)
}
```

Figure 6.3: Simple SSA controller that requires refinement by parts

6.4.1 The proteus dRealKit interface

Validity checking: The dReal interface can handle logical formulas with arbitrary linear and nonlinear arithmetic, but only if all of the variables that occur in the logical formula are universally quantified. If an existential quantifier occurs in the formula, the module throws an exception.

The dReal interface checks validity of a given logical formula by constructing its negation, and asking dReal to find a valuation that satisfies the logical formula. To increase the speed at

which dReal handles the query, we do not simply negate the formula and give it to dReal. Instead, we negate the formula and distribute the negation throughout the logical operators. Then we split the resulting formula into conjuncts, and print each conjunct as a separate constraint.

Figure 6.4 shows a sample validity query as generated by proteus. This is the query that was generated when Perseus requested a check of the validity of the formula in lines (6.28—6.32).

Finding an instance: To find an instance with dReal, proteus simply prints the desired constraints into a dReal input file, and invokes a dReal process on the file. Figure 6.5 shows a sample query to find an instance. This is the query that was generated when Perseus requested a value of e to satisfy (6.18)

If a satisfying instance exists, dReal is guaranteed to find it. However, an instance that dReal returns is not guaranteed to satisfy the given constraints—instead, the valuation returned by dReal δ-satisfies the given constraints. Throughout our procedure for refinement checking, this is usually not a problem. This is because any δ-counterexample can safely be treated as a true counterexample. As a result, our refinement conditions will be required to hold *robustly*.

However, there is one occasion in which we request an instance, and δ-satisfiability poses a problem: ensuring that the final value of the parameter e that witnesses validity of the refinement conditions is truly contained in the set E. Fortunately, this can easily be checked by substituting the final value of e into the characteristic of E, and checking that it evaluates to `True`. This is not an issue for intermediate values of e; since they are merely stepping stones to a final result and do not purport to guarantee anything.

6.4.2 The proteus MathematicaKit interface

Validity checking: The Mathematica interface checks logical validity by using quantifier elimination, implemented as the `Reduce` command. This command will succeed if the logical expressions contain purely polynomial arithmetic (e.g., no sines, cosines, exponentials, or logarithms), and when all variables that occur in the formula are quantified. The quantifiers may be existential or universal, but checking refinement with Mathematica is generally slower than with dReal.

The interface works by writing the desired logical formula into a Mathematica script, invoking a Mathematica process, and checking whether the result was `True` or `False`. Any other result signals an error, and an exception is thrown.

Figure 6.6 shows a sample query generated by proteus when Perseus wants to check the validity of formula (6.10), which establishes that the invariant is safe.

Finding an instance: This interface searches for a valuation that satisfies a given logical formula by using the Mathematica `FindInstance`. Unlike dReal, `FindInstance` is not guaranteed to find a satisfying instance whenever it exists. However, any valuation that is computed by `FindInstance` is guaranteed to satisfy the given logical formula exactly. As before, we write a Mathematica script query automatically and read back its result.

6.5 Summary

In this chapter, we described the tool Perseus, which can automatically check whether a given control law refines a parametric control envelope over its corresponding invariant. Perseus can also perform refinement by parts.

We stepped through Perseus's reasoning by working out an example for the simplified stop sign assist scenario. We described relevant pieces of the proteus logic library—namely, use of its parser and its interfaces to logic solvers.

We described the dReal interface and the Mathematica interface, as well as the strengths and weaknesses of these tools. In many ways, these two solvers complement each other.

1. dReal is much faster, with a time complexity in NP-complete, whereas Mathematica's quantifier elimination is doubly exponential.
2. dReal has support for logical formulas with any kind of linear or nonlinear arithmetic, whereas Mathematica (and quantifier elimination more generally) can only handle logical formulas with polynomials.
3. Quantifier elimination can check validity of formulas with arbitrary quantifier alternations, i.e., the logical formula of interest can contain existential and universal quantifiers, in any order. dReal can only be used to check validity of formulas in which all variables are universally quantified.
4. Mathematica's built-in heuristics for finding a satisfying instance to a set of logical formulas, as implemented by the `FindInstance` command, may not successfully find a satisfying valuation, whereas dReal produces a valuation that only δ-satisfies the desired constraints.
5. If Mathematica's `FindInstance` command does find a satisfying valuation for a logical formula, it is guaranteed to satisfy the formula exactly. On the other hand, dReal can only provide valuations that δ-satisfy given constraints. This means that we must check the final

envelope parameter e at the end of our refinement checking computations, and ensure that it does in fact belong fully in the set E.

With the help of Perseus, once a parametric envelope–invariant pair has been verified, a control engineer can—fully automatically—check that a controller design meets its safety requirements in a fully automatic way. As a result, safety verification requires a good deal of initial effort, but becomes nearly effortless afterwards.

```
;; Automatically generated by Proteus on Wed Apr 22 22:38:16 EDT 2015

;; Check (conjunctive) validity of:
;;
;; Formula 1:
;; Implies[ ( ( ( 0 - 21 ) <= d ) && ( d <= 0 ) ), ( ( False || ( ( d >=
( 0 - 0.01 ) ) && ( ( ( ( ( 0.11 * 0.8360437223836272 ) + d ) <= 0 ) &&
( 0 <= 0.8360437223836272 ) ) || ( 0 <= 0 ) ) ) ) || ( ( d < ( 0 - 0.01
) ) && ( ( ( ( ( 0.11 * 0.8360437223836272 ) + d ) <= 0 ) && ( ( 0 - (
0.04 * ( d + 0.1 ) ) ) <= 0.8360437223836272 ) ) || ( ( 0 - ( 0.04 * ( d
+ 0.1 ) ) ) <= 0 ) ) ) ) ]

(set-logic QF_NRA)


;; Variable declarations
(declare-fun d () Real )

;; Formula is (( ( 0 - 21 ) <= d ))
(assert (<= (- 0 21 ) d) )

;; Formula is (( d <= 0 ))
(assert (<= d 0) )

;; Formula is (True)
(assert true )

;; Formula is (( ( d < ( 0 - 0.01 ) ) || ( ( ( ( ( 0.11 *
0.8360437223836272 ) + d ) > 0 ) || ( 0 > 0.8360437223836272 ) ) && ( 0
> 0 ) ) ))
(assert (or (< d (- 0 0.01 )) (and (or (> (+ (* 0.11 0.8360437223836272
) d ) 0) (> 0 0.8360437223836272) ) (> 0 0) ) ) )

;; Formula is (( ( d >= ( 0 - 0.01 ) ) || ( ( ( ( ( 0.11 *
0.8360437223836272 ) + d ) > 0 ) || ( ( 0 - ( 0.04 * ( d + 0.1 ) ) ) >
0.8360437223836272 ) ) && ( ( 0 - ( 0.04 * ( d + 0.1 ) ) ) > 0 ) ) ))
(assert (or (>= d (- 0 0.01 )) (and (or (> (+ (* 0.11 0.8360437223836272
) d ) 0) (> (- 0 (* 0.04 (+ d 0.1 ) ) ) 0.8360437223836272) ) (> (- 0 (*
0.04 (+ d 0.1 ) ) ) 0) ) ) )

(check-sat)
(exit)
```

Figure 6.4: A sample validity check in dReal, as generated by proteus to check formulas on lines (6.28—6.32
)

```
;; Automatically generated by Proteus on Wed Apr 22 22:38:15 EDT 2015

;; Find a valuation of variables that satisfies the formulas:
;;

;; Formula 1:
;; ( ( e >= 0 ) && ( e <= 1000 ) )

(set-logic QF_NRA)


;; Variable declarations
(declare-fun e () Real )

;; Formula is (( ( e >= 0 ) && ( e <= 1000 ) ))
(assert (and (>= e 0) (<= e 1000) ) )

(check-sat)
(exit)
```

Figure 6.5: A sample instance-finding query as generated by proteus, for formula (6.18)

```
(* Automatically generated by Proteus on Wed Apr 22 22:35:04 EDT 2015 *)

(* Check the (conjunctive) validity of: *)
(* Formula 1: *)
(* Implies[ ( ( ( 0 - 21 ) <= d ) && ( d <= 0 ) ), ( ( ( 0 - 30 ) <= d )
&& ( d <= 0 ) ) ] *)

reducedFormula = Quiet[ Reduce[

        ForAll[ d, Implies[ ( ( ( 0 - 21 ) <= d ) && ( d <= 0 ) ), ( ( (
0 - 30 ) <= d ) && ( d <= 0 ) ) ] ]

, {}, Reals ] ];
Print[ reducedFormula ];
```

Figure 6.6: Sample validity check with Mathematica, for formula (6.6)

Chapter 7

Control Envelopes Case Studies

7.1 Introduction

Chapter 3 introduced the control envelope framework. A control envelope provides a state-dependent set of control inputs that is guaranteed to keep the system state within a proposed invariant, and any controller that acts in accordance with the control envelope will also preserve the corresponding invariant. If this invariant contains the initial set of the system and is in turn contained in the desired safe set, safety of the closed-loop system follows..

Chapter 4 described how to use control envelopes for verification. In the simplest case, the envelope–invariant pair contain no parameters, and refinement consists of simply checking validity of a formula of first order logic, which can be done either by quantifier elimination of use or an SMT solver. We also described a procedure to search over parameters of a parametric envelope–invariant pair to find one that certifies safety of the proposed controller. Chapter 5 described the procedure of *refinement by parts* to reduce conservativeness of the verification

procedure by allowing the use of multiple parametrizations of the envelope–invariant pair.

In this chapter, we consider a case study in which we demonstrate how control envelopes can be combined with traditional design techniques. As we will see, the use of control envelopes provides to key advantages over competing methods.

1. In the examples, we are able to use a control design technique, linear quadratic regulation, that does not in itself provide safety guarantees to solve a problem with hard safety specifications. This is possible because the control envelope provides the supplementary reasoning to ensure safety. In general, control envelopes provide a way to *retrofit conventional controller design techniques to handle safety constraints.*
2. The control envelope is reusable, and does not need to be recomputed even as we overhaul the controller to better meet the performance requirements.

7.2 Related work

The earliest driver assistance technology, anti-lock braking, works by ensuring that when the driver attempts to brake, the car always applies a braking strategy that results in the shortest stopping distance. This is accomplished by ensuring that the tires never skid on the pavement. Studies of the effectiveness of this technology in preventing or mitigating car accidents have found mixed results, generally finding that anti-lock braking provides modest benefit [58] [21] [19]. More recently, however, the technology of vehicle stability control, has proven to significantly reduce accident rates, mitigate damage, and reduce accident casualties [3][66]. Vehicle stability control works by applying different amounts of braking to each wheel, to prevent loss

of control when turning into a curve.

Increased automation of automobiles has been widely recognized as an important technology, which would greatly increase vehicle safety. To attack this research problem, DARPA has held a series of competitions to encourage the development of increasingly autonomous vehicles [20].

A great deal of research effort has been devoted to the varied challenges that arise from the development of autonomous vehicles, such as sensors and algorithms to detect the physical environment, navigation and trajectory planning techniques, and methods to obtain formal assurances that vehicles will respect some safety property.

One important aspect of autonomous driving systems is the availability of reliable, high-quality, inexpensive sensors. Maurelli et al. describe an inexpensive laser scanner capable of producing high quality 3D images, and evaluate its performance on the DARPA Urban Challenge [74]. Levinson et al. use a camera for traffic light detection with a camera in imperfect lighting conditions [64].

Laser scanners are not the only technology being explored for vehicle localization and navigation. Holden, for example, describes an autonomous vehicle platform that relies on GPS alone. It is a low-cost platform that allows for prototyping and research purposes, but is otherwise limited to obstacle-free environments, and does not fully address all the challenges of real-world autonomous driving [54].

On the navigation and trajectory planning front, De Lima and Silva Pereira investigate safe navigation for an autonomous vehicle. They use global motion planning techniques, but with a windowing approach to allow handling of obstacles that do not appear in the map [31].

Kim et al. describe self-adaptive evolutionary strategies to automatically design controllers

to drive a car around a track. The resulting controllers are tested on an open-source car racing simulation software to compare with traditional control design schemes [63].

In the near term, cars will increase their degree of autonomy by increasing the basic maneuvers that they can handle without driver intervention. On this front, Li et al. use techniques from fuzzy logic to develop controllers for a number of common driving scenarios, such as corner control, garage parking, and parallel parking [65].

Taking a different approach to vehicle control, Goehring et al. describe an approach to semi-autonomously control a car, in which a human user, through a brain computer interface, makes high-level decisions on trajectory planning, such as path selection at intersections and forks [45]. The low-level control, to provide lane following and obstacle avoidance, are handled by an autonomous controller.

Traffic patterns are chaotic, and often the strongest verification results are of a probabilistic nature. Althoff et al. abstract traffic dynamics for autonomous vehicles as Markov chains, allowing a probabilistic characterization of future positions [5] . As a result, it is possible to compute the probability of a crash within a given time horizon with a given control strategy.

Reachability verification techniques have been successfully employed to compute bounded-time maneuvers on the fly to handle emergency scenarios. Althoff and Dolan present an algorithm to compute all the reachable states of a car using a bicycle model with uncertain perturbations [4] . This technique is used to plan and verify safety of emergency maneuvers on the fly for a car to avoid an obstacle in the road, while avoiding collisions with an oncoming vehicle.

On the symbolic verification front, Loos et al. use the KeYmaera theorem prover to formally verify safety of a cruise control system [69]. The work in [68] also uses the KeYmaera theorem

prover to examine intersection scenarios, and [70] examines the influence of network delay on the conservativeness of a formally verified technique.

Modern luxury vehicles from many manufacturers are equipped with a variety of safety-enhancing technologies, such as pedestrian detection, lane departure warning, and emergency brake activation to prevent collisions [16] [17] [35] [22] [112] [111]. Google has presented a prototype for a fully automatic car, but it is not in production at this time [109].

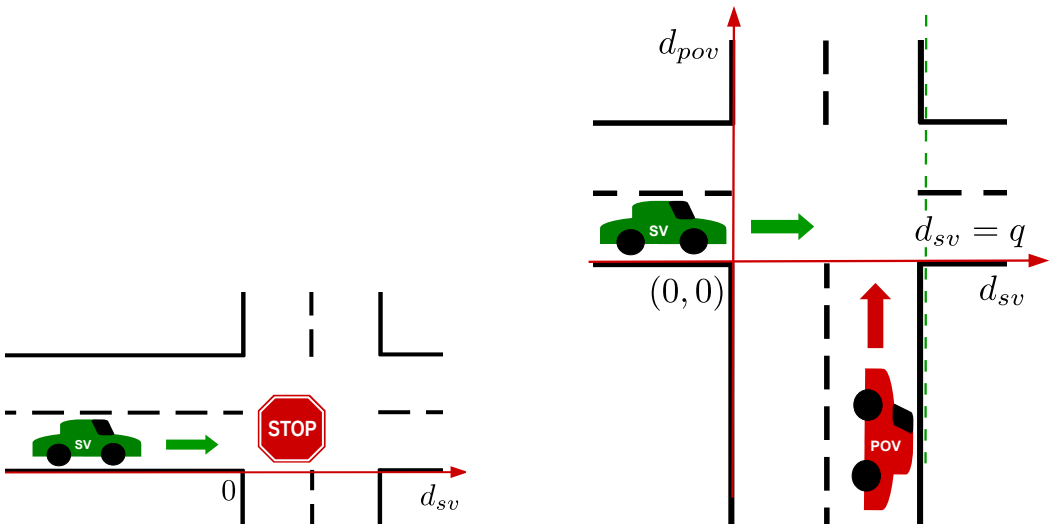
Our case study examines a driver assistance technology that allows a car to safely pull up to an intersection, and, with regard to another vehicle approaching the intersection, decide when it is safe to traverse the intersection. We use control envelopes to characterize a class of safe controllers, and use them to verify proposed controller designs.

7.2.1 Stop Sign Assist

Intersection management is an important opportunity to improve safety with automation, since these account for $97 billion in damages and 9,500 fatalities per year [24]. Cooperative Intersection Collision Avoidance Systems (CICAS) is a joint initiative between the government and industry partners to outfit intersections with sensing and communication infrastructure [81]. The goal is to build smart intersections to communicate and coordinate with intelligent vehicles to prevent accidents.

A number of different scenarios are considered under the CICAS initiative. In this case study, we consider the stop sign assist scenario[1], in which a car arrives at an intersection regulated by a stop sign. The car must come to a complete stop in front of the intersection, then decide when and how to traverse the intersection while avoiding a collision with a vehicle coming from

a perpendicular direction. This example was first presented in [10].



(a) Stop sign assist stage 1, SV approaches intersection

(b) Stop sign assist stage 2, SV must traverse intersection while avoiding POV

The controlled vehicle is called the *subject vehicle* (SV), and the other car is called the principal other vehicle (POV). We decompose the scenario into two stages. In the first stage, the subject vehicle is arriving at the intersection and needs to stop without violating the stop sign, as shown in figure 7.1a. In the second stage, the car starts stopped near the start of the intersection and detects a vehicle coming from a perpendicular direction, and needs to decide if and how to accelerate to safely traverse the intersection, as shown in 7.1b.

7.3 Stop sign assist stage 1

Stop Sign Assist, Stage 1. In the first stage, the automobile is approaching an intersection with a stop sign (Fig. 7.2). The position and velocity of the car are d_{sv} and v_{sv}, respectively. The task is to design a controller and verify that it does not allow the car to violate the stop sign at position

0, so the safe set is $d_{sv} \leq 0$. The plant parameters are A_p, B_p, d_{init}, and v_{init}, which represent the maximum acceleration and braking capabilities of the car and the initial position and velocity of the car. In the verification model, we allow a large set of values, with only the minimal restrictions required to prove safety of the envelope, so that $\mathbf{P} = \{(A_p, B_p, d_{init}, v_{init}) \in \mathbb{R}^2 \mid A_p > 0 \wedge B_p > 0\}$. We do not specify restrictions on d_{init} and v_{init} at this time, since it will only play a role when checking that the initial state set is contained in the forward invariant. The d$\mathcal{L}$ model is shown in Fig. 7.3. The car is modeled as a double integrator (Line 7.11) restricted to evolve only when $v_{sv} \geq 0$. The envelope parameters are A_e, B_e, the acceleration and braking limits of the envelope; d_{on}, a position at which the envelope starts controlling the plant; and δ, an upper bound on the time interval at which the envelope expects to receive state measurement updates. The proposed forward invariant (Line 7.12) is that it is always safe to brake, the car always drives forward, and its position is always ahead of where the control envelope was originally engaged. The class of sampling sequences J_s has a single parameter, $s \in S = \{\tau \in \mathbb{R} \mid \tau > 0\}$, a worst-case sampling time. The sampling sequences under consideration are (t_j) such that for all $j, t_{j+1} - t_j < \tau$.

The parameters for the physical cars are $P = \{(A_p, B_p, d_{init}, v_{init})\}$ with $A_p \in (1.8, 2] m/s^2$ and $B_p \in (2, 3] m/s^2$, the controller is engaged at a distance $d_{init} = -20m$, and an initial velocity of at most $40mph \approx 18m/s$, so that $v_{init} \in [0, 18]$. We choose the sampling parameter $s = \tau = 0.1s$, so that S is the singleton $\{0.1\}$. From these choices of sets $P \subseteq \mathbf{P}$ and $S \subseteq \mathbf{S}$, we can find $E \subseteq \mathbf{E}_{p,s}$, which constrains $A_e \leq 1.8$, $B_e \leq 2$ and $\delta \geq 0.1$.

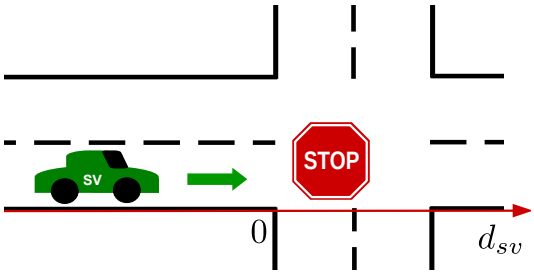


Figure 7.2: Car arriving at intersection.

$$\textbf{SSA1} \equiv \texttt{init} \rightarrow [\,(ctrl; plant)\,]\texttt{safety} \tag{7.1}$$

$$\texttt{init} \equiv \texttt{inv} \wedge\ 0 < A_e\ \wedge\ A_e \leq A_p \tag{7.2}$$

$$\wedge\ 0 < B_e\ \wedge\ B_e \leq B_p \tag{7.3}$$

$$\wedge\ \delta \geq \tau\ \wedge\ \tau > 0 \tag{7.4}$$

$$ctrl \equiv (a_{sv} := *; ?(\tag{7.5}$$

$$\left(\left(d_{sv} + \frac{v_{sv}^2}{2B_e} + \left(\frac{A_e}{B_e} + 1\right)\left(\frac{A_e}{2}\delta^2 + \delta v_{sv}\right) \leq 0\right)\right. \tag{7.6}$$

$$\wedge\ (a_{sv} \geq -B_e)\ \wedge\ (a_{sv} \leq A_e)) \tag{7.7}$$

$$\vee\ (a_{sv} = -B_e) \tag{7.8}$$

$$\vee\ ((v_{sv} = 0)\ \wedge\ (a_{sv} = 0)))) \tag{7.9}$$

$$plant \equiv t := 0; \tag{7.10}$$

$$\{d'_{sv} = v_{sv}, v'_{sv} = a_{sv}, t' = 1\ \&\ v_{sv} \geq 0, t \leq \tau\} \tag{7.11}$$

$$\texttt{inv} \equiv (d_{sv} + v_{sv}^2/(2B_e) \leq 0) \wedge (v_{sv} \geq 0) \wedge d_{sv} \geq d_{on} \tag{7.12}$$

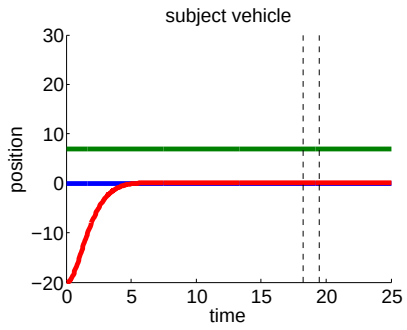
Figure 7.3: Stop Sign Assist, Stage 1.

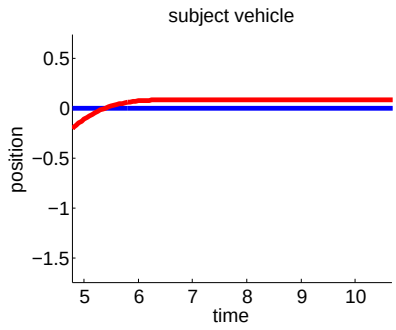
Design attempt without control envelopes

We first design a controller using linear quadratic regulation, without using the control envelope. We design a saturating state feedback controller of the following form.

$$g_c(d_{sv}, v_{sv}) = \begin{cases} 1.8 & \text{if} \quad -K_1 d - K_2 v > 1.8 \\ -2 & \text{if} \quad -K_1 d - K_2 v < -2 \\ -K_1 d - K_2 v & \text{otherwise} \end{cases} \tag{7.13}$$

The saturation bounds are the extremal bounds, which all of the cars can attain. The parameter vector is $c = (K_!, K_2)$. We use identity matrix for the LQR cost function, yielding the following





(a) Pure LQR controller violates the intersection (b) Pure LQR controller violates the intersection

gains.

$$K_1 = 0.9171 \qquad K_2 = 1.64 \tag{7.14}$$

A simulation of this controller is shown in 7.4a. Figure 7.4b shows a zoomed in version to clarify that the car violates the stop sign. Unfortunately, this controller violates the intersection. This is because the LQR design methodology is meant to stabilize a plant while minimizing a given cost function without regard for safety constraints. However, LQR is a valuable design tool, and we would like to be able to use control envelopes to retrofit this traditional technique to meet the requirements of today's control engineering challenges.

Design with control envelope

We will use the control envelope to retrofit LQR design to handle safety specifications. We consider control laws of the following form.

$$g_c(d_{sv}, v_{sv}) = \begin{cases} 1.8 & \text{if } -K_1 d - K_2 v > 1.8 \\ -2 & \text{if } -K_1 d - K_2 v < -2 \\ -K_1(d - d_{set}) - K_2 v & \text{otherwise} \end{cases} \tag{7.15}$$

As before, we use LQR to design the gains, and obtain

$$K_1 = 0.9171 \qquad K_2 = 1.64. \tag{7.16}$$

Next, we use quantifier elimination on the refinement conditions to provide bounds on the set-point, and obtain

$$d_{set} \leq -6.86. \tag{7.17}$$

Since we want the controller to leave the car as close as possible to the intersection, we choose $d_{set} = -6.86$ as our setpoint. A plot of the behavior of this controller is shown in 7.5. The car stops nearly $7m$ behind the stop sign. Although this controller is safe, it does a terrible job of satisfying the performance requirements.

Improving performance with refinement checking by parts

We have successfully retrofitted the LQR technique to handle safety constraints with the help of control envelopes. Unfortunately, the performance of the controller is poor, since the car ends up very far from the intersection. Next, we use the same parametric envelope–invariant pairs, with the help of refinement by parts, to design a controller that leaves the car closer to the intersection.

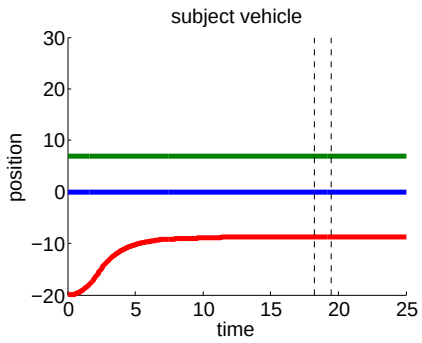


Figure 7.5: Safe controller with poor performance

We design a switching controller with four modes. The controller parameters include A_{c_i}, B_{c_i} with $i \in \{1, 2, 3, 4\}$, acceleration and braking saturation limits for each mode; K_1, K_2, feedback gains that are the same for all four modes; d_{switch_i}, switching thresholds; and d_{set_i}, setpoints. The parametrized control laws are of the form

$$g_c(d_{sv}, v_{sv}) = \begin{cases} h_1 & \text{if} \quad (d_{sv}, v_{sv}) \in D_1 \\ h_2 & \text{if} \quad (d_{sv}, v_{sv}) \in D_2 \\ h_3 & \text{if} \quad (d_{sv}, v_{sv}) \in D_3 \\ h_4 & \text{if} \quad (d_{sv}, v_{sv}) \in D_4 \end{cases}$$

with $D_i = \{(d_{sv}, v_{sv}) \mid v_{sv}^2/(2B_{c_i}) + d_{sv} \leq 0 \ \wedge \ d_{sv} \geq d_{switch_i}\}$ and each $h_i(d_{sv}, v_{sv})$ given by

$$h_i = \begin{cases} A_{c_i} & \text{if} \quad \gamma_i(d_{sv}, v_{sv}) \geq A_{c_i} \\ \gamma_i(d_{sv}, v_{sv}) & \text{if} \quad -B_{c_i} \leq \gamma_i(d_{sv}, v_{sv}) \geq A_{c_i} \\ -B_{c_i} & \text{if} \quad \gamma_i(d_{sv}, v_{sv}) \leq -B_{c_i} \end{cases}$$

$$\gamma_i = K_1(d_{set_i} - d_{sv}) - K_2 v_{sv}$$

The gains K_1, K_2 can be chosen using a standard design technique. We use LQR synthesis with a sampling rate of 0.1 and obtain $K_1 = 0.9171$, $K_2 = 1.64$. To complete the design, we need to choose values for the remaining controller parameters such that the controller is safe. We use Prop. 2 to verify each of the three modes of the controller with a different parametrization of the control envelope, to allow the car to get close to the intersection. Then $(A_{c_i}, B_{c_i}, d_{set_i}, d_{switch_i})$ with $i \in \{1, 2, 3, 4\}$ is the vector of the remaining parameters.

We can use the control envelope to examine parameter trade-offs. We assume that larger setpoints are preferred, since they allow the car to get closer to the intersection. We form the

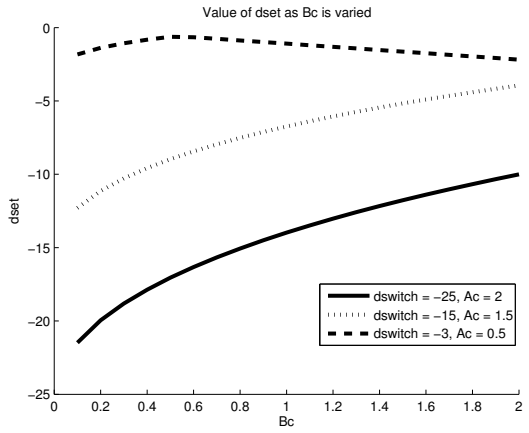
relevant quantifier elimination queries as described in Section 3.5.1 and solve for the remaining parameters. Fig. 7.6a shows how the largest safe values of the setpoint d_{set_i} generally increase as the braking bound B_{c_i} increases (with fixed d_{switch_i} and A_{c_i}), but very near the intersection a lower braking bound produces a better setpoint. Fig. 7.6b shows how the largest safe value of d_{set_i} generally decreases as the upper bound on acceleration A_{c_i} increases (with fixed d_{switch_i} and B_{c_i}, but we can see that d_{set_i} is significantly less sensitive to variations in A_{c_i}. Fig. 7.6c shows that the largest safe value of d_{set_i} generally increases as d_{switch_i} increases (for fixed choices of A_{c_i} and B_{c_i}).

With this information, we choose the following parameter valuations for the controller; $(A_{c_1} = 1.8, B_{c_1} = 2, d_{switch_1} = -20, d_{set_1} = -6.86)$, $(A_{c_2} = 1.8, B_{c_2} = 2, d_{switch_2} = -10, d_{set_2} = -2.2)$, and $(A_{c_3} = 0.1, B_{c_3} = 0.8, d_{switch_3} = -5, d_{set_3} = -0.97)$. $(A_{c_4} = 0.1, B_{c_4} = 0.3, d_{switch_4} = -2, d_{set_4} = -0.44)$. The corresponding envelope parameter valuations that ensure that the conditions of Prop. 2 hold are $(A_e = 1.8, B_e = 2, d_{on} = -20, \delta = 0.1)$, $(A_e = 1.8, B_e = 2, d_{on} = -10, \delta = 0.1)$, $(A_e = 0.1, B_e = 0.8, d_{on} = -5, \delta = 0.1)$, and $(A_e = 0.1, B_e = 0.3, d_{on} = -2, \delta = 0.1)$.

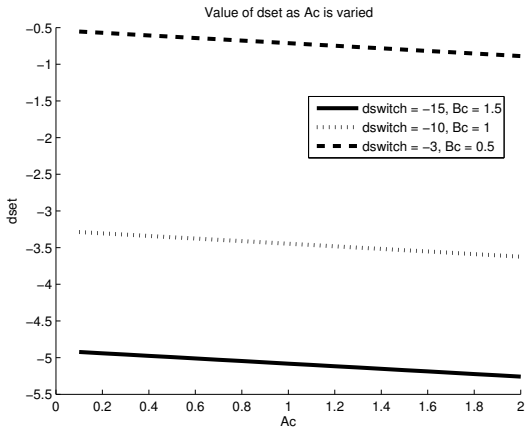
Since this is the controller we will use, we defer the plot of its behavior until after the controller for stage 2 has been designed.

7.4 Stop sign assist stage 2

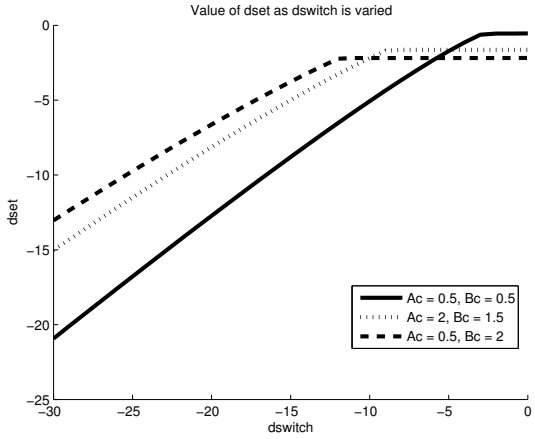
In SSA2 (Fig. 7.7), the SV is stopped at or slightly behind the entrance of the intersection, and wants to safely cross. Another vehicle, called the principal other vehicle (POV), is approaching



(a) d_{set} as a function of B_c.



(b) d_{set} as a function of A_c.



(c) d_{set} as a function of d_{switch}.

the intersection from a perpendicular direction. The goal is to produce a controller to safely drive the SV through the intersection without a collision. To do this, we require that the POV is not in the intersection at the same time as the SV, so X_{safe} is $d_{pov} < 0 \vee d_{sv} < 0 \vee d_{sv} \geq q$.

We take the initial position of the SV as the origin of the coordinate system. It is initially stopped and must travel at most a distance q to clear the intersection. For the POV, we assume that it never drives backwards, and that it has the same braking and acceleration bounds as the SV. We have $\mathbf{P} = \{(A_p, B_p, q) \in \mathbb{R}^3 \mid A_p > 0 \ \wedge \ B_p > 0 \ \wedge \ q > 0\}$. To examine the

worst-case scenario, the POV is assumed to be accelerating at its physical upper limit. The d$\mathcal{L}$ model is shown in Fig. 7.8. The state equation describes the two cars as double integrators (Lines 7.17 through 7.19) restricted to nonnegative velocities. We assume the same class of sampling sequences as before. We use the set of envelope parameters $A_e^{pov}, A_e^{sv}, B_e^{sv}, \delta, r, D_e$, constrained as shown in lines 7.4 through 7.8. The control envelope (Lines 7.10 through 7.15) may take one of four actions. If it is stopped at the intersection, it may choose to remain stopped, or accelerate into the intersection if it determines that it can traverse the intersection before the POV arrives by applying a minimal acceleration of D_e. If it is inside the intersection, it must continue driving through, with a minimal driving acceleration of D_e and a maximal acceleration A_e. If it has passed a position r outside the intersection, it may choose any acceleration or braking within envelope parameters A_e and $-B_e$. The forward invariant (Lines 7.20 through 7.24) is the union of three sets: either the car is stopped at the intersection, or it is past the intersection, or it can safely drive through the intersection with some acceleration between D_e and A_e, and if it is at its initial position, it is stopped. In all of the above cases, neither of the cars should be driving backwards.

As to the specific systems we wish to verify, we assume that the car starts within 1 meter of the intersection, that the intersection is between 4 and 6 meters long, and that we have the same braking and acceleration limits as before. If we set the origin of the coordinate system to be at the position of the car, we have $P = \{(A_p, B_p, q) \mid 2 \leq A_p \leq 3 \wedge 2 \leq B_p \leq 2.5 \wedge 4 \leq q \leq 7\} \subseteq \mathbf{P}$, and the initial set X_p^0 is $d_{sv} = 0$ and $v_{sv} = 0$. We consider control laws of the following form,

$$g_c(x) = \begin{cases} h_c(x) & \text{if} \quad (d_{pov} \leq -d_{gap} \ \wedge \ v_{pov} \leq v_{lim}) \ \vee \ d_{sv} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{7.25}$$

$$h_c(x) = \begin{cases} A_c & \text{if} \quad K_1(d_{set} - d_{sv}) - K_2 v_{sv} > A_c \\ a_c & \text{if} \quad K_1(d_{set} - d_{sv}) - K_2 v_{sv} < a_c \\ K_1(d_{set} - d_{sv}) - K_2 v_{sv} < a_c & \text{otherwise} \end{cases} \tag{7.26}$$

with parameters

$$C = \{(A_c, a_c, d_{gap}, v_{lim}, K_1, K_2, d_{set}) \mid \forall (A_p, B_p, q) \in \mathbf{P}. \ -B_p \leq a_c \leq A_c \leq A_p\}$$

From LQR, we obtain $K_1 = 0.9171$ and $K_2 = 1.64$. For the other parameters, we try $A_c = 2$, $a_c = 1$, $d_{gap} = 40$ $v_{lim} = 5$, $d_{set} = 7$. We find that the conditions of Prop. 1 are met with $A_e^{sv} = 2, B_e^{sv} = 2, A_e^{pov} = 3; \delta = 0.1, r = 7, D_e = 1$ and the controller is proven safe.

Now that both stages have been designed, we show a behavior of the overall system. Figure 7.9a shows the behavior of the subject vehicle, which approaches the intersection and then pauses at the stop sign while it waits for principal other vehicle to pass. The position of the principal other vehicle is shown in 7.9b. The principal other vehicle passes through the intersection during the time interval marked by the dashed lines. The subject vehicle continues through the intersection as soon as the principal other vehicle has passed.

7.5 Summary

In this chapter, we presented a case study of a controller design task for a stop sign assist system. This system is comprised of two stages. In the first stage, the controlled vehicle is approaching

a stop sign and must stop without entering the intersection. In the second stage, the vehicle is safely stopped at the beginning of the intersection, and must traverse the intersection while avoiding a collision with an oncoming car.

First, we designed a parametric envelope–invariant pair and verified it with theorem prover KeYmaera.

The main advantages of using an envelope–invariant pair were that a traditional control design technique that does not provide safety guarantees was supplemented with the control envelope to ensure safety. Also, the fact that the control envelope is an input–output relation agnostic of controller architecture allowed us to experiment with different controllers until the performance requirements were met.

For stage 1, we next used the technique of linear quadratic regulation to design a state-variable feedback controller for the vehicle. Then, we used the technique of refinement by parts to verify that the controller was safe when the setpoint was chosen according to a switched scheme to allow the car to approach the intersection.

For stage 2, we again used linear quadratic regulation to provide feedback gains, and then used the control envelope to verify that the controller would successfully avoid a collision if it guided the car to traverse the intersection when the distance of the oncoming car was a prespecified value.

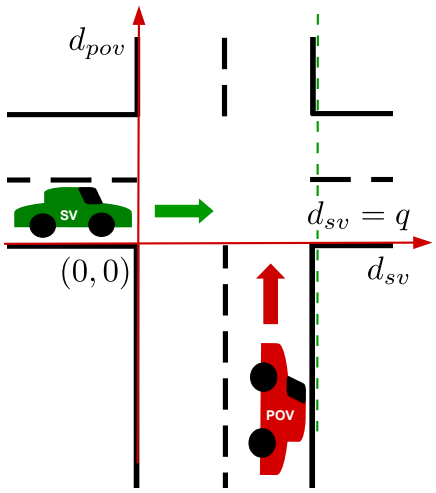


Figure 7.7: Stage 2.

$$\textbf{SSA2} \equiv \texttt{init} \rightarrow [\,(ctrl; plant)\,]\texttt{inv} \tag{7.1}$$

$$\texttt{init} \equiv \texttt{inv} \wedge A_p > 0 \wedge B_p > 0 \wedge q > 0 \tag{7.2}$$

$$\wedge\ \tau > 0 \wedge t = 0 \tag{7.3}$$

$$\wedge\ \delta > 0 \wedge \delta \geq \tau \tag{7.4}$$

$$\wedge\ 0 < B_e^{sv} \leq B_p \tag{7.5}$$

$$\wedge\ 0 < A_e^{sv} \leq A_p \wedge 0 < D_e \leq A_e^{sv} \tag{7.6}$$

$$\wedge\ A_p \leq A_e^{pov} \tag{7.7}$$

$$\wedge\ r \geq q \tag{7.8}$$

$$ctrl \equiv a_{sv} := *; \tag{7.9}$$

$$?((a_{sv} = 0 \wedge v_{sv} = 0 \wedge d_{sv} = 0) \tag{7.10}$$

$$\vee\ (d_{sv} \geq r \wedge\ -B_e^{sv} \leq a_{sv} \leq A_e^{sv}) \tag{7.11}$$

$$\vee\ ((\forall z \geq 0.\ \tfrac{1}{2} D_e z^2 < r) \tag{7.12}$$

$$\rightarrow \left(\tfrac{1}{2} A_e^{pov} z^2 + v_{pov} z + d_{pov} < 0\right) \tag{7.13}$$

$$\wedge\ D_e \leq a_{sv} \leq A_e^{sv} \wedge d_{sv} = 0 \wedge v_{sv} = 0) \tag{7.14}$$

$$(d_{sv} > 0 \wedge a_{sv} \geq D_e \wedge a_{sv} \leq A_e^{sv}) \tag{7.15}$$

$$plant \equiv t := 0; \tag{7.16}$$

$$\{d'_{sv} = v_{sv}, v'_{sv} = a_{sv}, \tag{7.17}$$

$$d'_{pov} = v_{pov}, v'_{pov} = A_p, \tag{7.18}$$

$$t' = 1\ \&\ (v_{sv} \geq 0,\ v_{pov} \geq 0,\ t \leq \tau)\} \tag{7.19}$$

$$\texttt{inv} \equiv (v_{sv} \geq 0 \wedge v_{pov} \geq 0 \wedge \tag{7.20}$$

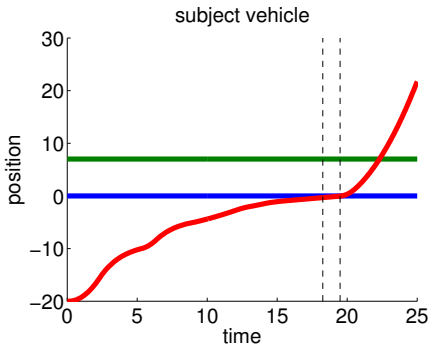
$$((d_{sv} \geq r) \vee (d_{sv} = 0 \wedge v_{sv} = 0) \tag{7.21}$$

$$\vee\ \Big(\left(\forall z \geq 0.\ \tfrac{1}{2} D_e z^2 + v_{sv} z + d_{sv} < r\right) \tag{7.22}$$

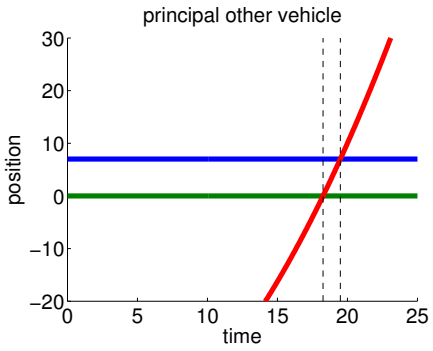
$$\rightarrow \left(\tfrac{1}{2} A_e^{pov} z^2 + v_{pov} z + d_{pov} < 0\right) \tag{7.23}$$

$$\wedge\ d_{sv} \geq 0 \wedge (d_{sv} = 0 \rightarrow v_{sv} = 0)\Big) \tag{7.24}$$

Figure 7.8: Stop Sign Assist, Stage 2.



(a) Stop sign assist overall controller, SV



(b) Trajectory of the POV

Chapter 8

Forward invariant cuts

8.1 Introduction

Theorem provers can address verification tasks that involve symbolic parameters. Further, theorem proving approaches are not prone to numerical artifacts that often affect explicit reachable-set computation techniques. Theorem provers, however, suffer from a severe lack of automation, and require human intervention to simplify proof tasks. In the specific case of safety verification, an important source of poor automation arises from the difficulty of computing global system invariants that contain all system behaviors starting from the initial set.

In many cases, it is easier to compute local, forward invariants that hold for all future time after a system trajectory enters it, but not necessarily from the initial set. Our contribution is to improve the degree of automation of a theorem prover by using forward invariants, computed through several heuristics.

Intuitively, if a safe forward invariant can be found, it is possible to simplify the verification

task by verifying only those behaviors that do not start within the forward invariant, and only as long as they do not enter the forward invariant, since any behavior that is safe until it enters the safe forward invariant will be always safe. We have implemented this reasoning as a cut-style proof rule for the logic d$\mathcal{L}$, which is supported by theorem prover KeYmaera.

A cut-style proof rule is a rule in which a logical formula that appears in the premises of the rule does not appear in its conclusion, i.e., it has been *cut*. Cut rules arise when an intermediate reasoning step is required to establish a proof result, such as in the case of a lemma.

We validated the benefits of our forward invariant cut on an example that theorem prover could not verify by searching for a global invariant. We found that the only necessary user interactions were the application of our proof rule. We also have a simplified model of an air-fuel controller, which we verify with the help of our proof rule and with some user interactions to aid theorem prover in handling the difficult arithmetic.

8.2 Background and related work

A *cut* rule is a rule in which something that appears in the premises does not appear in the conclusion, i.e., is *cut out*. Numerous cut rules exist for different logics, but perhaps the most famous and easily understood is Gentzen's cut rule for sequent calculus[40], which encodes the logical structure of the use of a lemma in a proof.

$$\frac{A \vdash C \qquad C \vdash B}{A \vdash B} \tag{8.1}$$

The first premise of the rule says that from the statement A we can prove the statement C, and the second premise says that from statement C we can prove statement B. Hence, the conclusion

of the rule *cuts out* the formula C, which functions as an intermediate lemma, and concludes that from statement A we can prove statement B.

Lemmas are a powerful tool within mathematics at large to organize proofs and simplify proof tasks, providing clarity and modularity. From the perspective of automatic proof generation, however, cut rules present a significant obstacle to automation [6]. To see why this is the case, consider the proof task $A \vdash B$, to prove assertion B from the assumption A.

If the logic does not contain proof rules with a cut structure, a proof can be constructed by simple enumeration, as follows. Consider all candidate proofs of length one, which are simply single proof rule applications. Next, attempt to use each of these proofs of length one to the desired proof task. If any succeeds, terminate. Otherwise, consider all possible proofs of length two, which consist of all (ordered) pairs of proof rules. Check whether any of these pairs constitutes a proof of the desired statement. If not, try again with candidate proofs of length three. If a proof for the desired statement exists, it has finite length n and will be eventually found with this procedure. Further developments in theorem proving for such a logic now only need to focus on efficiency improvements.

If the logic contains cut-style proof rules, some external procedure is required to creatively propose instantiations for the element that gets cut out. Indeed, moving from the premises to the conclusion of the cut rule destroys information, since the cut formula is lost, and constructing a proof tree requires some way of recovering that information.

In light of these difficulties, one may be tempted to suggest that the logic should be reformulated so that it does *not* contain any cut-style proof rules—however, it is not clear how to refactor proof systems in this way, and more importantly, it is not always possible. This explains

the significance of Gentzen's Cut Elimination Theorem [40], which states that for his proposed sequent calculus, any statement that has a proof with the cut rule also has a proof without the cut rule. As a result, the cut rule is superfluous and can simply be deleted from the set of proof rules. As a result, the proof system is profoundly simplified, and in principle can be reduced to a brute-force mechanism that enumerates and checks candidate proofs.

Cut rules, then, are a central figure in the quest for proof automation. Lamentably, positive results like Gentzen's cut elimination theorem are not always possible. In the specific case of logics for hybrid systems, the logic d$\mathcal{L}$ contains a rule called a *differential cut* [88] to manage its use of differential invariants, and in [92] it was shown that this proof rule cannot be eliminated. This means that there exist theorems of d$\mathcal{L}$ that have proofs with the differential cut rule but cannot be proven without it. To our knowledge, this is the only attempt to even address the issue of cut elimination for logics of hybrid systems. So far, this fundamental problem has received little attention, and the scarce available results suggest that it may not be possible to formulate a logic for hybrid systems in which cut rules can be eliminated.

To improve proof search automation in a deductive system with cut rules that cannot be eliminated, one must have a good way to produce instantiations for the formula that is cut by the proof rule application. The chief strength of our forward invariant cut rule is that it allows leveraging system designer insights, such as invariance of modes or regions of the hybrid state space, to find good candidates for the cut formula. In combination with the simulation-driven invariant search techniques of [60], the forward invariant cut can be a powerful tool to take coarse, high-level intuitions about system functionality to significantly simplify the proof process. As we will show in our case studies, this can significantly aid in increasing the degree of automation

of proof construction. The value of this contribution is heightened in light of the discouraging preliminary results that it may not be possible to construct a fully cut-free proof calculus for hybrid systems.

In other logical frameworks, *Craig interpolants* have been used to provide useful instantiations for terms that get cut [28], and they have been used successfully in a model checking framework [76].

An existing cut within the d$\mathcal{L}$ calculus, which in this work we refer to as the *safety certificate cut* requires the use of global invariants, that exclude the unsafe set and fully contain the initial set. For complex systems, it is generally difficult to compute these safety certificates. In the case of systems composed of simpler subsystems, however, forward invariants for the subsystems are often readily available, and it is important to be able to compose the forward invariants of the subsystems to reason about the behaviors of the entire system.

Platzer and Clarke [93] have proposed a technique called *loop saturation*, in which a safety certificate is constructed incrementally by using initialized invariant sets that are invariant and contain the initial set, but do not individually imply that the whole system is safe. This procedure can be thought of as providing an increasingly refined overapproximation of the reachable set. Each initialized invariant constraint that is added to the candidate safety certificate refines the approximation of the reachable set. The work in [93] also describes a technique to intelligently propose initialized invariant sets that may be useful in proving the desired safety property. Our use of forward invariant cuts can be thought of as a dual procedure. Instead of proposing initialized invariants to refine an overapproximation of the reachable set, we propose safe invariants to excise portions of the initial set that evolve into safe invariants. The two techniques are natural

complements of each other. In light of this contrast, a failed proof attempt using loop saturation can help to learn which parts of the safety condition are difficult to satisfy, whereas a failed proof attempt with forward invariant cuts can help to learn which parts of the initial set are problematic for safety verification.

Another existing cut rule within the d$\mathcal{L}$ calculus, called a differential cut, leverages knowledge of invariance properties of continuous portions of a hybrid program. Our forward invariant cut rule is different in that it can be applied to general hybrid programs.

In contrast with differential cuts, forward invariant cuts are not restricted to continuous programs, and can be applied equally effectively to purely discrete as well as fully hybrid programs. Furthermore, while differential cuts are applied late in the proof process, after extensive simplification, forward invariant cuts are applied early in the proof process *as a means of proof simplification* with designer insights. Also, forward invariant cuts use a set that is invariant and safe, in contrast with differential cuts, which are initialized and invariant. In this sense, they complement differential cuts. The two rules combine to provide an expressive proof calculus that encourages natural reasoning.

Damm et al. [29] leverage component-wise reasoning to develop a framework to support incremental design and verification using a library-based approach, using components with specified regions of safety and stability. In this way, a stable system can be constructed to be safe and stable by composing safe and stable subsystems, and ensuring that certain interface requirements are met. This is relevant to our work because, as we have seen, Lyapunov function sublevelsets serve as forward invariants.

In [80], a hybrid automaton is decomposed into strongly connected components to examine

its stability properties. A Lyapunov function is computed for each component using semidefinite optimization in a way that ensures that the resulting functions can be composed to guarantee stability of the entire system. This approach has three advantages, namely, (i) improved numerical stability of the optimization procedure, (ii) possibility of incremental construction of systems from stable components, and (iii) easy diagnosis of unstable portions of the system.

8.3 The forward invariant cut

The logic d$\mathcal{L}$ as implemented in KeYmaera supports several cut-style proof rules. Notably, a version of Gentzen's cut rule, a *differential cut* rule to manage differential invariants, and the following rule for using safety certificates.

$$\frac{I \rightarrow C \quad C \rightarrow [\alpha]C \quad C \rightarrow S}{I \rightarrow [\alpha^*]S} \tag{8.2}$$

The first premise states that in the initial conditions, in which I is true, C also holds, which means that all behaviors are initialized in the set C. The second premise says that when starting in a state such that C is true, any possible execution of the program α will terminate in a state such that C is true, which establishes that C is a forward invariant. The third premise states that if C is true, the safety condition S is also true, which establishes that C describes a safe set. The conclusion of the rule states that from any initial condition stipulated by I, arbitrarily many runs of the system will always remain in the safe set S. Note that the global invariant represented by C is cut and does not appear in the conclusion.

In general, global invariants may be difficult to find. Instead, we propose the use of forward invariants to simplify the proof process. A (safe) forward invariant will satisfy the last two

premises of rule (8.2), since it captures all system behaviors that enter it and is contained in the safe set. However, the first premise of the rule will not be satisfied, since the system does not necessarily begin in the forward invariant. Hence, the rule is not applicable. Instead, we propose a proof rule that replaces the first premise with a reduced proof task. To establish safety of the system, we require that the system trajectories that do not begin in C and do not pass through C during system execution be verified by a separate, reduced proof task. To do this, we replace the first premise with a premise of the form $I \wedge \neg C \rightarrow [(\alpha; ?\neg C)^*]S$. This leads to the *forward invariant cut rule*, shown below.

$$\frac{I \wedge \neg C \rightarrow [(\alpha; ?\neg C)^*]S \quad C \rightarrow [\alpha]C \quad C \rightarrow S}{I \rightarrow [\alpha^*]S} \tag{8.3}$$

The first premise in this rule is of the form $\hat{I} \rightarrow [\hat{\alpha}^*]S$, which is a safety proof obligation with the same structure as the original proof task, so that standard techniques can be applied to continue with the reasoning. The initial set $\hat{I}$ of this new obligation is $I \wedge \neg C$, the set of initial states that are not contained in the forward invariant. The modified hybrid program $\hat{\alpha}$ is $\alpha; ?\neg C$, which can reach the same states as α, except for those contained in C. In this way, the proof rule requires showing that behaviors of the system are safe until they enter the safe forward invariant C, after which time they are guaranteed safe.

8.4 Examples

8.4.1 Three mode system

Consider the system of Fig. 8.1, which has three stable modes and one failure mode. The verification task is to ensure that the system never reaches the failure mode. This example cannot

be verified automatically using KeYmaera's existing capabilities for finding global invariants. However, the system has known forward invariants. After applying the forward invariant cut rule to leverage knowledge of these invariants, KeYmaera is able to verify the system without further user interaction.

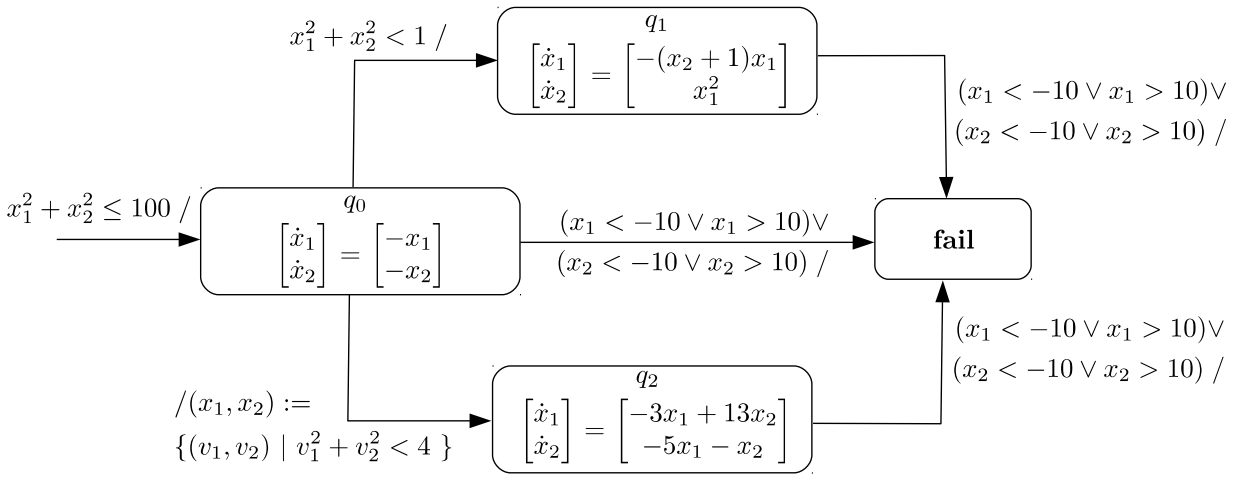


Figure 8.1: Three mode system

We use the canonical hybrid program representation described in [90], in which a hybrid program is written for each mode and each edge. The overall system is represented by the nondeterministic choice of these subprograms, repeated nondeterministically.

Mode q_1 has a Lyapunov function of the form $V_1(x_1, x_2) = \frac{1}{2}x_1^2 + \frac{1}{2}(x_2 - 2)^2$ as described in Example 4.10 of [62]. The sublevelset $V_1(x_1, x_2) \leq 5$ contains the guard into mode q_1. We apply the forward invariant cut rule with $C_1 = V_1(x_1, x_2) \leq 5 \wedge M = q_1$, a set that is a forward invariant contained in the safe set, but cannot be used with the conventional global invariant proof rule since it does not contain the initial mode q_0 of the hybrid system. The rule application causes the proof tree to split into three branches. The second branch requires showing that whenever the system begins in C_1, it remains in C_1. KeYmaera can readily check that since the proposed

sublevelset excludes the guard into **fail**, C_1 will in fact be satisfied by the end of each system trace. The third branch of the proof tree requires showing $C \rightarrow S$, which is trivial, since S is simply $M \neq q_{fail}$ and C stipulates $M = q_1$. We now turn our attention to the first branch.

Mode q_2 has a Lyapunov function $V(x_1, x_2) = 2x_1^2 + 4x_2^2$, computed using standard Lyapunov techniques for linear systems. The sublevelset $V_2(x_1, x_2) \leq 16$ contains the circle of radius 2. All incoming transitions to mode q_2 reset the system state within this circle. By applying a forward invariant cut with $C_2 = V_2(x_1, x_2) \leq 16 \wedge M = q_2$, we again get three branches. Since $V_2(x_1, x_2) \leq 16$ excludes the guard to **fail**, KeYmaera can show that C_2 represents a safe set. The next branch is to prove that C_2 implies safety, which is easy because C_2 requires $M = q_2$, which implies $M \neq q_{fail}$. This branch can now be easily proved with the standard tools of KeYmaera, using the loop invariant $M = q_0 \wedge x_1^2 + x_2^2 \leq 100$.

8.4.2 Non-autonomous Switched System

Consider an open two-mode system, where an external input can cause the system to arbitrarily switch between the system modes. This example is significant, because neither of the two modes is invariant, so we cannot use the forward invariant cut as in the previous examples to excise entire modes. Instead, each forward invariant cut excises regions of hybrid state space.

The continuous dynamics are define by matrices A_1 and A_2, as given below:

$$A_1 = \begin{bmatrix} -1.0 & 4.0 \\ -0.25 & -1.0 \end{bmatrix},$$

$$A_2 = \begin{bmatrix} -1.0 & -0.25 \\ 4.0 & -1.0 \end{bmatrix}.$$

Linear reset maps are applied to the state when a transition is made between Modes 1 and 2. The resets are defined by matrices R_{12} and R_{21}:

$$R_{12} = \begin{bmatrix} -0.0658 & -0.0123 \\ 0.1965 & -0.0658 \end{bmatrix},$$

$$R_{21} = \begin{bmatrix} -0.0658 & 0.1965 \\ -0.0123 & -0.0658 \end{bmatrix}.$$

Figure 1 defines the hybrid program for this system. For both modes, the continuous-time dynamics given by A_1 and A_2 are stable and linear. It is well known that even for switched-mode systems with stable linear continuous dynamics, switching conditions exists that lead to instability for the switched system [18]. We wish to prove that it is not possible to switch between A_1 and A_2 to create unstable behavior. The safety property for this system is that it should remain within $\|\mathbf{x}\|_\infty < 2.0$. We apply the forward invariant cut rule to the example to successfully prove the safety property. Below, we describe the steps of the proof.

Here, the designer provided two forward invariants of the system by independently solving the Lyapunov equation for the linear dynamics of the system in each of the modes. The designer then picked level set sizes to ensure that the resulting forward invariant is contained within the safe set S. The invariants are given below:

$$C_1 = \{\mathbf{x} \mid V_1(\mathbf{x}) < l_1\} \tag{8.4}$$

$$C_2 = \{\mathbf{x} \mid V_2(\mathbf{x}) < l_2\} \tag{8.5}$$

Here, $V_1(\mathbf{x}) = 0.3828x_1^2 + 0.9375x_1x_2 + 2.3750x_2^2$, and $l_1 = 1.0$, and $V_2(\mathbf{x}) = 2.3750x_1^2 + 0.9375x_1x_2 + 0.3828x_2^2$, and $l_2 = 1.0$.

$$\text{TS} \equiv I \rightarrow [(\mathtt{s} \cup \mathtt{m}_1 \ \cup \mathtt{m}_2)^*]S$$

$$\text{I} \equiv M = 1 \wedge x_1^2 + x_2^2 \leq 0.49$$

$$\mathtt{s} \equiv M := 1 \cup M := 2$$

$$\mathtt{m}_1 \equiv (?M = 1);$$

$$x_1 := -0.0658x1 + 0.1965x2;$$

$$x_2 := -0.0123x1 - 0.0658x2;$$

$$\{x_1' = -x_1 + 4x_2, x2' = -(1/4)x1 - x2\}$$

$$\mathtt{m}_2 \equiv (?M = 2);$$

$$x_1 := -0.0658x1 - 0.0123x2$$

$$x_2 := 0.1965x1 - 0.0658x2$$

$$\{x1' = -x1 - (1/4)x2, x2' = 4x1 - x2\}$$

$$\text{S} \equiv x1 > -2 \wedge x1 < 2 \wedge x2 > -2 \wedge x2 < -2$$

Figure 1: A d$\mathcal{L}$ model of the nonautonomous switched system

We sequentially apply two forward invariant cuts in order to prove Figure 1 safe. The first forward invariant cut rule uses the set C_1 as the cut. After applying C_1, the proof tree has three branches: $I \wedge \neg C_1$, $C_1 \rightarrow [\alpha]C_1$, and $C_1 \rightarrow S$. Of these, the third branch is trivially true as $C_1 \subseteq S$. To prove the second branch valid, KeYmaera needs to prove that C_1 is invariant for the disjuncts.

For the hybrid program $\mathtt{m}_1$, KeYmaera computes the forward image of the set C_1 when transformed by the linear transformation R_{21}, i.e., the set $F = \{\mathbf{y} \mid \mathbf{y} = R_{21}\mathbf{x} \wedge V_1(\mathbf{x}) < l_1\}$. Note that this step requires performing quantifier elimination, and KeYmaera utilizes Mathematica for this purpose. It then uses C_1 as a differential invariant to prove that $F \rightarrow [\{\mathbf{x}' = A_1\mathbf{x}\}]C_1$. This

is facilitated by the fact that C_1 is in fact invariant for the linear system $\dot{\mathbf{x}} = A_1\mathbf{x}$.

The difficult branch is the one requiring us to prove that C_1 is invariant for mode $\mathtt{m}_2$. To do so, we assist KeYmaera with certain lemmas; the intuition for these lemmas is as follows: Any state in set C_1 upon executing the program $\mathtt{m}_2$ gets linearly transformed by R_{12}. Let $\hat{C}_1 = \{\hat{\mathbf{x}} \mid \mathbf{x} \in C_1, \hat{\mathbf{x}} = R_{12}\mathbf{x}\}$ represent the forward image of C_1 under R_{12}. Next, we show that the set $\hat{C}_1$ is a subset of a specific sublevelset C_2^* of $V_2(\mathbf{x})$. As C_2^* is a sublevelset of $V_2(\mathbf{x})$, it is invariant under the dynamics $\dot{\mathbf{x}} = A_2\mathbf{x}$; thus, any state beginning in C_2^* will remain in C_2^*. Finally, we choose C_2^* in such a way that $C_2^* \subseteq C_1$. This essentially proves that any state starting in the set C_1 will be contained in set $\hat{C}_1$, of which any state will under the dynamics $\dot{\mathbf{x}} = A_2\mathbf{x}$ remain in the set C_2^*, i.e., in the state C_1.

Formally, we establish the following:

$$C_1 \rightarrow [\mathbf{x} := R_{12}\mathbf{x}]\hat{C}_1 \tag{8.6}$$

$$\hat{C}_1 \subseteq C_2^* \tag{8.7}$$

$$C_2^* \rightarrow [\{\mathbf{x}' = A_2\mathbf{x}\}]C_2^* \tag{8.8}$$

$$C_2^* \subseteq C_1 \tag{8.9}$$

We can combine these to infer that $C_1 \rightarrow [\mathtt{m}_2]C_1$.

Finally, the first branch of the proof considers $I \wedge \neg C_1$; this contains the set of initial states not in C_1. These can now be addressed by the second forward invariant cut (set C_2) following a symmetric argument as above. After applying the second cut C_2, the first branch has an empty antecedent ($I \wedge \neg C_1 \wedge \neg C_2$ is empty), i.e., the proof has accounted for all initial states, which closes the proof.

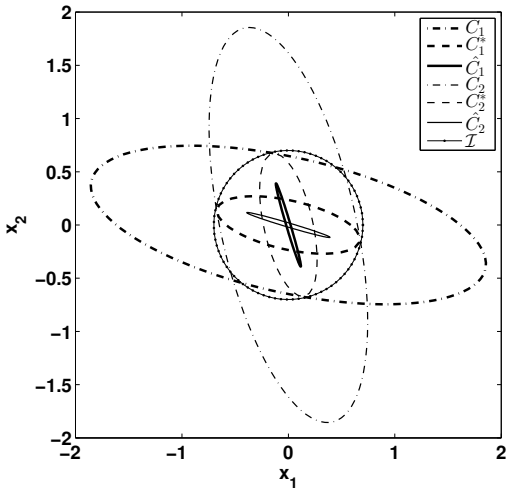


Figure 8.2: Illustration of forward invariant sets for Example 8.4.2.

8.4.3 Engine fuel control

Our second case study is a hybrid system representing an automotive fuel control application. Environmental concerns and government legislation require that the fuel economy be maximized and the exhaust gas emissions (e.g., hydrocarbons, carbon monoxide, and nitrogen oxides) be minimized. At the ideal air-to-fuel (A/F) ratio, also known as the *stoichiometric* value, both these quantities are optimized. We present an automotive control system whose purpose is to accurately regulate the A/F ratio.

The system dynamics and parameters were derived from a published model [56] and then simplified, as in [60]. The model consists of a simplified version of the physics of engine subsystems responsible for air intake and A/F ratio measurement, along with a computer control system tasked with regulating the A/F ratio. The objective of the controller is to maintain the A/F ratio within 10% of the nominal operating conditions. The experiment that we model involves an engine connected to a dynamometer—a device that can control the speed of the engine and measure

the output torque. In our setting, the dynamometer maintains the engine at a constant rotational velocity. The controller has two modes of operation: (1) a $recovery$ mode, which controls fuel in an open-loop manner, i.e., with only feedforward control action, where the system runs for at most 8ms, and (2) a normal run mode, which uses feedback control to regulate the A/F ratio.

The controller measures both the air flow through the air-intake manifold, which it uses to estimate the air pressure in the manifold, and the oxygen content of the exhaust gas, which it uses to compute the A/F ratio. The $recovery$ mode represents the behavior of the controller when recovering from a sensor fault (e.g., aberrant sensor readings, environmental conditions that cause suspicion of the sensor readings). During the $recovery$ mode, the controller has no access to oxygen sensor measurements and so must operate in a feedforward manner (i.e., using only the manifold air flow rate). The normal mode is the typical mode of operation, where the oxygen sensor measurements are used to do feedback control.

System dynamics for the Engine Fuel Control Model

We now present the model parameters and the ODEs for the Engine Fuel Control model. Figure 8.3 details the equations for the $recovery$ mode, and Fig. 8.4 provides the dynamic equations for the $normal$ mode. In the figures, $\frac{dp}{dt} = f_p$, $\frac{dr}{dt} = f_r$, $\frac{dp_{est}}{dt} = f_{p_{est}}$, and $\frac{di}{dt} = f_i$.

We translate the system so that the origin coincides with the $normal$ equilibrium point $p \approx 0.8987$, $r = 1.0$, $p_{est} \approx 1.077$, $i \approx 0.0$ and call the translated variables $\hat{p}$, $\hat{r}$, $\hat{p}_{est}$, and, $\hat{i}$, respectively.

$$
\begin{aligned}
f_p &= c_1 \left(2\hat{u_1} \sqrt{\frac{p}{c_{11}} - \left(\frac{p}{c_{11}}\right)^2} - \left(c_3 + c_4 c_2 p + c_5 c_2 p^2 + c_6 c_2^2 p\right) \right) \\
f_r &= 4 \left(\frac{c_3 + c_4 c_2 p + c_5 c_2 p^2 + c_6 c_2^2 p}{c_{13}(c_3 + c_4 c_2 p_{est}^2 + c_5 c_2 p_{est}^2 + c_6 c_2^2 p_{est})} - r \right) \\
f_{p_{est}} &= c_1 \left(2\hat{u_1} \sqrt{\frac{p}{c_{11}} - \left(\frac{p}{c_{11}}\right)^2} - c_{13} \left(c_3 + c_4 c_2 p_{est} + c_5 c_2 p_{est}^2 + c_6 c_2^2 p_{est}\right) \right) \\
f_i &= 0
\end{aligned}
$$

Figure 8.3: System dynamics for the Engine Fuel Control System in the $recovery$ mode.

$$
\begin{aligned}
f_p &= c_1 \left(2\hat{u_1} \sqrt{\frac{p}{c_{11}} - \left(\frac{p}{c_{11}}\right)^2} - \left(c_3 + c_4 c_2 p + c_5 c_2 p^2 + c_6 c_2^2 p\right) \right) \\
f_r &= 4 \left(\frac{c_3 + c_4 c_2 p + c_5 c_2 p^2 + c_6 c_2^2 p}{c_{13}(c_3 + c_4 c_2 p_{est}^2 + c_5 c_2 p_{est}^2 + c_6 c_2^2 p_{est})(1 + i + c_{14}(r - c_{16}))} - r \right) \\
f_{p_{est}} &= c_1 \left(2\hat{u_1} \sqrt{\frac{p}{c_{11}} - \left(\frac{p}{c_{11}}\right)^2} - c_{13} \left(c_3 + c_4 c_2 p_{est} + c_5 c_2 p_{est}^2 + c_6 c_2^2 p_{est}\right) \right) \\
f_i &= c_{15}(r - c_{16})
\end{aligned}
$$

Figure 8.4: System dynamics for the Engine Fuel Control System in the $normal$ mode.

Hybrid program for engine fuel control

Figure 2 is a hybrid program representing this system. The state variables $\hat{p}, \hat{r}, \hat{p}_{est}$, and $\hat{i}$ represent the manifold pressure, the ratio between actual air-fuel ratio and the stoichiometric value, the controller estimate of the manifold pressure, and the internal state of the PI controller. These variables have all been translated so that the equilibrium point coincides with the origin. In the $recovery$ mode, the continuous-time state $\mathbf{x}$ is the tuple $(\hat{p}, \hat{r}, \hat{p}_{est}, \hat{i}, \tau)$. The additional state variable in the $recovery$ mode represents the state of a timer that evolves according to the ODE

$\dot{\tau} = 1$. In the $normal$ mode, the state is given by $(\hat{p}, \hat{r}, \hat{p}_{est}, \hat{i})$.

We assume the system is within 1.0% of the nominal value at the initialization of the $recovery$ mode. This represents the case where the system was previously in a mode of operation that accurately regulated the A/F ratio to the desired setpoint. A domain of interest for the state variables is given by $\|\mathbf{x}\|_\infty < 0.2$.

The verification goal is to ensure that in the given experimental setting, the system always remains within 10% of the nominal A/F ratio after a fixed recovery time of 0.8 ms has passed. In other words, we wish to show that the system begins in the $recovery$ mode, with the initial set of continuous states defined by $\texttt{init} = \{\mathbf{x} \mid \|\mathbf{x}\|_\infty < 0.01\}$; the system transitions to the $normal$ mode after at most 8.0 ms; and the system never transitions to the unsafe set, where $|r| > 0.1$, within the domain of interest $\|\mathbf{x}\|_\infty < 0.2$.

Kapinski et al. established a forward invariant set for the $normal$ mode of operation using a barrier certificate formulation [60]. The authors formulated the barrier certificate using simulation-guided techniques to obtain a candidate Lyapunov function V and a number ℓ to propose a barrier function of the form $B(\mathbf{x}) = V(\mathbf{x}) - \ell$. Here, $V(\mathbf{x}) = \mathbf{z}^T P \mathbf{z}$, and $\mathbf{z}$ is a vector of all monomials of degree ≤ 2 of the state variables $\hat{p}$, $\hat{r}$, $\hat{p}_{est}$ and $\hat{i}$. Note that $\mathbf{z}$ thus contains 14 monomials, and P is a $14\text{x}14$ matrix.

We use the set enclosed by the barrier function to formulate the forward invariant cut

$$C \equiv (M = normal) \wedge (B(\mathbf{x}) \leq 0). \tag{8.10}$$

Application of the forward invariant cut inference rule generates three proof obligations that KeYmaera has to discharge.

Obligation 1. $C \rightarrow [\alpha]C$

Note that once we define C, the hybrid programs $\mathtt{m}_1$, $s_{1\mapsto 2}$ can be excised by KeYmaera, as both have the hybrid program $?M = recovery$ as their first item, which is inconsistent with C. Thus, KeYmaera can then focus on proving this obligation only for the programs $\mathtt{m}_2$, $\mathtt{s}_{\{1,2\}\mapsto fail}$ and $\mathtt{m}_{fail}$.

In order to discharge the obligation for the program $\mathtt{m}_2$, we first perform some simple simplifications with KeYmaera that leaves us with the following proof goal:

$$(B(\mathbf{x}) \leq 0) \rightarrow [\{\mathbf{x}' = f(\mathbf{x})\&H\}](B(\mathbf{x}) \leq 0) \wedge (M = normal) \tag{8.11}$$

To discharge (8.11), we can use the barrier certificate rule shown in (8.12) that we have added to KeYmaera's proof calculus.

$$\frac{init \rightarrow B(\mathbf{x}) \leq 0 \quad B(\mathbf{x}) = 0 \rightarrow \frac{\partial B}{\partial \mathbf{x}} \cdot f(\mathbf{x}) < 0 \quad B(\mathbf{x}) \leq 0 \rightarrow safe}{init \rightarrow [\{x' = f(\mathbf{x})\}]safe} \tag{8.12}$$

In our application of the barrier certificate rule, we substitute $init$ with $(B(\mathbf{x}) \leq 0)$ and $safe$ with $(B(\mathbf{x}) \leq 0) \wedge (M = normal)$. The first and the third proof obligations in the barrier certificate rule are then trivially satisfied. For the remaining (middle) proof obligation KeYmaera uses the SMT solver dReal [39]. To do this, the user must select dReal as the arithmetic solver in the appropriate KeYmaera drop-down menu. Then, KeYmaera asks dReal if the query $(B(\mathbf{x}) = 0) \wedge (\frac{\partial B}{\partial \mathbf{x}} \cdot f_{normal}(\mathbf{x}) > -\epsilon)$ is unsatisfiable, where ϵ is a small positive number.

In order to discharge the proof obligation for $\mathtt{m}_2$, $\mathtt{s}_{\{1,2\}\mapsto fail}$, KeYmaera needs to show that if $B(\mathbf{x}) < 0$ holds, either of these programs cannot invalidate C by transitioning to mode $fail$. It proves this by showing that the set $B(\mathbf{x}) < 0$ is a subset of the safe set using dReal.

Obligation 2. $C \rightarrow S$

This obligation is trivial as S requires the mode to be $fail$, while C says that the mode is $normal$ mode.

Obligation 3. $I \wedge \neg C \rightarrow [(\alpha; ?\neg C)^*]S$

To prove this obligation, we use the lemma that the set $C1$ is an invariant for all states remaining in $I \wedge \neg C$. This is a bounded-time invariant certificate.

$$C1 \equiv (M \neq \mathtt{m}_{fail}) \wedge (0 \leq \tau \leq 0.008) \wedge (x \in S_{reach}) \tag{8.13}$$

Here S_{reach} is an overapproximation of reachable sets by using upper and lower bounds on $\dot{p}$ and $\dot{r}$ computed using dReal. The proof for this branch continues using standard KeYmaera deduction procedures. There is one additional barrier certificate application to show that the normal mode, when starting from this set, lands within the barrier certificate and therefore also respects this invariant. This requires a derivative negativity argument, which KeYmaera again handles via an external dReal query.

8.5 Implementation: The Manticore Preprocessor

An important challenge in the development of deductive verification techniques is to improve automation. To address this, we have developed a software tool to assist with the application of the techniques we have described.

Although a number of techniques can be harnessed to generate forward invariant candidates,

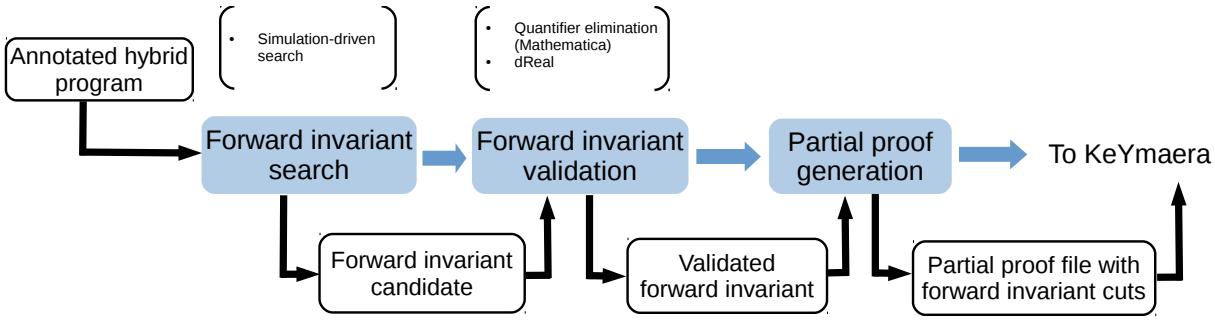


Figure 8.5: Forward invariant proof assistant.

KeYmaera does not currently leverage them automatically. We have developed a tool to (1) search for forward invariants, and (2) generate partial proof files with forward invariant cut suggestions. As a result, this tool, when used in conjunction with KeYmaera, reduces user effort.

We have already written a software tool that takes as input a system specification as a hybrid program representation of a hybrid automaton. The automaton is modeled by writing a hybrid program for each transition and a hybrid program for each mode, as discussed in [90].

In the input file, the user specifies modes that are believed to be stable. The tool (Fig. 8.5) parses this input file and uses simulation driven search for a Lyapunov function in the specified mode, and a sublevelset that excludes the outgoing transition of the mode in question. If the tool succeeds, it writes a partial proof file for KeYmaera, using forward invariant cuts to make use of the forward invariants that have been computed. Then, KeYmaera may be used to load the partial proof file and complete the verification procedure. As a result, the degree of automation of the process is greatly improved.

8.6 Summary

In light of the sparse but discouraging results on construction of a cut-free proof calculus for hybrid systems, techniques that allow proposing useful lemmas for proof task simplification are of special importance.

In this chapter, we presented the forward invariant cut, a proof rule to simplify proof tasks by leveraging designer insights. The forward invariant cut uses insights such as stability of certain regions of state space to simplify the proof task. The strength of the approach is heightened when used in conjunction with the simulation-driven invariant search techniques of Kapinski et al. [60].

Furthermore, the forward invariant cut is an elegant and useful complement to the differential cut already existing in the d$\mathcal{L}$ calculus. The differential cut rule is applied at the leaves of a hybrid verification task, after a great deal of simplification has been performed. Differential cuts are initialized and invariant, giving them a forward inductive flavor. Forward invariant cuts are applied at the start of the proof task to simplify the query and need not be applied to purely continuous programs. Forward invariant cuts are safe and invariant, so in a sense they can be seen to work backwards from the safe set in a backward inductive fashion.

We illustrated our approach on three examples, including an engine fuel control example from the literature. We also discussed our preprocessor tool Manticore, to help with automation concerns.

Table 8.1: Model Parameters for the Engine Fuel Control System.

Parameter	Value
c_1	0.41328
c_2	200.0
c_3	-0.366
c_4	0.08979
c_5	-0.0337
c_6	0.0001
c_7	2.821
c_8	-0.05231
c_9	0.10299
c_{10}	-0.00063
c_{11}	1.0
c_{12}	14.7
c_{13}	0.9
c_{14}	0.4
c_{15}	0.4
c_{16}	1.0
$\hat{u_1}$	23.0829

$$\text{EFC} \equiv I \rightarrow [\left(\mathtt{m}_1 \cup \mathtt{m}_2 \cup \mathtt{s}_{1\mapsto 2} \cup \mathtt{s}_{\{1,2\}\mapsto fail} \cup \mathtt{m}_{fail}\right)^*]S$$

$$I \equiv (-0.001 \leq p \leq 0.001)$$

$$\wedge\, (-0.001 \leq r \leq 0.001) \wedge$$

$$\wedge\, (p_{est} = 0 \wedge i = 0 \wedge M = 1) \wedge$$

$$\mathtt{m}_1 \equiv (?M = \mathit{recovery}; ?\tau \leq 0.008;$$

$$\{\exists \ell_1.\exists \ell_2.\exists \ell_3$$

$$(-0.86 \leq \ell_1 \leq 0.74)$$

$$(-0.17 \leq \ell_2 \leq 0.18)$$

$$(-0.81 \leq \ell_3 \leq 0.68)$$

$$\wedge\, (p' = \ell_1)\ \wedge\ (r' = \ell_2)\ \wedge\ (p'_{est} = \ell_3)$$

$$i' = 0 \& \tau' = 1\ \wedge\ \tau \leq 0.008$$

$$\mathtt{s}_{1\mapsto 2} \equiv (?M = \mathit{recovery}; ?\tau \geq 0.008;$$

$$M := \mathit{normal};)$$

$$\mathtt{m}_2 \equiv (?M = \mathit{normal};$$

$$\{p' = f_p$$

$$r' = f_r$$

$$p'_{est} = f_{p_{est}}$$

$$i' = f_i$$

$$\& -0.02 \leq p \leq 0.02\ \wedge\ -0.02 \leq r \leq 0.02$$

$$\& -0.02 \leq p_{est} \leq 0.02\ \wedge\ -0.02 \leq i \leq 0.02\}$$

$$\mathtt{s}_{\{1,2\}\mapsto fail} \equiv (?(r < -0.1 \vee r > 0.1);$$

$$M := fail)$$

$$\mathtt{m}_{fail} \equiv (?(r < -0.1 \vee r > 0.1);$$

$$M := fail)$$

$$S \equiv M \neq fail$$

Figure 2: A d$\mathcal{L}$ model of a closed-loop fuel control system

Chapter 9

Conclusions

Most traditional techniques for controller development focus on stability and performance specifications such as rate of convergence and magnitude of overshoot. Modern cyberphysical systems have requirements that go beyond these traditional concerns, such as safety and liveness.

A number of new techniques that have been proposed within the research community to attack these new issues. Some focus squarely on verification after a controller has been designed, without providing much guidance for the design process. Others provide full synthesis, but do not leave much room for design paradigms that require flexibility.

Control envelopes provide a reusable abstraction that can be used to validate or reject controller designs without the costly techniques associated with the verification of closed-loop models. Also, they are flexible and can be combined with existing control engineering techniques. In a sense, control envelopes can be used to retrofit existing control design techniques to address modern correctness requirements.

9.1 Outlook

Developments in the fields of logic and automated reasoning have produced rich theoretical insights and powerful computational procedures which have yet to fully make an impact in engineering practice as a whole, and control engineering in particular.

However, this is quickly changing. The work we have presented is part of an exciting movement to provide automatic tools for system engineering.

The intricacy of modern cyberphysical systems implies that it is no longer possible to rely on system testing and designer intuition to vet the correctness of system designs. There is a growing need for automatic reasoning tools that can aid the designer in ensuring formal correctness properties throughout the development lifecycle.

To get a glimpse of what is possible in systems engineering with the use of automatic reasoning tools, we can look to the impact of compilers in software development. A compiler takes, as input, a system specification in a formal language, and automatically synthesizes detailed code in a lower level language.

1. The pace of software development has greatly increased. The automation afforded by compilers has made possible the development of enormously complex software systems in relatively short periods of time.
2. The barrier to entry in computer programming has been significantly lowered. While previously requiring large amounts of specialized training, computer programming (in certain domains) can now be learned in a relatively short time period. As a result, the software sector has exploded, especially in the platforms in which programming is easiest, such as

internet languages.

We foresee that development of formal methods and automatic reasoning tools for control engineering will have similarly transformative effects, as the ability to manage complexity increases. Systems with rich features will be developed faster and more correctly than ever before, and the market will soon be awash with interesting systems developed by freeelancers, tinkerers, and small businesses.

9.2 Directions for future work

This section provides a summary of promising directions for future investigation.

1. **Control envelopes for liveness-style reachability specifications.** Apart from safety, another important correctness specification for control systems is that of reaching a set of target states in finite time. We have given some thought to a framework for using control envelopes for these reachability specifications, and the preliminary results look promising. Instead of using invariants, which are sets that do not allow behaviors to escape, we reason about *attractants*, sets which all behaviors reach in finite time. Then a controller meets the reachability specification if it satisfies the envelope over the entire state space, or at least over an initialized invariant. Sources of attractants exist in the literature, such as the differential variants described in [90]. Sublevelsets of Lyapunov functions are attractants (except for the zero sublevelset). The equilibrium point is not an attractant in general, except for special cases such as deadbeat control [100]. Attractants can also be constructed from barrier certificates by adding time as a state variable.

2. **Automatic generation of control envelopes.** We have an ongoing collaboration with the Model-Based Development group at Toyota Technical Center to automatically produce control envelopes for continuously controlled systems. In a later stage, we hope to extend this framework to discretely controlled systems. The technique considers input-affine systems. First, the simulation-driven technique of [60] is used to compute a near-barrier certificate for the autonomous portion of the dynamics (i.e., the dynamics of the system when the control is zero). This near-barrier certificate should separate the initial and unsafe sets, and its Lie derivative with respect to the autonomous dynamics at the zero levelset should be small. The zero sublevelset constitutes our invariant candidate, and the set of control values that render the Lie derivative with respect to the controlled dynamics negative is our control envelope. By adding time as a state variable, this approach can also be used to construct attractants.

3. **Automatic control synthesis.** We have provided a mechanical procedure for controller verification by refinement checking and refinement checking by parts, but there is still not a satisfactory technique for using control envelopes for automatic synthesis. It is possible to apply quantifier elimination to the refinement conditions with a parametric controller template to obtain constraints on parameters that can later be used in an optimization problem. However, this approach is only feasible when the logical formulas are small enough. On the one hand, we are exploring the possibility of formulating an optimization problem with the refinement conditions as feasibility constraints. However, it is not clear how to guide the gradient descent procedure in a feasible direction with the quantified constraints, other than querying to see if a candidate direction is feasible as part of a logic solver query.

Another possible approach is to leverage the simplex architecture [103] to first design a (simple) safety controller, and then design a performance controller. The control envelope would then be used as an online monitor to determine which controller to use.

Bibliography

[1] Cooperative intersection collision avoidance systems-stop sign assist (cicas-ssa). http://www.dot.state.mn.us/guidestar/2006_2010/cicas.html.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, ThaiSon Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010. ISSN 1433-2779. doi: 10.1007/s10009-010-0145-y. URL `http://dx.doi.org/10.1007/s10009-010-0145-y`.

[3] M. Aga and A. Okada. Analysis of vehicle stability control's effectieffect from accident data. In *ESV Conference*, 2003.

[4] M. Althoff and J. M. Dolan. Set-based computation of vehicle behaviors for the online verification of autonomous vehicles. In *14th International IEEE Conference IEEE Conference on Intelligent Transportation Systems*, 2011.

[5] M. Althoff, O. Stursberg, and M. Buss. Safety assessment of autonomous cars using verification techniques. In *American Control Conference, 2007. ACC '07*, 2007.

[6] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, 2002.

[7] K. Appel and W. Haken. Every planar map is four colorable part i. discharging. *Illinois Journal of Mathematics*, 1977.

[8] K. Appel, W. Haken, and J. Koch. Every planar map is four colorable part ii. reducibility. *Illinois Journal of Mathematics*, 1977.

[9] N. Aréchiga. URL `https://github.com/narechiga/KeYmaera-finvcut.git`.

[10] N. Aréchiga and B. H. Krogh. Using verified control envelopes for safe controller design. In *To be presented at American Control Conference*, June 2014.

[11] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II, Lecture Notes in Computer Science 999*, pages 1–20. Springer, October 1995.

[12] Ralph-Johan Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, bo Akademi, Department of Computer Science, Helsinki, Finland, 1978. Report A–1978–4.

[13] A. Balkan, J. Deshmukh, J. Kapinski, and P. Tabuada. Simulation-guided contraction

analysis. In *Indian Control Conference*, 2015.

[14] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[15] F. Blanchini. Ultimate boundedness control for uncertain discrete-time systems via set-induced lyapunov functions. *IEEE Transactions on Automatic Control*, 1994.

[16] BMW. BMW driving assistant. http://www.bmw.com/com/en/newvehicles/x/x3/2014/showroom/driver_assistance/driving_assistant.html, 2004. Accessed on April 18th, 2015.

[17] BMW. BMW park assistant. http://www.bmw.com/com/en/newvehicles/5series/sedan/2013/showroom/driver_assistance/park_assistant.html, 2013. Accessed on April 18th, 2015.

[18] Michael S Branicky. Stability of switched and hybrid systems. In *Decision and Control, 1994., Proceedings of the 33rd IEEE Conference on*, volume 4, pages 3498–3503. IEEE, 1994.

[19] J. Broughton and C. Baughan. The effectiveness of antilock braking systems in reducing accidents in great britain. *Accident Analysis & Prevention*, 2002.

[20] M. Buehler, K. Iagnemma, and S. Singh, editors. *The DARPA Urban Challenge*. Springer, 2009.

[21] D. Burton, A. Delaney, S. Newstead, D. Loganm, and B. Fildes. Evaluation of anti-lock braking system effectiveness. Technical report, Royal Automotive Club of Victoria, 2004.

[22] Cadillac. Ats luxury sedan how-to videos. http://www.cadillac.com/ats-luxury-sedan/how-to-videos.html. Accessed on April 18th 2015.

[23] P. Cegielski. Théorie élémentaire de la multiplication des entiers naturels. *Model Theory and Arithmetic*, 1981.

[24] J. Chang, L. Cohen, D.and Blincoe, R. Subramanian, and L. Lombardo. Cicas-v research on comprehensive costs of intersection crashes. In *Proceedings of the 20th International Technical Conference on the En- hanced Safety of Vehicles*, 2007.

[25] E. M. Clarke and E. A. Emerson. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming*, 1980.

[26] A. Colmerauer and P. Roussel. The birth of prolog. In *ACM SIGPLAN Notices*, 1993.

[27] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, 1977.

[28] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Sym. Logic*, 3:269–285, 1957.

[29] W. Damm, H. Dierks, J. Oehlerking, and Amir Pnueli. Towards component based design of hybrid systems: Safety and stability. *Time for Verification, Lecture Notes in Computer Science*, June 2010.

[30] James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential.

Journal of Symbolic Computation, 5(12):29 – 35, 1988. ISSN 0747-7171. doi: http://dx.doi.org/10.1016/S0747-7171(88)80004-X.

[31] D. A. de Lima and G. A. Silva-Pereira. Navigation of an autonomous car using vector fields and the dynamic window approach. *Journal of Control, Automation, and Electrical Systems*, 2013.

[32] G. De-Nicolao, L. Magnani, L. Magni, and R. Scattolini. A stabilitizing receding horizon controller for nonlinear discrete time systems. In *American Control Conference*, 2000.

[33] E. 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. URL `http://doi.acm.org/10.1145/360933.360975`.

[34] J. Ding and C. J. Tomlin. Robust reach-avoid controller synthesis for switched nonlinear systems. In *IEEE Conf. on Decision and Control*, December 2010.

[35] Fiat. Fiat city brake control. `h`ttp://www.euroncap.com/en/ratings-rewards/euro-ncap-advanced-rewards/2013-fiat-city-brake-control/, 2013. Accessed on April 18th, 2015.

[36] M. J. Fischer and M. O. Rabin. Super-exponential complexity of presburger arithmetic. Technical report, Massachussetts Institute of Technology, 1974.

[37] S. Gao, J. Avigad, and E. Clarke. Delta-complete decision procedures for satisfiability over the reals. In *Logic in Computer Science*, 2012.

[38] S. Gao, S. Kong, and E. Clarke. dreal: An smt solver for nonlinear theories of the reals (tool paper). In *Conference on Automated Deduction*, 2013.

[39] S. Gao, S. Kong, and E. M. Clarke. dReal: An SMT solver for nonlinear theories of the reals. In *Int. Conf. on Automated Deduction*, June 2013.

[40] G. Gentzen. Studies of Logical Reasoning I,II (in German). Mathematische Zeitschrift 39(2), 39(3): 176–210, 405–431, 1935.

[41] K. Ghorbal and A. Platzer. Characterizing algebraic invariants by differential radical invariants. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, April 2014.

[42] A. Girard. Controller synthesis for safety and reachability via approximate bisimulation. *Automatica*, 48(5):947–953, August 2012.

[43] A. Girard. Low-complexity quantized switching controllers using approximate bisimulation. *Nonlinear Analysis: Hybrid Systems*, 10:34–44, November 2013.

[44] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte fr Mathematik*, 1931.

[45] D. Ghring, D. Latotzky, M. Wang, and R. Rojas. Semi-autonomous car control using brain computer interfaces. In *Intelligent Autonomous Systems 12*. Springer Berlin Heidelberg, 2013.

[46] T. Hales, M. Adams, G. Bauer, D. Tat Dang, J. Harrison, T. Le Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. Tat Nguyen, T. Quang Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. Hoai Thi Ta, T. N. Tran, D. Thi Trieu, J. Urban, K. Khac Vu, and

R. Zumkeller. A formal proof of the Kepler conjecture. *ArXiv e-prints*, January 2015.

[47] T. C. Hales. A computer verification of the kepler conjecture. In *International Congress of Mathematicians*, 2002.

[48] T. C. Hales. A proof of the kepler conjecture. *Annals of Mathematics*, 2005.

[49] T. C. Hales. Introduction to the flyspeck project. In *Dagstuhl Seminar on Mathematics, Algorithms, and Proofs*, 2006.

[50] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Foundations of Computing Series, 2000.

[51] D. Hilbert and W. Ackermann. *Grundzge der theoretischen Logik*. Springer-Verlag, 1928.

[52] C. A. R. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM*, 1969.

[53] C. Holden. Stacking up the evidence. *Science*, 2003.

[54] M. E. Holden. Low-cost autonomous vehicles using just gps. In *Americna Society for Engineering Education Annual Conference and Exposition*, 2004.

[55] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, 2004.

[56] Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. Powertrain Control Verification Benchmark. In *Hybrid Systems: Computation and Control*, 2014.

[57] M. Johansson. *Piecewise linear control systems*. Springer, 2003.

[58] D. J. Kahane. Preliminary evaluation of the effectiveness of antilock brake systems for passenger cars. Technical report, National Highway Traffic Safety Administration, 1994.

[59] J. Kapinski and J. Deshmukh. Discovering forward invariant sets for nonlinear dynamical systems. In *Applied Mathematics, Modeling, and Computational Science*, 2013.

[60] J. Kapinski, J. Deshmukh, S. Sankaranarayanan, and N. Aréchiga. Simulation-guided Lyapunov analysis for hybrid dynamical systems. In *Int. Conf. on Hybrid Systems: Computation and Control*, April 2014.

[61] Eric C. Kerrigan. Robust constraint satisfaction: Invariant sets and predictive control. Technical report, 2000.

[62] H. K. Khalil. *Nonlinear Systems*. Prentice Hall, 2002.

[63] T. S. Kim, J. C. Na, and K. J. Kim. Optimization of an autonomous car controller using a self-adaptive evolutionary strategy. *International Journal of Advanced Robotic Systems*, 2012.

[64] J. Levinson, J. Askeland, J. Dolson, and S. Thrun. Traffic light mapping, localization, and state detection for autonomous vehicles. In *Robotics and Automation (ICRA)*, 2011.

[65] T.S. Li, S. J. Chang, and Y. X. Chen. Implementation of human-like driving skills by autonomous fuzzy behavior control on an fpga-based car-like mobile robot. *Industrial Electronics, IEEE Transactions on*, 2003.

[66] A. Lie, C. Tingvall, M. Krafft, and A. Kullgren. The effectiveness of esc (electronic

stability control) in reducing real life crashes and injureis. *Traffic injury prevention*, 2006.

[67] S. Lie. On differential invariants (in German). *Teubner*, 6, 1884.

[68] S. M. Loos and A. 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.

[69] S. M. Loos, A. Platzer, and L. 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.

[70] S. M. Loos, David Witmer, Peter Steenkiste, and A. Platzer. Efficiency analysis of formally verified adaptive cruise controllers. In Andreas Hegyi and Bart De Schutter, editors, *ITSC*, pages 1565–1570, 2013. ISBN 978-1-4799-2914-613. doi: 10.1109/ITSC.2013.6728453.

[71] A. M. Lyapunov. On the stability of ellipsoidal figures of equilibrium of a rotating fluid (in Russian). 1884.

[72] A.M. Lyapunov. General problem of the stability of motion (in Russian). 1892.

[73] Maurice Mashaal. *Bourbaki: A secret society of mathematicians*. American Mathematical Society, 2002.

[74] F. Maurelli, D. Droeschel, T. Wisspeintner, S. May, and H. Surmann. A 3d laser scanner system for autonomous vehicle navigation. In *Advanced Robotics, 2009. ICAR 2009. International Conference on*, 2009.

[75] D. Q. Mayne, J. B. Rawlings, C.V. Rao, and P. O. M. Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 2000.

[76] Kenneth L McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, pages 123–136, 2006.

[77] D. A. Miller, E. Longini-Cohen, and P. B. Andrews. A look at tps. In *6th Conference on Automated Deduction*, 1982.

[78] Stefan Mitsch, Jan-David Quesel, and André Platzer. Refactoring, refinement, and reasoning: A logical characterization for hybrid systems. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM*, volume 8442, pages 481–496. Springer, 2014. doi: 10.1007/978-3-319-06410-9_33.

[79] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[80] J. Oehlerking and O. E. Theel. Decompositional construction of lyapunov functions for hybrid systems. In R. Majumdar and P. Tabuada, editors, *Int. Conf. on Hybrid Systems: Computation and Control*, volume 5469 of *Lecture Notes in Computer Science*, pages 276–290. Springer, April 2009.

[81] US Department of Transportation. Cooperative intersection collision avoidance systems. http://www.its.dot.gov/cicas.

[82] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak

Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. URL `http://www.csl.sri.com/papers/cade92-pvs/`.

[83] A. Papachristodoulou and S. Prajna. On the construction of Lyapunov functions using the sum of squares decomposition. In *IEEE Conf. on Decision and Control*, December 2002.

[84] P. A. Parrilo. *Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization*. PhD thesis, California Institute of Technology, 2000.

[85] P. A. Parrilo. Lectures on algebraic techniques and semidefinite optimization: Quantifier elimination. http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-972-algebraic-techniques-and-semidefinite-optimization-spring-2006/lecture-notes/lecture_18.pdf, April 2006. Accessed on April 26th, 2015.

[86] A. Platzer. Differential Dynamic Logic for Verifying Parametric Hybrid Systems. Technical Report 15, Reports of SFB/TR 14 AVACS, May 2007. ISSN: 1860-9821, http://www.avacs.org.

[87] A. Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, August 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9103-8.

[88] A. Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *J. Log. Comput.*, 20(1):309–352, February 2010. doi: 10.1093/logcom/exn070. Advance Access published on November 18, 2008.

[89] A. Platzer. Differential-algebraic Dynamic Logic for Differential-algebraic Programs. *J. Log. Comput.*, 20(1):309–352, 2010. doi: 10.1093/logcom/exn070. Advance Access published on November 18, 2008.

[90] A. Platzer. *Logical Analysis of Hybrid Systems*. Springer, 2010.

[91] A. Platzer. A differential operator approach to equational differential invariants. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 28–48. Springer, August 2012.

[92] A. Platzer. The structure of differential invariants and differential cut elimination. *Logical Methods in Computer Science*, 8(4):1–38, 2012. doi: 10.2168/LMCS-8(4:16)2012.

[93] A. Platzer and E. M. Clarke. Computing differential invariants of hybrid systems as fixedpoints. In *Computer Aided Verification*, July 2008.

[94] A. Platzer and J. D. Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Int. Joint Conf. on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer, August 2008. ISBN 978-3-540-71069-1. doi: 10.1007/978-3-540-71070-7_15.

[95] S. Prajna. *Optimization-based methods for nonlinear and hybrid systems verification*. PhD thesis, California Institute of Technology, Caltech, Pasadena, CA, USA, 2005.

[96] S. Prajna. Barrier certificates for nonlinear model validation. *Automatica*, 42(1):117–126, 2006.

[97] S. Prajna and A. Jadbabaie. Safety Verification of Hybrid Systems Using Barrier Certifi-

cates. In *Hybrid Systems: Computation and Control*, pages 477–492, March 2004.

[98] M. Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du I congrs de Mathmaticiens des Pays Slaves*, 1929.

[99] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, 1982.

[100] K. J. Åström and B. Wittenmark. *Computer Controlled Systems: Theory and Design*. Prentice Hall, 1997.

[101] Matthias Rungger, Manuel Mazo, Jr., and Paulo Tabuada. Specification-guided controller synthesis for linear systems and safe linear-time temporal logic. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, pages 333–342, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1567-8. doi: 10.1145/2461328.2461378. URL http://doi.acm.org/10.1145/2461328.2461378.

[102] A. Seidenberg. A new decision method for elementary algebra. *Annals of Mathematics*, 1954.

[103] D. Seto, B. Krogh, L. Sha, and A. Chutinan. The simplex architecture for safe online control system upgrades. In *American Control Conference, 1998. Proceedings of the 1998*, volume 6, pages 3504–3508 vol.6, Jun 1998. doi: 10.1109/ACC.1998.703255.

[104] T. Skolem. Über einige Satzfunktionen in der arithmetik. In *Skrifter Vitenskapsakadetiet i Oslo*, volume 1, 1931.

[105] A. Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013. ISSN 1433-2779. doi: 10.1007/s10009-012-0249-7. URL http://dx.doi.org/10.1007/s10009-012-0249-7.

[106] G. Szpiro. Does the proof stack up? *Nature*, 2003.

[107] P. Tabuada. *Verification and Control of Hybrid Systems*. Springer, 2009.

[108] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, 2nd edition, 1951.

[109] Sebastian Thrun. What we're driving at. Google Official Blog, http://googleblog.blogspot.com/2010/10/what-were-driving-at.html, 2010. Accesed on April 18, 2015.

[110] U. Topcu, P. Seiler, and A. Packard. Local stability analysis using simulations and sum-of-squares programming. *Automatica*, 44:2669–2675, 2008.

[111] Toyota. Brake Assist. http://www.toyota-global.com/innovation/safety_technology/safety_technology/technology_file/active/brake.html. Accessed on April 18th, 2015.

[112] Audi USA. 2016 audi a6 earns iihs highest rating of top safety pick+. http://www.audiusa.com/newsroom/news/press-releases/2015/03/2016-audi-a6-earns-iihs-highest-rating-of-top-safety-pick. Accessed on April 18th 2015.

[113] R. Vidal, S. Schaffert, J. Lygeros, and S. Sastry. Controlled invariance of discrete time

systems. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 1999.

[114] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1910.

[115] E. M. Wolff, U. Topcu, and R. M. Murray. Optimization-based control of nonlinear systems with linear temporal logic specifications. Submitted to Int. Conf. on Robotics and Automation, available at http://www.cds.caltech.edu/~murray/papers/wtm14-icra.html, May 2014.

[116] A. Zheng and F. Allgöwer. Towards a practical nonlinear predictive control algorithm with guaranteed stability for large-scale systems. In *Proceedings of the American Control Conference*, 1998.