Algebraic Verification of Hybrid Systems in Isabelle/HOL

University of Sheffield

Jonathan Julian Huerta y Munive

Supervisor: Dr. Georg Struth

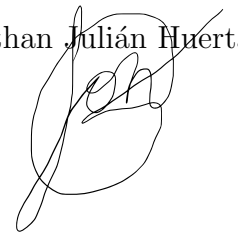
This dissertation is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (PhD) in Advanced Computer Science

Sheffield, UK. October, 2020

Signed Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Jonathan Julián Huerta y Munive

Signature: 

Date: 30 October 2020

Abstract

The thesis describes an open modular semantic framework for the verification of hybrid systems in a general-purpose proof assistant. We follow this approach to create the first algebraic based verification components for hybrid systems in Isabelle/HOL.
The framework benefits from various design choices. Firstly, an algebra for programs such as Kleene algebras with tests or modal Kleene algebras captures the verification condition generation by providing rules for each programming construct. Intermediate relational or state transformer semantics instantiated to a concrete model of the program store allow the framework to handle assignments and ordinary differential equations (ODEs). The verification rules for ODEs require user-provided solutions, differential invariants or analytical descriptions of the continuous dynamics of the system.
The construction is a shallow embedding which makes the approach quickly extensible and modular. Taking advantage of these features, we derive differential Hoare logic ($\mathsf{d}\mathcal{H}$), a minimalistic logic for the verification of hybrid systems, and the differential refinement calculus ($\mathsf{d}\mathcal{R}$) for their stepwise construction. Yet the approach is not limited to these formalisms. We also present a hybrid weakest liberal precondition calculus based on predicate transformers which subsumes powerful deductive verification approaches like differential dynamic logic.
The framework is also compositional: we combine it with lenses to vary the model of the program store. We also support it with a formalisation of affine and linear systems of ordinary differential equations in Isabelle/HOL. This integration simplifies various certifications that the proof assistant requires such as guarantees of existence and uniqueness of the corresponding solutions.
Verification examples illustrate the approach at work. Formalisations of our solutions to problems of the international friendly competition ARCH2020, where our components participated, further evidence their effectiveness. Finally, a larger case study certifying an invariant for a PID controller of the roll angle in a quadcopter's flight complements these verifications.

Acknowledgements

Firstly, I wish to show gratitude to Georg Struth for his above and beyond support and guidance. I know I am a far better academic than I would have been if anyone else were my supervisor. From the beginning, he has been a role model for me due to his ethical integrity and high standards. He is a very comprehensive person who has patiently shared with me his knowledge, expertise and enthusiasm. For this and many more experiences, thank you.

Next, I want to thank my examiners Ana Cavalcanti and Andrei Popescu for their comments on this thesis. I also appreciate the interesting discussion we had during my examination and their enthusiasm with the contributions of my work. Your expert observations have helped me make this project more robust.

I also would like to offer my sincere thanks to various researchers whom I have had interesting discussions during my time as a PhD student. Great individuals like Uli Fahrenberg, Simon Foster, Walter Guttmann, Peter Höfner, Christian Johansen, Achim Jung, Iván Martínez Ruiz, Stefan Mitsch, Olaf Owe, Lawrence Paulson, and André Platzer have influenced my way of thinking with their insightful observations and points of view. Honourable appreciation goes to Achim D. Brucker, Neil Walkinshaw and Joab Winkler for being an excellent PhD panel that not only guided me but also demanded the best of me.

Special thanks also go to the members of the verification reading group in Sheffield. I appreciate a lot the huge efforts that Harsh Beohar, Kirill Bogdanov, Rayna Dimitrova, Michael Foster, Michael Herzberg, Rob Hierons, Raluca-Elena Lefticaru, George O'Brien, and Mike Stannett have taken for us to better understand a small part of the human knowledge. It has been a pleasure to study them together.

I am also grateful to my friends and colleagues in Sheffield. The people from the verification and testing lab provided entertaining conversations every day at lunchtime; the Mexicans added spiciness to the bland weather of the city, and the Taoists supplied long hours of fun pain by asking me to stay still.

I thank my friends and family in Mexico for finding time to nurture our friendship with online video calls and for their unconditional support. Gracias Bren, Davids, Dulce, Esaú, Fer, Gus, Hecsari, Hector, Hugo, Karen, Liz, Omar, Roque y Yasmin. Gracias mamá, gracias papá y gracias hermanos.

Finally, I wish to thank my beautiful wife for growing up with me another three years.

Contents

List of Figures

Chapter 1

Introduction

At least since the First Industrial Revolution, there has been interest in mathematically modelling the interactions of machines with physical phenomena [15, 94]. On one hand, differential equations are the standard representation for the continuous dynamics in nature. On the other, with the advent of computers, our abstractions for digital systems have focused more on discrete transitions and state updates. Hybrid systems combine both types of models to describe the evolution in time of physical events controlled via computers. Nowadays, many forms of hybrid systems abound in the scientific literature [5, 39–41, 51, 61, 66, 72, 86, 108]. In particular, researchers use them to verify the correct behaviour of the systems they intend to represent [3, 24, 27, 48, 50, 77].

Among the many techniques for the verification of hybrid systems, deductive verification stands out. In contrast with other methods [5, 39], the deductive approach usually tackles complex and simple dynamics with the same techniques, making it generic. Another benefit is that, in comparison, it adequately represents the compositional nature of engineered systems [108]. Moreover, after a successful deductive verification, the result is a mathematical proof of the correct behaviour of the hybrid system relative to a specification while other methods focus on guaranteeing lack of failure [17, 108]. A prominent approach in this line of research is differential dynamic logic $\mathsf{d}\mathcal{L}$ [108]. It extends traditional dynamic logic based on regular programs and forward box and diamond operators [58], with differential equations of continuous dynamical systems. Its domain-specific proof assistant KeYmaera X [50] implements it, and multiple case studies back the effectiveness of its methods [50, 77, 81, 89, 96, 116]. Yet, for reasons of decidability, the approach still limits its language for differential equations and their solutions. This forces the logic to provide alternative proof strategies to reason about ODEs [107, 117].

On the other hand, many general-purpose proof assistants (or interactive theorem provers) are at least as expressive as higher-order logic (HOL) [28, 101]. Loosely speaking, this means that their users can formalise well-established mathematics as part of the prover's developments. Specifically, this also applies to embeddings of logics such as $\mathsf{d}\mathcal{L}$ inside them. Furthermore, two important features have improved general-purpose proof assistants considerably [6, 32, 91, 101, 104, 105]: their increased proof automation and the growth of their libraries of formalised mathematics, specifically those for ordinary differential equations. The leading proof assistant with these characteristics is Isabelle/HOL [23, 124], which provides its Sledgehammer tool [95, 106] that calls external automated theorem provers and SAT solvers

to suggest proofs to its users. Accordingly, this has allowed them to formalise various results, many of which are available in Isabelle's reviewed Archive of Formal Proofs (AFP). A fitting example of this is Isabelle's Analysis library [67]. It starts from topology and filters to calculus, limits and derivatives, ending with an entry in the AFP for Ordinary Differential Equations (ODEs) [74, 75]. Thus, Isabelle/HOL can reason about a more general class of ODEs than existing deductive methods for hybrid systems.

Another relevant formalisation in Isabelle/HOL is that of Kleene algebras [9, 10], a model for regular programs. Their extensions in the form of Kleene algebras with tests (KAT) [8] and modal Kleene algebras (MKA) [54, 55] generalise typical deductive verification methods like Hoare logic [65] and dynamic logic [56, 58] respectively. Thus, the operations and rules of inference of these logics can be shallowly embedded in the proof assistant with the algebraic setting. Then, these rules can be instantiated to concrete models of the program store and algebras such as relations or state transformers. The shallow embedding enables faster developments of verification tools as it skips syntactic and proof-theoretic results by directly encoding the semantics. The result of this process consists of verification components for regular and while-programs in the proof assistant [55, 56]. As each step of the construction occurs inside the interactive theorem prover, the implemented tool is correct by construction. Ultimately, with the Isabelle/HOL libraries for KATs and MKAs, users can do software verification.

As a consequence of these observations, we are at a point where integrating the Kleene algebra verification components and the libraries of ODEs seems feasible. Such an integration would allow us to do deductive verification of hybrid systems in a more expressive manner than what current methods offer. In particular, the integration could be used to reason about more general ODEs and their solutions than the state of the art in deductive verification. This expressive potential is only limited by the underlying logic of the proof assistant. Moreover, in contrast to existing tools, such a combination would become an open framework with a potentially wider user-base. The combination of this added expressivity and the extended user-base would make this approach more prone to be extended and include other forms of dynamics not covered by the continuous ones, like stochastic or adversarial dynamics. Furthermore, contributions to the interactive theorem prover, both in terms of improving its automation and in terms of extending its libraries, would reverberate positively in the integration. Both the expressiveness and the openness could lead to the development of more general deductive verification techniques. In the end, the integration of these libraries could be a crucial step in the dissemination of proof assistants in engineering applications. It could open the possibility of integrating deductive verification with other common approaches like model checking, simulation and testing.

Hence, this thesis describes a modular and open semantic framework for hybrid system verification with a general-purpose interactive theorem prover. It details the algebraic foundations, their integration with ODEs, and uses both of them to derive established and innovative verification inference rules. The thesis also explores the modularity of the framework by displaying the consequences of alternating the algebraic foundations, the representation of the program store, and the model for the program algebras. It shows that extending the framework not only benefits the end result but it also contributes to growing the libraries of formalised mathematics. Finally, the thesis contains several verification examples and case studies that exhibit the capabilities and limitations of the current state

of the framework.

1.1 Contributions

The main contribution of this work is twofold. On the one hand, we provide a detailed description of a generic, modular and open platform for the verification of hybrid systems. On the other hand, we give its full formalisation in the general-purpose proof assistant Isabelle/HOL. Figure 1.1 depicts the steps needed in order to implement this verification framework in an interactive theorem prover. Essentially, the formalisation of an *algebra of programs*, like a Kleene algebra or an algebra of predicate transformers, provides the structure to reason soundly about regular programs inside the proof assistant. For instance, the program algebras used in this work already capture the rules for verification condition generation of Hoare logic or the weakest liberal precondition calculus. The instantiation of this algebra to an *intermediate semantics* that models programs, like binary relations, adds expressivity to the framework and it allows us to encode other programming constructs such as assignments. It is at this second stage of development, that we can also integrate the ordinary differential equations, provided that their formalisation is available in the theorem prover. A final instantiation to a concrete *program store model* completes the development as this enables us to access and update program states. Once this is in place, we can use the proof assistant's object logic to *extend* the proving capabilities of the framework. We evidence how to do this in various ways. For instance, we provide specific rules for verification condition generation for assignments and ODEs in both, Hoare-logic style or as weakest liberal preconditions. We have also derived rules of $\mathsf{d}\mathcal{L}$ and provided alternative ways to verify safety properties about hybrid systems that were not available previously. The application of all these verification rules through the proof assistant is what constitutes our *verification components*.

To our knowledge, this is the first development of a shallow and semantic embedding of algebraic based verification components for hybrid systems within a general-purpose interactive theorem prover. Previous works are deep embeddings of concrete logics and have been used to certify the result of external provers [22], or are limited to correctness specifications in the form of Hoare triples [138]. In contrast, our components are open and flexible because they do not adhere to a specific formal system. Potentially, any other development for deductive verification of hybrid systems can be integrated into our framework if it is expressible within the proof assistant's object logic. In the case of our implementation in Isabelle, this is restricted to higher order logic (HOL).

Moreover, as evidence for the relevance of our framework, we mention its use for verification of an autonomous marine vehicle at the University of York [43]. The referred publication discusses the integration of traditional control-development processes, like simulation, with the deductive approach used in our framework. Because of their focus on applications, their work extends a version of our framework with powerful automation techniques for statements involving ODEs. This clearly shows that the platform presented in this thesis is open, modular and extensible.

Below we provide a classification of our concrete technical contributions in four categories.

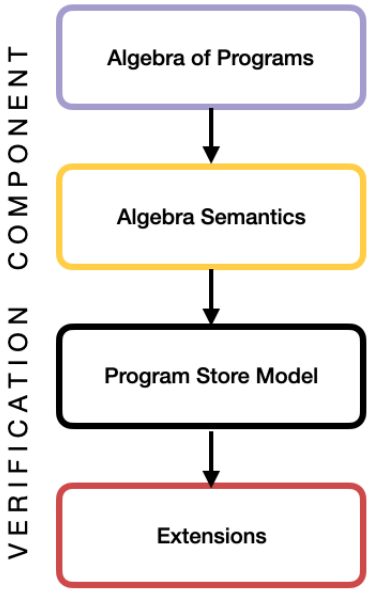


Figure 1.1: Development of the verification framework in a general purpose proof-assistant

Formalisations. Writing libraries of mathematical theorems checked by machines is relevant for the scientific community as it improves rigour, increases confidence in our knowledge and, as this thesis exemplifies, opens a path to new ways of doing research. As part of our work, we have formalised several mathematical results. The simplest of our formalisations are mere instantiation proofs showing that state transformers form KATs and MKAs. These results are available in Section 3.5. Our reformulations of Picard-Lindelöf's theorem using Isabelle's locales from Section 4.2 form another contribution. One of these reformulations includes and added parameter representing the flow of systems of ODEs. It also includes the proof that these flows behave as monoid actions, a relevant result not formalised explicitly before. In that very same section, we also provide a tactic for automatic certifications of derivatives in Isabelle/HOL. Finally, our largest formalisation connects linear algebra and ordinary differential equations to obtain affine and linear systems of ODEs. It includes generic results about existence and uniqueness of solutions for all affine/linear systems, as well as the description of their general solution and methods for simplifying it. Its description and use as a non-trivial extension for the verification components is in Section 6.5.

Conceptual. Mathematically, three cacluli for the verification of hybrid systems emerged from our work: differential Hoare logic $\mathsf{d}\mathcal{H}$, the differential refinement calculus $\mathsf{d}\mathcal{R}$, and the hybrid weakest liberal precondition calculus. They became evident after changing the framework's underlying algebra of programs which exemplifies the value of the framework's modularity. Roughly speaking, these calculi correspond to Kleene algebras with tests, refinement Kleene algebras with tests, and modal Kleene algebras respectively. In particular, the author of this thesis proved the soundness of all inference rules of these calculi and contributed to the derivation of the rules of $\mathsf{d}\mathcal{H}$ and $\mathsf{d}\mathcal{R}$ described in Sections 4.4 and 6.1. He also generalised some mathematical concepts from the dynamical systems literature.

Specifically, he came up with the generalisation of orbits and invariants available in Sections 5.1 and 5.2. Nevertheless, their categorical characterisations and properties are due to his coauthor in [72], Georg Struth. Finally, in Section 5.3, the alternative representation of dynamical systems as a hybrid program using the solution to a system of ODEs instead of the ODE itself is also a contribution from the author of the thesis.

Components. Due to our exploration of the modularity of the framework presented in this thesis, we ended up with various versions of the verification components. Figure 6.1 depicts the different levels of the framework where we can get a new version of the components. By choosing KATs as our algebraic foundation, we obtain $\mathsf{d}\mathcal{H}$ components. These intend to be minimalistic as they provide the least amount of inference rules for verification of hybrid systems. On the other hand, using MKAs or predicate transformers, we obtain a hybrid weakest liberal precondition calculus as in Sections 3.3, 6.2, and 6.3. These verification components provide equational reasoning not available in the $\mathsf{d}\mathcal{H}$ components. We extend KAT with a refinement operation in Section 6.1 and get $\mathsf{d}\mathcal{R}$, hence, a component for refining hybrid programs. By using lenses, we can instantiate various program store models and a new verification component for each of them. For this, see Section 6.4. Based on the semantics of the algebra, we can use the relational or state transformer model to derive the verification rules. Because of the simplicity with which we can alternate algebraic models, each of our previously described components has also a relational and a state transformer implementation. Finally, the most lightweight verification components skip the algebraic semantics and use direct encodings of predicate transformers. These components are faster to implement and easy to automate but they loose part of their formal rigour because of their missing algebraic instantiation.

Verifications. All verifications in this thesis from Chapter 7 and the verification examples in the rest of the thesis are contributions from our work.

1.2 Outline

Since this thesis describes a generic open platform for the development of verification components for hybrid systems in a proof assistant, we support each mathematical description with a corresponding formalisation. Thus, in each chapter we provide sections with Isabelle code that implement the concepts discussed there. Similarly, we accompany the descriptions of the technical concepts with examples. In particular, throughout the thesis we use a running example of a thermostat that intends to keep the temperature of a room within a comfortable range.

Due to the fact that we have formalised our results in an interactive theorem prover, we omit most proofs in the thesis or present them in the corresponding formalisation sections. Also, to match the presentation of Isabelle's lemmas, we adopt its notation for the presentation of mathematics. For instance, for function application we use juxtaposition (with a space) $f\,x$ instead of the conventional $f(x)$ and we use indistinctly the words "proposition", "lemma" and "theorem" as synonym for "formalised result". In our work, the interaction of various areas of mathematics in the creation of the components forces us

to use diverse symbols, variables and terminology. For the most part, we remain consistent with our use of them. Yet, at the end of the thesis, we provide a list of symbols and an index for the interested reader.

Below we describe in more detail the contents of this written work:

- In Chapter 2, we give a more refined explanation of the technical concepts discussed in this introduction. We review basic terminology about hybrid systems and their verification. We then discuss related work. In particular, we briefly explain the most popular model for hybrid systems: hybrid automata. Then, we describe the alternative representation of hybrid programs which is our method of choice in the deductive approach for verification. We culminate by explaining the benefits of using a general-purpose proof assistant like Isabelle/HOL and by giving a brief introduction to the syntax of the prover.

- Chapter 3 describes Kleene algebras and its variants for verification. We explain their axioms and their interpretation as models for programs. In particular, we describe two of their semantics: the canonical relational version and the state transformer semantics. During this process, we take the opportunity to explain the isomorphism between both of them that goes beyond mono-typed instances. We then extend Kleene algebras to obtain KATs and MKAs and derive verification rules through each of them. We describe the Hoare logic generated by KAT and the weakest liberal precondition calculus with predicate transformers obtained with MKA. We end the section with an explanation of a common approach to formalise algebraic structures in Isabelle/HOL. We use it to describe the formalisation of Kleene algebras in the proof assistant as well as the implementation of their semantics.

- Chapter 4 connects the algebraic verification approach with the ordinary differential equations. For this, we take a quick detour at the beginning to recall basic notions about ODEs including Picard-Lindelöf's theorem that provides the conditions to guarantee existence and uniqueness of solutions to ODEs. The specific details in the formalisation of this crucial result have many consequences for the verification components. Therefore, we describe its implementation in Isabelle/HOL. Then, we describe the model for the program states and use this to add assignments and differential equations to the programming constructs obtained with Kleene algebras. It serves us to derive verification rules for both constructs. We end the chapter by detailing the corresponding formalisation of this integration.

- Next, in Chapter 5, we focus on the verification procedures generated by our construction of the components. For this, we generalise the semantics for ODEs introduced in the previous chapter. We use our generalisations to describe alternative deductive verification styles for ODEs with the same components. In particular, we explain how to reason about invariants of systems of ODEs and how to skip certain certifications that the proof assistant requires by providing in the specification the analytic dynamics of the system. We end the chapter with the formalisation of both approaches and the derivation of some rules of inference of $\mathsf{d}\mathcal{L}$ with our components. This allows them to also reason in the style of this logic.

- Once the verification components are built, we discuss their modularity, flexibility and extensibility in Chapter 6. Specifically, extending KATs with a refinement operation gives us a differential refinement calculus $\mathsf{d}\mathcal{R}$ in the style of Moargan [99]. Also, we consider how changing MKAs for predicate transformers, whether à la Back and von Wright [12] via quantales or through the powerset monad [90], generates the same weakest liberal precondition calculus. We further extend the modularity of the components by replacing the program store model with lenses. Finally, we formalise affine systems of ODEs in Isabelle/HOL to extend the capabilities of the components. This allows them to use matrix representations of the systems of ODEs and it simplifies some of the verification procedures discussed in the precedent chapters. We end with a brief explanation of which version of the components to use for different purposes.

- In Chapter 7, we provide various verification examples to illustrate the capabilities and limitations of the components. The first verification is simple enough so that we can tackle it using various procedures from previous chapters in full detail. The second and third examples help us illustrate different ways to use our formalisation of affine systems of ODEs with our verification components. The fourth verification is a canonical example from the literature [4, 108] that tests the component's ability to tackle most programming constructs. We use our $\mathsf{d}\mathcal{R}$ components with lenses to refine the fifth example. Having exemplified in detail the verification approaches, we then discuss some problems of a friendly competition for verification of hybrid systems where we participated. They help us discuss the current limitations of our approach. Finally, we provide an invariant for a hybrid system that models a larger case study: a common controller to regulate the roll angle in a quadcopter's flight.

- We finish the thesis with the conclusions in Chapter 8.

1.3 Publications

The thesis summarises most of my work during three years of my PhD studies. Along that process, my coathors and I have written and published various scientific papers. Given that these publications have served as a guide for the structure and contents of the thesis, I provide below a complete list of them with a discussion of their relation to this written work.

- *J. J. Huerta y Munive and G. Struth. Verifying hybrid systems with modal Kleene algebra. In* RAMiCS 2018*, volume 11194 of* LNCS*, pages 225–243. Springer, 2018* [71]. This publication describes the first implementation of the verification components. It contains a formalisation with a different model for the program store in terms of lists that is not discussed in this thesis.

- *J. J. Huerta y Munive and G. Struth. Predicate transformer semantics for hybrid systems: Verification components for Isabelle/HOL.* arXiv:1909.05618 *[cs.LO], 2019* [72]. We generalise most of the techniques of the first publication in [72] which is under review. Therefore, we prefer the latter article for our explanations here. Specifically, Chapters 3, 4, and Sections 6.2 and 6.3 of Chapter 6 describe in more

detail the corresponding parts of [72]. Although Chapter 5 also shares content with [72], it offers a generalisation of the concepts discussed there. Similarly, Sections 7.1 and 7.4 extend the verification examples of [72] by providing alternative methods for proving them.

- *J. J. Huerta y Munive. Verification components for hybrid systems.* Archive of Formal Proofs, *2019* [68]. Our formalisation of all the ideas in [71] and [72] is available in the Archive of Formal Proofs in [68]. Parts of the formalisation of [68] are available and explained throughout all chapters.

- *S. Foster, J. J. Huerta y Munive, and G. Struth. Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In* RAMiCS 2020[postponed]*, pages 169–186, 2020* [44]. Our work deriving differential Hoare logic $\mathtt{d}\mathcal{H}$ and the differential refinement calculus is published in [44]. Thus, it shares contents with our Sections 3.2 and 6.1. The running example of the thermostat and the example of Section 7.5 also appear in [44], although in this thesis we extend them.

- *J. J. Huerta y Munive. Affine systems of ODEs in Isabelle/HOL for hybrid-program verification. In* SEFM 2020*, volume 12310 of* LNCS*, pages 77–92. Springer, 2020* [69]. In this publication, we find the discussion of affine systems of ODEs in Isabelle/HOL. This is the only publication from which paragraphs are taken verbatim for the thesis as I am its sole author. The paragraphs are only used in Section 6.5. The examples of Sections 7.2 and 7.3 are also from this publication although we expand our explanations for them and do not copy their contents from [69]

- *J. J. Huerta y Munive. Matrices for ODEs.* Archive of Formal Proofs, *2020* [70]. The formalisation of affine systems of ODEs corresponding to [69] is available in [70], which is also an entry to the AFP.

- *S. Mitsch, J. J. Huerta y Munive, X. Jin, B. Zhan, S. Wang, and N. Zhan. ARCH-COMP20 category report: Hybrid systems theorem proving. In* ARCH20.*, pages 141–161, 2019* [97]. The examples of Section 7.6 are part of a collection of verification benchmarks for the friendly competition of the 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH2020). We contributed to the report [97] that explains the results of the competition in the Hybrid Systems Theorem Proving category.

Readers can find all other formalisations discussed in this thesis that are not available in [68] and [70] in the online repository https://github.com/yonoteam/CPSVerification. Shortly, we intend to add the ARCH2020 competition examples and the case study for the verification of the PID controller to the Archive of Formal Proofs.

Chapter 2

Related Work

This chapter contextualises our work within the existing body of research on verification of hybrid systems. For this reason, we start with a high-level definition of hybrid systems and their safety verification problem in Section 2.1. Here we also describe our assumptions on a running example in the thesis: a digital thermostat. After that, we discuss the techniques used to tackle that problem starting with the methods employed for formal analysis of software: model checking (Section 2.2) and deductive verification (Section 2.3). Later on, we focus on hybrid systems, in particular we discuss reachability analysis through hybrid automata in Section 2.4. We proceed to describe deductive verification and their model for hybrid systems, hybrid programs, in Section 2.5. We also discuss related work in Section 2.6 on doing deductive verification and formalising differential equations in interactive theorem provers. Finally, in Section 2.7, we describe the basic syntax of Isabelle/HOL, our proof assistant of choice to implement our verification framework.

2.1 Hybrid Systems Verification

Hybrid systems serve as mathematical representations for digital controllers interacting with physical phenomena. For this reason, they need to combine standard models for discrete behaviour (automata, n-ary relations, ...) and continuous dynamics (differential equations and inequalities). Figure 2.1, for instance, provides a schematic of a thermostat interacting with the temperature in a room. The discrete part corresponds to the program that the thermostat uses in order to decide whether to turn the heater on or off. A differential equation provides the mathematical representation of the temperature's continuous behaviour.

Example 2.1.1. Throughout the thesis up to Chapter 6, we use the thermostat as a running example of a hybrid system that helps us illustrate the concepts presented in our work. This is why we state our assumptions of the system here. Variable θ represents whether the heater is turned on, $\theta = 1$, or off $\theta = 0$. We use variable T to represent the room's temperature. The objective of the thermostat is to keep T within a comfortable region $T_m \leq T \leq T_M$, where $0 \leq T_m, T_M \leq T_L$ and T_L is the room's temperature attained when the heater radiates maximally. The differential equation $T' = -a \cdot (T - T_\theta)$ models a simplified behaviour of T. The constant $a > 0$ indicates how fast the temperature changes. If the heater is off, $T_\theta = 0$ and the temperature decreases. If it is on, it increases with $T_\theta = T_L$. For illustration

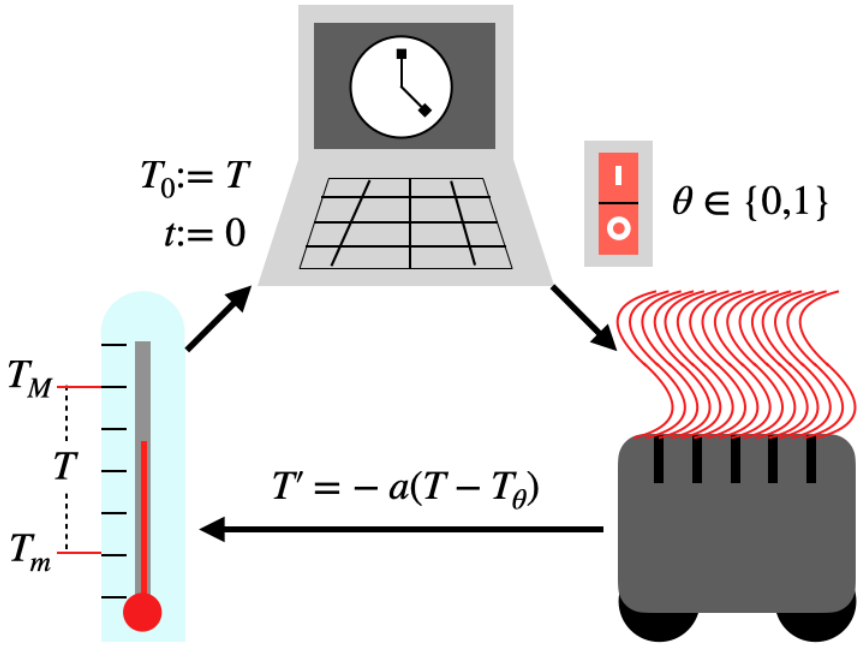


Figure 2.1: Schematic representation of the behaviour of a thermostat

of the concepts presented in the thesis, we also assume that the thermostat is digital and it includes a chronometer that helps it intervene at the "right moment". Thus, we use variable t to store elapsed time and model its passage with the differential equation $t' = 1$. We feel free to overload t because the solution to this ODE with initial condition $t\,0 = 0$ is the identity function, that is, the value of the function t after τ units of time is τ. When the thermostat's discrete control intervenes, it resets the chronometer $t := 0$, "measures" the temperature $T_0 := T$, and decides whether to turn on or off the heater. As its customary in the hybrid systems literature, we assume that the control intervention is instantaneous. This is a safe assumption if the duration of the control's intervention is considerably faster than the unit of time used to measure the length of the physical process it regulates. □

The integration of continuous and discrete models into a single hybrid system allows controller designers to do a thorough analysis through precise mathematical semantics. Typically, the main body of the analysis consists of a collection of proofs showing that the hybrid system X satisfies a given specification which itself is often stated as a *safety verification problem*. Such a problem requires showing that every execution of X remains safe. That is, a precondition P describes the set of initial states of the system and a postcondition Q indicates the states where X would be safe. Formal analysts implement the specifications through a formal system that extends a classical logic. They can carry out the proofs at the level of the formal system's proof calculus or with its semantics.

In the literature, two predominant kinds of hybrid systems are hybrid automata and hybrid programs. They roughly correspond to two approaches for tackling the safety verification problem: reachability analysis and deductive verification. These also are related to other two methods for doing software verification: model checking and deductive

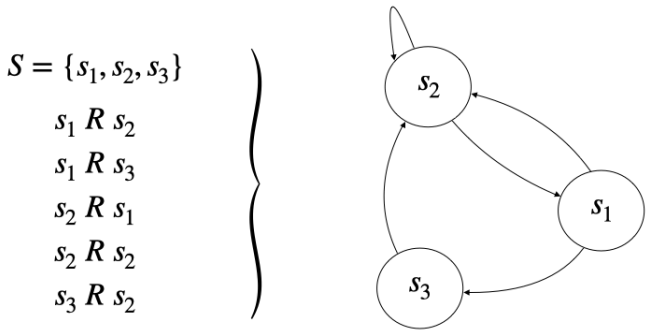


Figure 2.2: A system represented with a finite set of states S and a relation R

verification. In the following sections, we describe these terms in more detail and discuss their respective merits.

2.2 Model Checking

A common mathematical representation of finite-state systems is via a relation R on a finite set of states S. Graphically, we can depict such a pair as a collection of nodes connected by edges (see Figure 2.2). The benefit of this representation is that computer scientists can write algorithms that exhaust the state space by traversing all the possible paths that the system may follow. This also helps to check if a postcondition is true at each state or in generating a trace where it is not. Clarke and Emerson introduced the term *model checker* [29] for their algorithm that decided the satisfiability of specifications via this process. Their work allowed automated reasoning about temporal properties for finite-state systems [30].

In model checking, the most common underlying formalisms to encode specifications are *temporal logics* [121] restricted to future operators. Linear temporal logic (LTL) [118] is a frequent example, although for their first automated model checker, Clarke and Emerson used computation tree logic (CTL) [30]. These logics extend classical logics by adding temporal operators. For instance, the next operator $\bigcirc P$ asserts that property P holds in the following state, the always operator $\Box P$ states that P is true in all upcoming states, eventually $\Diamond P$ indicates the existence of a future state where P holds, and the until operator $P \; \mathrm{U} \; Q$ says that P will hold up to a point where Q becomes true. The finite state semantics helps in the decidability of formulas involving these operators.

Improvements to model checking, like those done for *symbolic* [25] or *bounded* [17, 18] model checkers, have made this technique capable of handling more complex systems over time. By 2017, model-checking could compete with testing, a frequently used approach in industry, to the point where model checkers could even find more software bugs than testing tools [16]. Nevertheless, current model checking techniques are still better suited for falsification rather than verification, that is, they are better at finding logical errors than showing that they do not exist [17]. The reason for this is that, in many applications, the state space is considerably larger than what current methods can handle. For this reason, in the case of safety-critical systems like some hybrid systems or security protocols,

model checking complements other more time-consuming but also more exhaustive methods like deductive verification. In these approaches, mathematical ingenuity and expertise take a more relevant role aside from computing power. In the upcoming section, we explain the techniques employed in deductive verification as this is the style that our verification components implement.

2.3 Deductive Verification of Software

Another method for validating software is *deductive verification*. Through a formal language, the hybrid system designer encodes safety verification problems as correctness assertions to derive in a logical calculus. Proofs of these assertions represent proofs of correctness of the system. Hoare logic was one of the pioneering and most influential works in this area where the statements that need to be proved are *Hoare triples* $\{P\}\, X\, \{Q\}$ [65]. These are *partial correctness specifications* stating that if the system starts in a state satisfying P, and if the execution of X terminates, then it does so in a state where Q holds. This is in contrast with total correctness that requires proving termination of X instead of assuming it. The particular case of digital controllers interacting with their physical environment is not generally considered a process that terminates, thus, total correctness in deductive verification of hybrid systems is often delegated as future work [103, 134] or not mentioned [108]. Moreover, in some cases, convergence of time is an unsafe property [60, 61]. For these reasons and to simplify our presentation, in this work we will focus only on partial correctness assertions.

Theoretically, Hoare triples provide a general formalisation for modal reasoning about any system X and any constrains P and Q on it. The obstacle then becomes finding a language to model systems like X and (a possibly different language to model) assertions P, Q. In one of its simplest conceptions for verification of programs [58, 65], the underlying language for assertions is first order logic and that for programs consists of *while-programs*. The latter are described with the grammar

$$X \;::=\; x := e \mid X\,;X \mid \texttt{IF}\ P\ \texttt{THEN}\ X\ \texttt{ELSE}\ X \mid \texttt{WHILE}\ P\ \texttt{DO}\ X,$$

where x is a variable, e is a term, and P is a formula. That is, the grammar builds programs recursively well-known basic commands such as *assignments* ($x := e$), *sequential compositions* (;), if-then-else branching statements and while-loops. Afterwards, Hoare logic requires the presentation of rules of inference about Hoare triples for each programming construct (see Section 3.2). The emerging proof-calculus allows us to obtain formal proofs about correctness specifications involving this simple class of programs.

Another formalism for deductive software verification is *dynamic logic* [58]. It subsumes Hoare logic by making more expressive its language for assertions and for programs. In the case of programs, dynamic logic uses *regular programs* defined with the grammar

$$X \;::=\; x := e \mid ?P \mid X\,;X \mid X + X \mid X^*.$$

Regular programs include *tests* (?), *(nondeterministic) choices* (+), and *finite iterations* (*). Intuitively, a test checks whether property P holds. If it does, it ends without changing

the state, otherwise, it blocks the execution without terminating. Similarly, a choice $X + Y$ indicates the execution of program X or Y without us knowing which one. Finally, the finite iteration of a program X^* expresses that X repeats an unknown amount of times. Regular programs generalise while-programs by combining nondeterminism with tests, for example, the if-then-else branching corresponds to `IF` P `THEN` X `ELSE` Y `=` $(?P\,;X) + (?(\neg P)\,;Y)$.

At the level of formulas, dynamic logic adds box $|X]\,Q$ and diamond $|X\rangle\,Q$ modal operators to first order logic. The box operator guarantees that Q holds after termination of X while the diamond operator asserts the existence of a state where Q is true after the execution of X. Thus, the diamond operation provides a different way to reason about programs, namely, in terms of their possible outputs. Furthermore, for deterministic programs like while-programs, it helps the logic to reason about total correctness. Meanwhile the forward box allows us to encode the Hoare triple $\{P\}\,X\,\{Q\}$ with the formula $P \to |X]\,Q$. This is consistent with another formal language for reasoning about programs, namely Dijkstra's weakest liberal precondition calculus [36]. That is, $|X]\,Q$ is a precondition $\{|X]\,Q\}\,X\,\{Q\}$, and it is the weakest in the sense that every other precondition for X and Q implies it. This feature allows us to reason in this logic equationally as opposed to relationally with Hoare logic's rules of inference. This is because most rules for weakest liberal preconditions are equations.

As seen in this section, Hoare logic and related formalisms are a modular way to reason about programs. In particular, the approach is general and may be adapted to include other features or systems. For instance, we have just described the extension from while-programs to regular programs but there is also separation logic that allows computer scientists to reason about programs with shared mutable data structures [122]. In this work, we respect this modularity and extensibility and apply it to our own development of deductive verification components for hybrid programs. That is, we extend the language of regular programs and add a command that represents differential equations as pioneered in differential dynamic logic $\mathsf{d}\mathcal{L}$ [108, 113]. Yet, our implementation is not restricted to dynamic logic but can adapt and absorb other languages for specifications as exemplified in Sections 3.2, 3.3, 6.2 and 6.3. Before describing the integration of differential equations into deductive verification, we explain how this is done for finite state systems in the upcoming section.

2.4 Hybrid Automata and Reachability Analysis

Hybrid automata appeared in the literature in 1993 as a generalisation of timed automata [5]. They are finite labelled transition systems enriched with continuous dynamics φ at each node. Additionally, for modelling boundary conditions, a predicate G restricts the domain of evolution of the dynamics. Other predicates on nodes, P and Q, indicate the initial and final conditions respectively. Finally, the edges of hybrid automata also contain jump conditions that indicate variable updates and when edges can be traversed.

Figure 2.3 displays a hybrid automaton that models our simplified thermostat of Example 2.1.1 that measures the temperature T in a room, and turns a heater on or off accordingly. In the ON node, we annotate the differential equations modelling the temperature when the heater is on and the corresponding restriction (predicate G) on the domain of evolution. The OFF node does the same thing for when the heater is off.

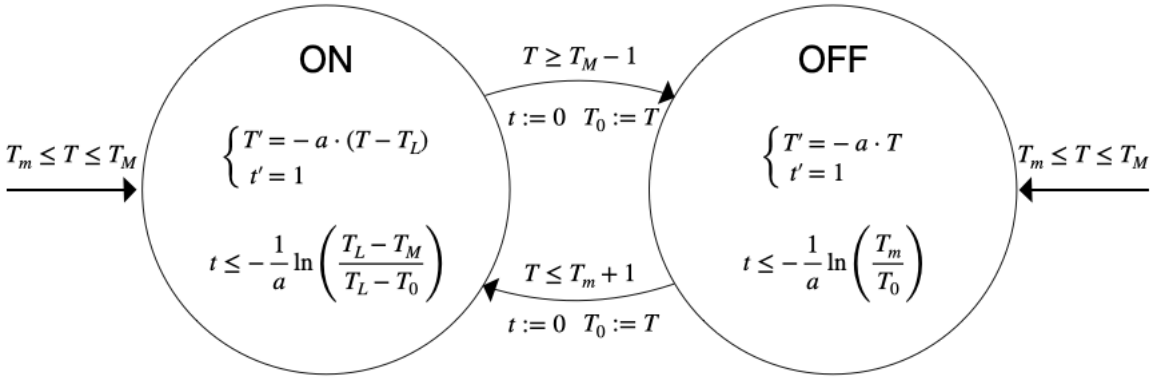


Figure 2.3: Hybrid automaton for a (simplified) thermostat

Traditional representations of the thermostat use temperature restrictions, $T \leq T_M$ for ON and $T_m \leq T$ for OFF, instead of time restrictions. We abandon this approach because it trivialises the property we want to prove, namely $T_m \leq T \leq T_M$. With the time bounds that we provide, at least we have to reason about the solution of the differential equation in order to see that that property holds. Thus, our time bounds guarantee that T remains within the comfortable range $T_m \leq T \leq T_M$ (with $0 \leq T_m$ and $T_M \leq T_L$) which is also the initial condition (P) as indicated by the outer arrows in Figure 2.3. The postcondition (Q) is not explicit, but it coincides with this comfortable range. The thermostat cannot alter the heater unless T is close to the boundary of the comfortable zone as per the jump conditions $T \geq T_M - 1$ and $T \leq T_m + 1$. These also state that whenever the thermostat interferes, it records the current temperature of the room ($T_0 := T$) and resets t ($t := 0$). See Figure 2.6 and its accompanying paragraph in the following section for a graph describing a possible evolution in time of this hybrid system.

With a hybrid automaton, the verification process iterates the transition relation of the automaton to reach states where the safety specification is true. There are various approaches to perform this procedure. One of them borrows ideas from bounded model checking. That is, the procedure encodes the safety verification problem in a propositional formula that SAT methods can tackle [17, 40, 51]. Yet, the preferred approach of many tools like BACH [24], CORA [2], Flow* [27] and SpaceX [48] is a *reachability analysis*. A state is reachable if there is a finite sequence of discrete and continuous transitions leading to it. Reachability analysis thus approximates the set of all reachable states as the union of some state set representations like boxes, polyhedra or ellipsoids [39]. That is, the procedure successively iterates the transition relation of the automaton starting from an approximation of the set of initial states. The process stops if this iteration reaches a fixed point or if it reaches a state that violates the safety specification.

As one of the pioneering models for hybrid systems, hybrid automata are ubiquitous in the literature. Choosing them for verification relies primarily on their potential for automation. Many algorithms exist for solving somewhat complex verification problems with them. However, as with model checking, these approaches are more suitable for finding errors rather than proving that they do not exist [62, 108]. Moreover, verification problems for hybrid automata are generally undecidable and their algorithms have to deal with

computational infeasibility [39]. It is also difficult to handle complex continuous dynamics with these methods. Their algorithms are often restricted to simpler subclasses of automata, either by the class of constraints or the class of differential equations allowed in the nodes [39]. In general, vast parts of the research on this area focus on improving the approximations for the reachable states for specific subclasses of automata. In contrast, in deductive verification of hybrid systems, computer scientist can use well-known mathematical techniques for manipulating differential equations and use them to reason about more complex continuous dynamics. The trade-off is less automation and a more time-consuming process but this is acceptable for highly safety-critical systems such as many of those modelled with hybrid systems. In this work, we show that we can do deductive verification of hybrid systems in the style of $\mathtt{d}\mathcal{L}$ (Section 5.6) that uses hybrid programs. Given that there is an embedding from hybrid automata into hybrid programs [108], our framework could also be extended to provide tools for reasoning about hybrid automata.

2.5 Hybrid Programs and Deductive Verification

The appeal of using deductive verification for hybrid systems is that its techniques are not constrained to finite-state systems or relatively easy continuous dynamics. Moreover, these methods allow modular reasoning because of the compositional nature of their syntax driven approach. In fact, this is one of the main arguments for the introduction of *hybrid programs* [108] that extend regular programs with *evolution commands* ($x' = f \,\&\, G$). The grammar

$$X \;::=\; x := e \mid x' = f \,\&\, G \mid ?P \mid X \,;\, X \mid X + X \mid X^*,$$

where G models boundary conditions like in hybrid automata, concisely describes the syntax for hybrid programs. Intuitively, evolution commands depict the differential equations that model the continuous behaviour of a hybrid system while composition of the remaining programming constructs model the discrete control. More specifically, while assignments and tests represent an instantaneous change in the value of a hybrid system's state, the evolution command describes a continuous change guided by the solution to the differential equation it depicts (see Figure 2.4). Furthermore, evolution commands can only execute while they satisfy the boundary condition G. This means two things: if we try to execute an evolution command in a state outside of G, it has the same effect as a failed test. Similarly, if we execute an evolution command in a state satisfying G, it will end up in any of the states in the trajectory of the solution to the ODE—as long as that solution remains within G.

Integrating hybrid programs into a proof calculus like Hoare logic allows the user to prove correctness specifications. For instance, the Hoare triple of Figure 2.5 uses the hybrid program $\mathtt{thermostat} = (\mathtt{ctrl}\,;\mathtt{dyn})^*$ to model a thermostat regulating the temperature of a room through a heater like in Sections 2.1 and 2.4. The variables $a, t, \theta, T, T_0, T_m, T_M$ and T_L denote the same parameters as before. The hybrid program `ctrl` models the discrete intervention of the thermostat, that is, it resets variables t and T_0 and if T is close to an uncomfortable region, it turns the heater on or off accordingly. On the other hand, `dyn` describes the continuous dynamics: if the heater is not radiating, then the room's temperature decreases and vice versa. Finally, `thermostat` is just a finite iteration of the control followed by the

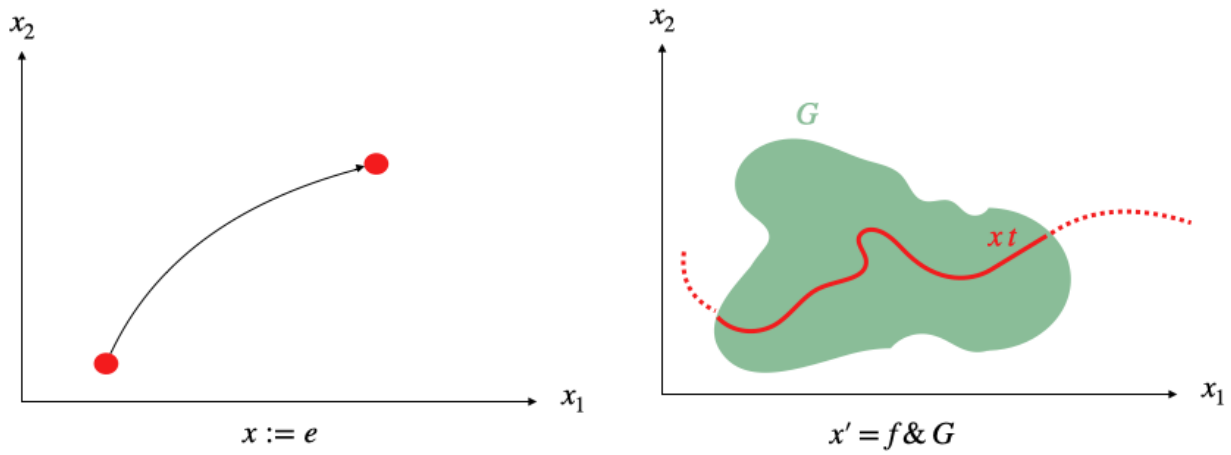


Figure 2.4: Assignments change the hybrid system's state instantaneously. Evolution commands change it continuously. The dotted lines represent parts of the solution to the differential equation where evolution commands cannot execute because they are outside of the boundary condition G.

dynamics. There is a proof of the safety specification for `thermostat` using the rules of Hoare logic and an extra rule for evolution commands [44].

To provide further intuition behind the abstraction of hybrid systems as hybrid programs, we provide a non-rigorous depiction of the evolution of our thermostat in Figure 2.6. It starts with the heater on, which explains the exponential increase in temperature. Then, after the first intervention of the control (represented with a vertical dotted line) and due to the fact that it is close to the upper bound of the comfortable region, it turns the heater off. The second and third interventions of the control do nothing as indicated by the "else" branch of the if-statement. Notice that control interventions need not be evenly spaced because the duration of the evolution command is nondeterministic. The control intervenes at the right moment on the first, fourth and fifth dotted lines because the boundary conditions of

$$
\begin{array}{l}
\texttt{ctrl} \;=\; (t := 0)\,;(T_0 := T); \\
\qquad\quad \texttt{IF}\ \ (\theta = 0 \wedge T_0 \leq T_m + 1)\ \ \texttt{THEN}\ \ (\theta := 1)\ \ \texttt{ELSE} \\
\qquad\quad \texttt{IF}\ \ (\theta = 1 \wedge T_0 \geq T_M - 1)\ \ \texttt{THEN}\ \ (\theta := 0)\ \ \texttt{ELSE skip}
\end{array}
$$

$$
\begin{array}{l}
\texttt{dyn} \;=\; \texttt{IF}\ \ \theta = 0\ \ \texttt{THEN} \\
\qquad\quad T' = -a \cdot T,\, t' = 1\ \ \&\ \ t \leq -\frac{1}{a} \ln\left(\frac{T_m}{T_0}\right) \\
\qquad\quad \texttt{ELSE} \\
\qquad\quad T' = -a \cdot (T - T_L),\, t' = 1\ \ \&\ \ t \leq -\frac{1}{a} \ln\left(\frac{T_L - T_M}{T_L - T_0}\right)
\end{array}
$$

$$\{T_m \leq T \leq T_M\}\,(\texttt{ctrl}\,;\texttt{dyn})^*\,\{T_m \leq T \leq T_M\}$$

Figure 2.5: Hybrid program for a (simplified) thermostat

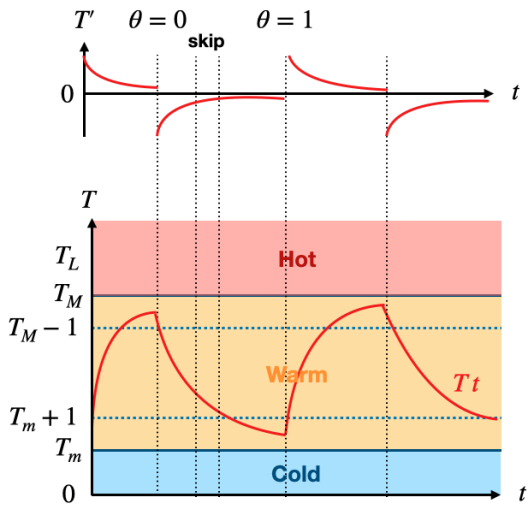


Figure 2.6: Depiction of a possible evolution in time of the thermostat hybrid system

the evolution command prohibit the solution to the differential equation to go beyond the comfortable region. Thus, as soon as it gets close to them, the thermostat turns the heater on or off accordingly. In contrast with the continuous evolution of the temperature, the derivative of the temperature discretely jumps from positive to negative and vice versa after the heater changes its state.

Deductive verification of hybrid programs has been successfully applied in many case studies [77, 81, 116]. Logical calculi and their corresponding tools designed for this approach include the *hybrid communicating sequential processes (HCSP)* [86] with the *HHL prover* [135], and *differential dynamic logic* ($\mathsf{d}\mathcal{L}$) and its various extensions [108, 113] with its flagship proof assistant *KeYmaera X* [50]. The logic $\mathsf{d}\mathcal{L}$ is a prominent approach because of its pioneering work introducing hybrid programs and a proof calculus to reason about invariants of differential equations [107] (see Section 5.2 for definitions of these invariants). For decidability concerns, it uses expressions of real arithmetic as terms and first order logic formulas for efficient quantifier elimination [108]. The restriction on its term-language further allows $\mathsf{d}\mathcal{L}$ to guarantee that all differential equations expressible in the logic have unique solutions (see Section 4.1 for requirements on existence and uniqueness of solutions to ODEs). These are domain-specific tools and formalisms with a relatively small user-base and focused research groups behind them that constantly maintain and improve the tools. Moreover, their underlying logics restrict the expressivity of the continuous dynamics and mathematical expressions available to the user. From an extremely distrustful point of view, their tools still require uncertified external input for solving differential equations and generating invariants. In contrast, throughout this work we discuss an alternative, open, modular and extensible approach to deductive verification of hybrid programs with wide adoption potential.

2.6 Verification with General-purpose Proof Assistants

Instead of depending on domain-specific tools, we can apply a technique from deductive verification of software: embed the programs and the verification process in the logic of a general-purpose proof assistant [9, 100, 125]. An advantage of this is that the most widely used general purpose interactive provers have larger user-bases than specialised provers and the scientific community actively develops them. Moreover, many proof assistants have been available since the late 1980s [59] and each year, their libraries of formalised mathematics and their user base grow. This means that embedded verification components benefit in two ways. Firstly, they can exploit improvements in proof automation for the interactive theorem prover, and secondly, users of the proof assistant can quickly adopt them and enhance them. Another advantage is that the base logic of general-purpose proof assistants is usually more expressive than first-order logic. Therefore, the verification components can also be more expressive than current domain-specific tools. Finally, embedding a verification tool makes it correct by construction relative to the trust on the interactive theorem prover. Yet, trust on the components themselves can be made equal to that of domain-specific proof assistants. That is, by connecting the general-purpose proof assistant to external software, the verification process can either certify or trust the information provided by external tools and use it.

For deductive verification of hybrid systems, the choice of a proof assistant is difficult as many of them share similar developements. In terms of automation, at the start of our project, *Isabelle/HOL* was leading in proof-automation thanks to its Sledgehammer tool that connects it to automated theorem provers like Z3 [33], SPASS [137] or CVC4 [13]. In contrast, we had no knowledge of similar tools [6] for the *Coq* proof assistant and development of a "hammer" for Coq is still in progress [19, 32] as it is for HOL-ligtht [80]. Moreover, in terms of formalised libraries, Isabelle/HOL provides a huge library of Analysis included in its distribution. It starts with topological spaces and filters which helps it define in a general way limits and derivatives [67]. The crucial formalisation of Picard-Lindelöf's theorem that guarantees existence and uniqueness of solutions is available online in en entry about ordinary differential equations [74] in Isabelle's *Archive of Formal Proofs* (AFP). The AFP is a refereed collection of proof libraries mechanically checked by Isabelle and organised as a scientific journal. In contrast, Coq provides various developments of real numbers and ways to do analysis. For instance, the Coquelicot library [23] is inspired on Isabelle's HOL-Analysis library and similarly develops filters to formalise convergence. It has been successfully used for the formalisation of the control function of an inverted pendulum [124]. However, the formalisation of Picard-Lindelöf's theorem is only available in a separate library, CoRN, of constructive real numbers [31, 91]. In the end, we choose Isabelle/HOL because it is a proof assistant that combines a high level of automation with a unique big and coherent library of theorems about differential equations. However, if this coherence is attained in other proof assistants, we foresee no obstacle for them to include a version of our framework described in Section 1.1.

Roughly speaking, there are two ways of embedding a verification component inside a proof assistant. The first consists in doing a *deep embedding*, that is, using the proof assistant's logic to define the syntax and semantics of the components. Afterwards, by formalising the soundness of the inference rules, one can use these results to reason about the entity to be verified. An example of a deep embedding of a Hoare-style logic in Isabelle/HOL

is in [125]. The alternative to this approach is a *shallow embedding*. As opposed to their deep counterparts, shallow embeddings implement the tool's terms and syntax as direct functions on the semantics within the host language. Examples of these embeddings are [9] and [100] in Isabelle/HOL and Coq respectively. More specifically, the difference relies on the fact that deep embeddings force the traversal of an abstract syntax tree for evaluation of the terms while deep embeddings bypass this intermediate traversal and directly operate on the semantics. As a consequence, shallow embeddings make development of new terms easier [52] because they are just direct function definitions whereas new terms in deep embeddings involve creating new branches in the syntax tree and assigning them a semantic meaning. However, a downside of shallow embeddings is that they do not adapt quickly if the semantics change because the type of the functions may need to change too. In contrast, in deep embeddings, one may just instantiate the semantics to the already existing syntax and hide the underlying changes to the user. The AFP already contains a shallow embedding for deductive verification of while-programs [9, 56] through an intermediate semantics of Kleene algebras. Extending those verification components to reason about hybrid programs benefits us because the program algebras remove the downsides of using a shallow embedding. As we show in sections 3.2, 3.3 and 3.5, changing the semantics is easy because a modular instantiation of the Kleene algebras to any new semantics is possible through Isabelle's type polymorphism. Moreover, the Kleene algebras provide variants of the laws of propositional dynamic logic on which $\mathsf{d}\mathcal{L}$ is based. Therefore, in order to have verification components for hybrid programs in Isabelle/HOL, one only needs to instantiate the Kleene algebras to the semantics of $\mathsf{d}\mathcal{L}$ and define the missing terms like the evolution commands in this semantics. In this thesis, we explore this framework for verification of hybrid systems inside a general-purpose proof assistant, its extensibility, modularity and limitations.

Apart from the work on deductive verification of software and development of libraries for ODEs mentioned in previous paragraphs, hybrid systems verfication in general-purpose proof assistants is an active field of research. In PVS [102], there is work based on the semantics of hybrid automata for invariant reasoning about hybrid systems [1] and some first steps in doing verification in the style of $\mathsf{d}\mathcal{L}$ by formalising semi-algebraic sets and real analytic functions [126]. In Coq, the ROSCoq framework (short for robot operating system) is also a shallow embedding for reasoning about hybrid systems, but instead of using hybrid programs and intermediate algebraic semantics it is based on a Logic of Events (LoE) and models time with Coq's CoRN library of constructive real numbers. However, its automation for handling derivatives is limited. Therefore, users need to supply various manual proofs for their formalisations. Alternatively, the VeriDrone project [123] uses the Coquelicot library and it is based on a variant of the temporal logic of actions (TLA) [85]. However, to reason about ODEs they only use $\mathsf{d}\mathcal{L}$'s rule for invariants. In contrast, and as stated before, our framework is not restricted to a particular formal system and can potentially absorb both styles of reasoning. In Isabelle/HOL there are other relevant projects. The HHL prover has had two main versions, first as a deep embedding and then as a shallow embedding [135]. However, the terms of its underlying logic, the calculus of hybrid communicating sequential processes (HCSP) [86], are still deeply embedded. The logic itself implements a hybrid Hoare logic to reason about HCSP processes. It also benefits from the LZZ method [87] for finding semi-algebraic invariants for polynomial dynamical systems that we still need to add to our verification components. Two $\mathsf{d}\mathcal{L}$-related Isabelle/HOL formalisations consist of a

term-checker for dℒ's domain-specific proof assistant, KeYmaera X [22], and a formalisation of differential game logic (dG ℒ) [114], an extension of dℒ to also reason about adversarial dynamics. Both of them are deep embeddings and while the proof-checker provides some simple examples, the entry on dG ℒ does not. Our framework still lacks the inclusion of adversarial dynamics, but theoretically, if they are available in the dG ℒ's deeply embedding, they can be shallowly embedded in our framework. A different hybrid systems verification framework is based on Hoare and He's unifying theories of programming (UTP) [45,46]. Its abstraction of the state space allows it to divide program variables in discrete and continuous. We have recently combined our efforts in [44] and are now working towards an improved verification tool combining the best of both approaches.

2.7 Introduction to Isabelle/HOL's Notation

Due to the fact that existing Isabelle libraries guide our developments and because we frequently present the formalisation of each new mathematical concept, in this section we provide a short introduction to Isabelle's notation and terminology. As a consensus, we think of *Isabelle* as a generic theorem prover. This means that it is a logical framework capable of supporting various object logics with Isabelle's logic as their meta-theory [105]. Naming of the object logics requires adding to the word "Isabelle" a slash "/" and the abbreviation for the logic, for example Isabelle/FOL, Isabelle/ZF or Isabelle/HOL. The last of these is the most popular and most developed of all the object logics.

Isabelle's meta-logic is a type theory with base type *prop* and three functions on it: implication ($P \Longrightarrow Q$), universal quantification ($\bigwedge x.\ P\ x$), and equality ($P \equiv Q$). Theorems of Isabelle/HOL are then terms of type *prop*.

In the object logic, base types of interest to us are the boolean constants *True* :: *bool* and *False* :: *bool*; natural numbers, $0, 1, 2 \ldots$ :: *nat*; integers *int*, and real numbers *real*. Furthermore, Isabelle/HOL also provides type variables *'a*, *'b*, ... (read *alpha*, *beta*, ...) and type constructors to form new types. Below we provide a brief description for the main type constructions.

Function type The constructor for this type is $'a \Rightarrow 'b$ and represents all functions from $'a$ to $'b$. Function application for function $f :: 'a \Rightarrow 'b$ and term $t :: 'a$ is by juxtaposition $f\ t :: 'b$ as in functional programming languages. Similarly, we can introduce functions in Isabelle as lambda-abstractions $\lambda x.\ f\ x :: 'a \Rightarrow 'b$ which input $x :: 'a$ and output $f\ x :: 'b$. The identity function constant $id :: 'a \Rightarrow 'a$ is available for every type $'a$. Finally, important operations involving this type include function composition $(f \circ g)\ t = f\ (g\ t)$ and function updates $f(a := b)$ that map a to b and every other t to $f\,t$.

Product type As many other typed systems, Isabelle has a product of types $'a$ and $'b$ whose intended terms are ordered pairs $(x, y) :: 'a \times 'b$. Thus, it includes projection functions $fst\ (x, y) = x$ and $snd\ (x, y) = y$ as well as the constructor $Pair\ x\ y = (x, y)$.

Sum type Opposingly, the type $'a + 'b$ is the sum of $'a$ and $'b$. It represents a disjoint union of terms of $'a$ and $'b$. As such, it provides a left inclusion $Inl :: 'a \Rightarrow 'a + 'b$ and a right

inclusion *Inr* :: $'b \Rightarrow 'a + 'b$, as well as their converse left *projl* and right *projr* projections.

Type of sets A frequently used type in our formalisations is the type of sets $'a\ set$ for a given type $'a$. If we interpret $'a$ as a set, $'a\ set$ corresponds to its powerset. Some important terms of this type are the universal set of a type *UNIV* :: $'a\ set$, containing all elements of type $'a$, and the empty set $\{\}$ :: $'a\ set$ with no elements. We also have traditional set constructors like set comprehension $\{x \in A.\ P\,x\}$, for the subset of A whose elements satisfy the predicate P :: $'a \Rightarrow bool$, and a version of set replacement $\{f\,x \mid x.\ P\,k\}$ for all terms of shape $f\,x$ that make $P\,k$ true. In turn, union $A \cup B$, intersection $A \cap B$ and complementation $A - B$ for sets use well-known notations. The same applies for the elementhood relationship $x \in A$ and the subset relation $A \subseteq B$.

Type of lists Isabelle provides a list data type $'a\ list$. An inductive definition via the empty list [] or *Nil*, and the constructor *cons* with infix notation # creates the terms of this type. The expressions $x_1 \#\ x_2 \#\ x_3 \#\ []$ and $[x_1, x_2, x_3]$ denote the same list of type $'a\ list$. Among other functions, it includes the append operation $l@x$ for the list l :: $'a\ list$ and term x :: $'a$, and $l\,!\,n$ for the retrieval of the nth element in l.

Moreover, in our formalisations along the thesis, we use some syntactic abbreviations for specific types. For instance we heavily use $'a\ rel$ to denote the type of relations over $'a$, but in reality it corresponds to the type $('a \times 'a)\ set$. Important constants for this type include the identity relation *Id* :: $'a\ rel$ and relational composition $R_1\,;R_2$ between relations R_1 and R_2. Similarly, we provide an abbreviation $'a\ pred$ for the type of predicates $'a \Rightarrow bool$.

To add new theorems to Isabelle/HOL, we use the command **lemma** together with a term of type *prop*. Alternative commands are **theorem**, **proposition**, and **corollary**, yet operationally they perform the same action. To add new constants c :: $'a$ to the logic, we must use the command **definition**. The latter receives a defining equality $c = f\,x$, where the codomain of f is $'a$, and adds it as a theorem. When users wish to provide representations of specific objects without adding theorems about them, they can use the command **abbreviation** instead. This has the same structure as a definition, except that the defining equality $c \equiv f\,x$ is at the meta-level. We use these commands in Figure 2.7 that depicts the formalisation of the correctness specification for the hybrid program of Section 2.5, that is, it has a proof of the Hoare triple $\{\!|I\ T_m\ T_M|\!\}\ therm\ T_m\ T_M\ a\ T_L\ t\ \{\!|I\ T_m\ T_M|\!\}$.

Each time that users employ the command **lemma**, they have to provide a proof of the theorem. The statement accompanying **lemma** then becomes the proof obligation. Users can modify proof-obligations and eventually discharge them with the **apply** command and a tactic. The first line in the proof of Figure 2.7 uses this command with our tactic *hyb-hoare* to turn the proof-obligation into various proofs about the semantics of the hybrid program. Isabelle also offers the scripting language *Isar* that uses keywords to make proofs resemble mathematical practice. These keywords are available in the declaration of lemmas too. Figure 2.7, for instance, uses the keywords **assumes**, **and**, and **shows**.

An Isabelle file containing lemmas, definitions, theorems and abbreviations is a *theory*, as in collection of lemmas, hence their file-extensions (.thy). Theories can import other theories to extend their developments. We refer to a collection of theories imported by a working theory as its *theory stack*.

abbreviation $I\ T_m\ T_M \equiv \mathbf{U}(T_m \leq T \wedge T \leq T_M \wedge (\vartheta = 0 \vee \vartheta = 1))$

abbreviation *ctrl* $T_m\ T_M \equiv$
$(t ::= 0);\ (T_0 ::= T);$
$(IF\ (\vartheta = 0 \wedge T_0 \leq T_m + 1)\ THEN\ (\vartheta ::= 1)\ ELSE$
$IF\ (\vartheta = 1 \wedge T_0 \geq T_M - 1)\ THEN\ (\vartheta ::= 0)\ ELSE\ skip)$

abbreviation *dyn* $T_m\ T_M\ a\ T_L\ t \equiv$
$IF\ (\vartheta = 0)\ THEN\ x' = f\ a\ 0\ \&\ (G\ T_m\ T_M\ a\ 0)\ on\ \{0..t\}\ UNIV\ @\ 0$
$ELSE\ x' = f\ a\ T_L\ \&\ (G\ T_m\ T_M\ a\ T_L)\ on\ \{0..t\}\ UNIV\ @\ 0$

abbreviation *therm* $T_m\ T_M\ a\ T_L\ t \equiv$
$LOOP\ (ctrl\ T_m\ T_M;\ dyn\ T_m\ T_M\ a\ T_L\ t)\ INV\ (I\ T_l\ T_h)$

lemma *thermostat-flow*:
assumes $0 < a$ **and** $0 \leq t$ **and** $0 < T_m$ **and** $T_M < T_L$
shows $\{I\ T_m\ T_M\}$ *therm* $T_m\ T_M\ a\ T_L\ t\ \{I\ T_m\ T_M\}$
apply(*hyb-hoare* $\mathbf{U}(I\ T_m\ T_M \wedge t{=}0 \wedge T_0 = T)$)
prefer *4* **prefer** *8* **using** *local-flow-therm assms* **apply** *force+*
using *assms therm-dyn-up therm-dyn-down* **by** *rel-auto'*

Figure 2.7: Isabelle/HOL code for the `thermostat` hybrid program

A guiding policy for Isabelle developments is that the logical core remains small while each new theorem or model has to be correct relative to it and the correctness of the theory stack. A consequence of this is that any external input needs to be certified within the proof assistant. For instance, when proving a theorem, Isabelle/HOL users may employ the *Sledgehammer* tool that calls external SMT solvers and automated theorem provers. If any of them returns a proof of the theorem, Sledgehammer reconstructs it in Isabelle's syntax which then the user can paste to let the proof assistant certify it. The same correctness criteria apply to our formalisations in the remaining chapters: each Isabelle lemma in this thesis has been certified with Isabelle/HOL. Thus, they may be regarded as true mathematical results relative to Isabelle's small logical core and their theory stack.

Chapter 3

Kleene Algebras

In this chapter, we present variants of Kleene algebras to serve as the algebraic foundations for deductive verification of hybrid systems. Originally, they were algebras for regular expressions [82], although our interest in them is as models to reason about regular programs. This is why in Section 3.1, we formally define Kleeene algebras and explore this interpretation together with some of their semantics. Then, in Section 3.2, we present a variant of Kleene algebras that includes tests. We also extend the semantics of Section 3.1 to capture these tests and provide definitions that encode while-programs with these algebras. In Section 3.3, we introduce a more expressive variant capable of doing equational reasoning with predicate transformers. Finally, in Section 3.5, we describe the formalisations of these algebras in the interactive theorem prover Isabelle/HOL. This chapter provides more detailed explanations about Kleene algebras than [72] and [44].

3.1 Kleene Algebra

In this section, we introduce the basic structure of Kleene algebras and their canonical relational semantics. Additionally, we explain their alternative state transformer semantics in more detail because of their connection with dynamical systems.

A *dioid* $(S, +, \cdot, 0, 1)$ is a structure formed of two monoids, a multiplicative monoid $(S, \cdot, 1)$ and an idempotent abelian monoid $(S, +, 0)$, that satisfy distributivity and annihilation axioms. That is, for all $\alpha, \beta, \gamma \in S$, the dioid laws are

associativity	$\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$		$\alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$	
identity	$\alpha + 0 = \alpha$	$0 + \alpha = \alpha$	$\alpha \cdot 1 = \alpha$	$1 \cdot \alpha = \alpha$
commutativity	$\alpha + \beta = \beta + \alpha$			
idempotency	$\alpha + \alpha = \alpha$			
distributivity	$(\alpha + \beta) \cdot \gamma = \alpha \cdot \gamma + \beta \cdot \gamma$		$\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma$	
annihilation/absorption	$\alpha \cdot 0 = 0$	$0 \cdot \alpha = 0.$		

By defining $\alpha \leq \beta \leftrightarrow \alpha + \beta = \beta$ for all $\alpha, \beta \in S$, the introduced relation $\leq$ is a partial order due to idempotency and associativity of $+$. Moreover, for any two elements $\alpha, \beta \in S$, their addition $\alpha + \beta$ is their lowest greater bound. In other words, $(S, \leq)$ is a *join semilattice*.

Furthermore, both $\cdot$ and $+$ preserve this order in both arguments and, by the identity laws, 0 is the least element of S.

If we interpret the elements of dioids as programs, their operations already provide two of the hybrid program constructors from Section 2.5, that is, addition corresponds to nondeterministic choice and multiplication to sequential composition. However, dioids come not only with operations but with axioms too. These are equalities asserted under the assumption that equivalent programs generate the same output from the same inputs. For instance, the associativity laws state that local (binary) choices ($+$) do not affect global outputs and that the output of sequentially ($;$) executing three programs should ignore the (binary) composites involved in the process. Furthermore, dioids provide constants for the ineffective and aborting programs, 1 and 0 respectively. Therefore, their corresponding laws also describe their behaviour relative to all other programs. The absorption laws dictate that the output is abortive irrespective of the order of the execution of the abortion. Similarly, the multiplicative identity laws state that an ineffective program in sequential composition with another program provides the same output as one where no ineffective program executes. Finally, the order ($\leq$) on programs reveals redundancy. The inequality $\alpha \leq \beta$ says that a choice between α or β has the same input/ouput behaviour as simply β.

Along this line of reasoning, we use *Kleene algebras* $(K, +, \cdot, 0, 1, ^*)$ to capture finite iteration of hybrid programs. These algebras enrich dioids with a Kleene star operation $(-)^* : K \to K$ that satisfies, for all $\alpha, \beta, \gamma \in K$, the axioms

$$\begin{array}{lll} \textit{unfold} & 1 + \alpha \cdot \alpha^* \leq \alpha^* & 1 + \alpha^* \cdot \alpha \leq \alpha^* \\ \textit{induction} & \gamma + \alpha \cdot \beta \leq \beta \to \alpha^* \cdot \gamma \leq \beta & \gamma + \beta \cdot \alpha \leq \beta \to \gamma \cdot \alpha^* \leq \beta. \end{array}$$

The intuition behind these star axioms becomes more apparent by applying the dioid laws to the first unfold axiom:

$$\begin{array}{rl} 1 + \alpha \cdot \alpha^* \leq \alpha^* & \text{unfold} \\ (1 + \alpha \cdot \alpha^*) \cdot \gamma \leq (\alpha^*) \cdot \gamma & \text{monotonicity of } (\cdot) \\ \gamma + \alpha \cdot \alpha^* \cdot \gamma \leq \alpha^* \cdot \gamma & \text{distributivity} \end{array}$$

Therefore, by the first induction axiom, $\alpha^* \cdot \gamma$ is the least solution β to the linear inequality $\gamma + \alpha \cdot \beta \leq \beta$. Similarly, $\gamma \cdot \alpha^*$ is the least prefixpoint of the monotone function $\lambda\beta.\ \gamma + \beta \cdot \alpha$. The special case α^* behaves like the Kleene star string of formal language theory or a reflexive transitive closure in relational algebra. In fact, there is a semantic interpretation of all Kleene algebra operations in both semantics.

In the relational interpretation, a program is a set of input-output pairs of states, that is, programs are *binary relations* $R \subseteq S \times S$ (equivalently $R \in \mathcal{P}(S \times S)$). We use $s_1\ R\ s_2$ instead of $(s_1, s_2) \in R$. Then recall that if $R_1 \subseteq S_1 \times S_2$ and $R_2 \subseteq S_2 \times S_3$, their composition $R_1\,;R_2$ is a relation such that if $s_1\ R_1\ s_2$ and $s_2\ R_2\ s_3$, then $s_1\ (R_1\,;R_2)\ s_3$. Moreover, $Id_S \subseteq S \times S$ relates every element of S to itself, and the reflexive transitive closure of a relation $R \subseteq S \times S$ is $R^* = \bigcup_{i \in \mathbb{N}} R^i$ where $R^0 = Id_S$ and $R^{i+1} = R^i;R$. Translating to Kleene algebras, multiplication corresponds to composition, addition to union, the multiplicative unit to the identity relation, the additive unit to the vacuous relation, the Kleene star to the reflexive transitive closure, and the dioid order to set containment. This is summed up in the next proposition.

Proposition 3.1.1. *For any set S, the tuple $(\mathcal{P}(S \times S), \cup, ;, \emptyset, Id_S, ^*)$ forms a Kleene algebra—the* full relation Kleene algebra *over S.*

By formalising Kleene algebras and their relational semantics in Isabelle/HOL, we immediately obtain large parts of what we need to encode hybrid programs in the proof assistant (contrast with Section 2.5). However, there is an added benefit to our approach: we can overcome the limited adaptability of shallow embeddings to new semantics. That is, by instantiating to a different semantics of Kleene algebras, our components preserve their algebraic foundations. For instance, there is a bijection between binary relations $R \in \mathcal{P}(S_1 \times S_2)$ and *state transformers* $F \in (\mathcal{P}\, S_2)^{S_1}$ that provides an alternative semantics for Kleene algebras. Indeed, the function $\mathcal{F} : \mathcal{P}(S_1 \times S_2) \to (\mathcal{P}\, S_2)^{S_1}$ such that $\mathcal{F}\, R\, s_1 = \{s_2 \in S_2 \mid s_1\ R\ s_2\}$ has an inverse $\mathcal{R} : (\mathcal{P}\, S_2)^{S_1} \to \mathcal{P}(S_1 \times S_2)$ defined by $s_1\ (\mathcal{R}\, f)\ s_2 \Leftrightarrow s_2 \in f\, s_1$. Moreover, these functions are functorial between the *Kleisli composition* $(f \circ_K g)\, s_1 = \bigcup\{g\, s_2 \mid s_2 \in f\, s_1\}$ and the relational composition, that is

$$\mathcal{F}(R_1 ; R_2) = \mathcal{F}\, R_1 \circ_K \mathcal{F}\, R_2, \quad \text{and} \quad \mathcal{R}(f \circ_K g) = \mathcal{R}\, f ; \mathcal{R}\, g.$$

Similarly, $\mathcal{R}\, \eta_S = Id_S$ and $\mathcal{F}\, Id_S = \eta_S$ where $\eta_S\, s = \{s\}$. The terminology and notation come from category theory. The powerset operator $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ is an endofunctor on the category **Set** of sets such that

$$\mathcal{P}\, X = \{Y \mid Y \subseteq X\} \quad \text{and} \quad \mathcal{P}\, f\, S = \{f\, s \mid s \in S\}$$

is the direct image of S. Then, $\eta : \mathbf{1_{Set}} \to \mathcal{P}$ and $\mu : \mathcal{P}^2 \to \mathcal{P}$ such that $\mu_S : \mathcal{P}(\mathcal{P}\, S) \to \mathcal{P}\, S$ and $\mu_S\, X = \bigcup X$ are natural transformations, where $\mathbf{1_{Set}}$ is the identity functor on **Set** and $\mathcal{P}^2 = \mathcal{P} \circ \mathcal{P}$. The triple $(\mathcal{P}, \eta, \mu)$ then forms the *powerset monad* that has an associated *Kleisli category* $\mathbf{Set}_\mathcal{P}$. The objects of $\mathbf{Set}_\mathcal{P}$ are sets, while its morphisms are state transformers. The composition of $f \in \mathbf{Set}_\mathcal{P}(X, Y)$ and $g \in \mathbf{Set}_\mathcal{P}(Y, Z)$ is then

$$g \circ_{\mathbf{Set}_\mathcal{P}} f = g^\dagger \circ f = f \circ_K g,$$

where $(-)^\dagger$ is the *Kleisli extension* $g^\dagger = \mu \circ (\mathcal{P}\, g)$. Yet, we use $\circ_K$ to preserve covariance with our relational composition as this is the default in Isabelle/HOL. Finally, η provides the identities $id_S \in \mathbf{Set}_\mathcal{P}(S, S)$, that is $id_S = \eta_S$. Thus, $\mathcal{F}$ and $\mathcal{R}$ form an isomorphism between $\mathbf{Set}_\mathcal{P}$ and the category **Rel** whose objects are sets and morphisms are binary relations.

From here, we can apply the mono-typed instance of the isomorphism $\mathcal{F}$ to the full relational Kleene algebra over a set S to obtain state transformer semantics. In this interpretation, the state transformer represents a program by associating to each input state the set of all possible output states. By defining $f^{*_K} : S \to \mathcal{P}\, S$ such that

$$f^{*_K}\, s = \bigcup_{i \in \mathbb{N}} f^i\, s,$$

where $f^0 = \eta_S$ and $f^{i+1} = f^i \circ_K f$, we can check that $(\mathcal{P}\, S)^S$ satisfies the Kleene algebra axioms. That is, addition is pointwise union $(f \cup g)\, s = (f\, s) \cup (g\, s)$, multiplication is the Kleisli composition, the additive unit is the function that maps $s \in S$ to $\emptyset$, the multiplicative unit is the Kleisli unit η_S, and the Kleene star is $(-)^{*_K}$. We can also see that the subset order $\subseteq$ in relations translates to pointwise containment in state transformers, that is $f \leq g$ if and only if $f\, s \subseteq g\, s$ for all $s \in S$. In other words, the following holds.

Proposition 3.1.2. *For any set S, the tuple $((\mathcal{P}\,S)^S, \cup, \circ_K, \lambda s.\ \emptyset, \eta_S, {}^{*_K})$ forms a Kleene algebra—the* full state transformer Kleene algebra *over S.*

The state transformer model for Kleene algebras is important in our presentation because it fits naturally with concepts of the dynamical systems literature (see Chapter 4). Yet, we also keep the presentation for the relational model because it is better known and also because it clearly exemplifies the simple modularity of our approach. Due to our intention to use variants of Kleene algebras to model hybrid programs, in the sequel we replace the algebraic $\cdot$ for its program-counterpart ;. We also use ; indistinctly for the relational and Kleisli composition. A detailed hierarchy of variants of Kleene algebras, their calculational properties and their most important computational models are in the AFP [10]. Similarly, the formalisation of the state transformer model via the powerset monad and its Kleisli category is in [130].

3.2 Kleene Algebra with Tests

We wish to extend Kleene algebras $(K, +, ;, 0, 1, {}^*)$ so as to have the test constructor for hybrid programs of Section 2.5. Thus, we follow Kozen [83] and consider a subset $B \subseteq K$ and an operation $\neg : B \to B$ such that $(B, +, ;, 0, 1, \neg)$ is a boolean algebra. That is, $+$ is its join, ; is its meet, $\neg$ is its complementation, 0 is its least element, and 1 is its greatest element. The two-sorted structure $(K, B, +, ;, 0, 1, {}^*, \neg)$ is a *Kleene algebra with tests* (KAT).

In the program-interpretation of KAT, tests $p \in B$ either hold or fail in a program state. A test sequentially composed with $\alpha \in K$ restricts its execution. For instance, the test p restricts the input of α to those states where p holds in the composition $p\,;\alpha$. Similarly, the output of $\alpha\,;p$ corresponds to a state that satisfies p.

The relational model of Kleene algebras also holds for KATs. We can encode predicates $P : S \to \mathbb{B}$ as subsets of the identity relation Id_S via the bijection $\lceil P \rceil_{\mathcal{R}} = \{(s, s) \mid P\,s\}$. This suggests an isomorphism between the boolean algebra of predicates on S and a relational analogue. Indeed, the tuple $(\mathcal{P}\,Id_S, \cup, ;, \emptyset, Id_S, \overline{\ })$ is a boolean algebra where $\overline{R} = Id_S \setminus R$ for $R \subseteq Id_S$ and $\setminus$ is set-complementation. In particular, the following proposition holds.

Proposition 3.2.1. *For any set S, the tuple $(\mathcal{P}\,(S \times S), \mathcal{P}\,Id_S, \cup, ;, \emptyset, Id_S, {}^*, \overline{\ })$ forms a Kleene algebra with tests—the* full relation KAT *over S.*

The isomorphism $\mathcal{F}$ between relations and state transformers also provides insight into the state transformer model of KATs. By applying it to $\lceil - \rceil_{\mathcal{R}}$, we can lift predicates $P : S \to \mathbb{B}$ to this semantics

$$\lceil P \rceil_{\mathcal{F}} = \mathcal{F}\,\lceil P \rceil_{\mathcal{R}}\,s = \begin{cases} \{s\} & \text{if } P\,s, \\ \emptyset, & \text{otherwise.} \end{cases}$$

An analogue process applied to complementation yields the negation in the state transformer semantics. That is, for $f : S \to \mathcal{P}\,S$ such that $f\,s \subseteq \{s\}$ for all $s \in S$,

$$\overline{f}\,s = \mathcal{F}\,\overline{\mathcal{R}\,f}\,s = \begin{cases} \{s\} & \text{if } f\,s = \emptyset, \\ \emptyset, & \text{otherwise.} \end{cases}$$

The underlying set for this boolean operation is $\{f : S \to \mathcal{P}\, S \mid \forall s.\ f\, s \subseteq \{s\}\}$ which is the downwards closure $\downarrow(\mathcal{P}\, S)^S\, \eta_S$, where $\downarrow Y\, x = \{y \in Y \mid y \leq x\}$. The emerging boolean algebra is therefore $(\downarrow(\mathcal{P}\, S)^S\, \eta_S, \cup, ;, \lambda s.\ \emptyset, \eta_S, \overline{\ })$ and the following proposition is true.

Proposition 3.2.2. *For any set S, the tuple $((\mathcal{P}\, S)^S, \downarrow(\mathcal{P}\, S)^S\, \eta_S, \cup, ;, \lambda s.\ \emptyset, \eta_S, {}^{*_K}, \overline{\ })$ forms a Kleene algebra with tests—the* full state transformer KAT *over S.*

To avoid polluting notation with the isomorphisms $\mathbb{B}^S \cong \mathcal{P}\, S \cong \mathcal{P}\, Id_S \cong \downarrow(\mathcal{P}\, S)^S\, \eta_S$, we will freely identify elements of these sets and refer to them collectively as predicates.

KATs are already expressive enough to capture within their operations simple algebraic semantics for while programs. That is, we can define for $p \in B$ and $\alpha, \beta \in K$

$$\begin{aligned} \mathbf{skip} &= 1, \\ \mathbf{abort} &= 0, \\ \mathbf{if}\ p\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta &= p \cdot \alpha + \neg p \cdot \beta, \\ \mathbf{while}\ p\ \mathbf{do}\ \alpha &= (p \cdot \alpha)^* \cdot \neg p. \end{aligned}$$

Beyond that, KATs also cover the propositional part—disregarding assignments—of Hoare logic [84]. We encode Hoare triples $\{p\}\, \alpha\, \{q\}$ in KATs through any of the following equivalent assertions for $p, q \in B$ and $\alpha \in K$

$$p \cdot \alpha \cdot \neg q = 0 \quad \leftrightarrow \quad p \cdot \alpha \leq \alpha \cdot q \quad \leftrightarrow \quad p \cdot \alpha = p \cdot \alpha \cdot q.$$

Semantically, Hoare triples indicate if a predicate holds after the execution of a program that started satisfying some initial condition. Below we write the semantic representation of this statement in both the relational and state transformer model,

$$\begin{aligned} \{P\}\, R\, \{Q\} &\leftrightarrow (\forall s_1.\ P\, s_1 \to (\forall s_2.\ s_1\ R\ s_2 \to Q\, s_2)), \\ \{P\}\, f\, \{Q\} &\leftrightarrow (\forall s_1.\ P\, s_1 \to (\forall s_2.\ s_2 \in f\, s_1 \to Q\, s_2)). \end{aligned}$$

The proofs of these facts are automatic by unfolding definitions in our formalisations (as an example see lemma *wp-rel* in Section 4.5). The definition of Hoare triples allows us to derive the following implications in KATs:

$$\begin{aligned} p_1 \leq p_2 \wedge \{p_2\}\, \alpha\, \{q_2\} \wedge q_2 \leq q_1 &\to \{p_1\}\, \alpha\, \{q_1\}, && \text{(h-cons)} \\ \{p\}\, \alpha\, \{r\} \wedge \{r\}\, \beta\, \{q\} &\to \{p\}\, \alpha\, ;\, \beta\, \{q\}, && \text{(h-seq)} \\ \{t\, ;\, p\}\, \alpha\, \{q\} \wedge \{\neg t\, ;\, p\}\, \beta\, \{q\} &\to \{p\}\, \mathbf{if}\ t\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta\, \{q\}, && \text{(h-cond)} \\ \{t\, ;\, p\}\, \alpha\, \{p\} &\to \{p\}\, \mathbf{while}\ t\ \mathbf{do}\ \alpha\, \{\neg t\, ;\, p\}. && \text{(h-while)} \end{aligned}$$

These derivations describe the rules of inference of propositional Hoare logic. However, we can go beyond this calculus and obtain rules for other operations. For instance, by defining $\mathbf{loop}\ \alpha = \alpha^*$, we get

$$\begin{aligned} \{p\}\, \mathbf{skip}\, \{p\}, && \text{(h-skip)} \\ \{p\}\, \mathbf{abort}\, \{q\}, && \text{(h-abort)} \\ \{p\}\, \alpha\, \{q\} \wedge \{p\}\, \beta\, \{q\} &\to \{p\}\, \alpha + \beta\, \{q\}, && \text{(h-choice)} \\ \{p\}\, \alpha\, \{p\} &\to \{p\}\, \mathbf{loop}\ \alpha\, \{p\}. && \text{(h-loop)} \end{aligned}$$

We can interpret these results in a forward or backward-style reasoning. In the traditional forward reading, rule (h-choice) says that if two programs share a postcondition after starting in the same precondition, then a choice between these two programs will still respect the pre and postcondition. For interactive verification in a proof assistant, the backward reasoning style is more common. The characteristic of this style is its reading in terms of *proof obligations*. For example, the rule (h-loop) says that in order to prove that the loop of a program preserves certain property, it is enough to prove that a single iteration does so. We can automate this backward style reasoning to iteratively obtain proof obligations. That is, the rules of Hoare logic derived so far enable us to do *verification condition generation*.

Example 3.2.1 (`thermostat`'s control). Recall from Figures 2.3 and 2.5 that a simple thermostat can be modelled via the hybrid program `thermostat` $=$ **loop** (`ctrl` ; `dyn`) where

$$\begin{aligned}\texttt{ctrl} = \ & (t := 0)\,;(T_0 := T);\\ & \textbf{if } \theta = 0 \land T_0 \leq T_m + 1 \textbf{ then } \theta := 1 \textbf{ else}\\ & \textbf{if } \theta = 1 \land T_0 \geq T_M - 1 \textbf{ then } \theta := 0 \textbf{ else skip}.\end{aligned}$$

In this example, we focus on the discrete part of this hybrid program, leaving the continuous part for Example 4.4.1. For the time being, we also assume that assignments correspond to some atomic programs $\alpha_i \in K$ with $i \in \{1,2,3,4\}$ that do not modify the program store. We remove this assumption in Example 4.3.1. Therefore, under this assumption, Kleene algebras model all the program operations in `ctrl`. To prove the correctness specification

$$\{T_m \leq T \leq T_M\}\,\texttt{ctrl}\,\{T_m \leq T \leq T_M\},$$

we only need to apply backwardly the rules of Hoare logic. After two applications of (h-seq), where we leave r equal to the precondition, we obtain the proof obligations

$$\{T_m \leq T \leq T_M\}\,t := 0\,\{T_m \leq T \leq T_M\} \text{ and } \{T_m \leq T \leq T_M\}\,T_0 := T\,\{T_m \leq T \leq T_M\},$$

which we will hold as true due to the fact that the variable T is not being changed in the assignments. Thus, we can discharge these branches of the proof tree and consider them solved. The third branch corresponds to the specification

$$\begin{aligned}&\{T_m \leq T \leq T_M\}\\ &\textbf{if } \theta = 0 \land T_0 \leq T_m + 1 \textbf{ then } \theta := 1 \textbf{ else}\\ &\textbf{if } \theta = 1 \land T_0 \geq T_M - 1 \textbf{ then } \theta := 0 \textbf{ else skip}\\ &\{T_m \leq T \leq T_M\}.\end{aligned}$$

To tackle this branch, we apply twice the rule (h-cond), generating three proof obligations.

$$\begin{aligned}\{\theta = 0 \land T_0 \leq T_m + 1 \land T_m \leq T \leq T_M\}\,\theta := 1\,\{T_m \leq T \leq T_M\}\\ \{\theta = 1 \land T_0 \geq T_M - 1 \land \neg(\theta = 0 \land T_0 \leq T_m + 1) \land T_m \leq T \leq T_M\}\,\theta := 0\,\{T_m \leq T \leq T_M\}\\ \{\neg(\theta = 1 \land T_0 \geq T_M - 1) \land \neg(\theta = 0 \land T_0 \leq T_m + 1) \land T_m \leq T \leq T_M\}\,\theta := 0\,\{T_m \leq T \leq T_M\}\end{aligned}$$

Given that the three branches are true due to our assumption on assignments, this shows that if we execute `ctrl` in a state satisfying $T_m \leq T \leq T_M$, then this property will remain true after the execution. □

An additional benefit of the KAT setting is that it allows us to study invariants for programs. These are closely related to invariant sets for dynamical systems as we will see in Section 5.2. An *invariant* for $\alpha \in K$ is a test $i \in B$ such that $\{i\}\,\alpha\,\{i\}$. We can annotate these invariants in partial correctness specifications by defining α **inv** $i = \alpha$. These annotations are useful for automating the verification condition generation because the proof assistant does not need to come up with the invariant and instead can use it directly from the specification by applying the following rules.

$$p \leq i \wedge \{i\}\,\alpha\,\{i\} \wedge i \leq q \;\rightarrow \{p\}\,\alpha \textbf{ inv } i\,\{q\}, \tag{h-inv}$$
$$\{i\}\,\alpha\,\{i\} \wedge \{j\}\,\alpha\,\{j\} \rightarrow \{i\,;j\}\,\alpha\,\{i\,;j\}, \tag{h-conj-inv}$$
$$\{i\}\,\alpha\,\{i\} \wedge \{j\}\,\alpha\,\{j\} \rightarrow \{i+j\}\,\alpha\,\{i+j\}, \tag{h-disj-inv}$$
$$p \leq i \wedge \{i\,;t\}\,\alpha\,\{i\} \wedge \neg t\,;i \leq q \;\rightarrow \{p\}\textbf{ while } t \textbf{ do } \alpha \textbf{ inv } i\,\{q\}, \tag{h-while-inv}$$
$$p \leq i \wedge \{i\}\,\alpha\,\{i\} \wedge i \leq q \;\rightarrow \{p\}\textbf{ loop } \alpha \textbf{ inv } i\,\{q\}. \tag{h-loop-inv}$$

The rule (5.2.4) is a variant of (h-cons). We use it together with (h-conj-inv) and (h-disj-inv) for a procedure to prove invariance for evolution commands in Section 5.2. The rule (h-loop-inv) is the analogue for loops to the standard (h-while-inv) for while loops.

Despite all this added expressiveness, when proving in a backward style, the rules of Hoare logic still require input from the user as in (h-seq). Yet, this is not necessary with equational reasoning and Dijkstra's weakest precondition calculus [12]. KATs however cannot express predicate transformer semantics like weakest preconditions [128]. Therefore, we explore another algebra that can in the next section.

For a formalisation of KATs in Isabelle/HOL, see [8]. It includes preliminary algebraic structures, their derivable lemmas, and their most important models. The invariant perspective for KATs, its formalisation and that of the state transformer model in Section 3.5 are an addendum to the knowledge on these algebras from our work [44, 68].

3.3 Modal Kleene Algebra

There are other extensions of Kleene algebras for verification. By adding modal box and diamond operators like those of dynamic logic, the resulting algebras can encode not only tests and assertions, but predicate transformers too. In this section, we outline these alternative extensions of Kleene algebras for modal reasoning in the style of dynamic logic. Just like KATs capture the propositional part of Hoare logic (without assignments), these extensions capture propositional dynamic logic [58].

The first extension consists in adding an *antidomain operation* [35, 55] $ad : K \rightarrow K$ that satisfies the axioms

$$ad\,\alpha\,;\alpha = 0, \qquad ad\,\alpha + d\,\alpha = 1, \qquad ad(\alpha\,;\beta) \leq ad(\alpha\,;d\,\beta),$$

for all $\alpha, \beta \in K$ where $d = ad^2 = ad \circ ad$ is the *domain operation*. The resulting structure $(K, +, \cdot, 0, 1, ^*, ad)$ is an *antidomain Kleene algebra*.

In the interpretation of Kleene algebras as programs, the antidomain of a program $ad\,\alpha$ models those states from which α cannot be executed. In turn, the domain of a program

$d\,\alpha$ is the set of those states from which it can. Thus, the first two equations state that a program cannot run after its antidomain and that both operations complement each other in the boolean sense. The final inequality states that the antidomain of a composition should not be greater than that of the domain of the second composite following the first one.

From these axioms, one can check that in antidomain Kleene algebras the equality $d^2 = d$ holds [54], hence $p \in (\mathcal{P}\ ad\ K) \leftrightarrow (d\,p = p)$. Thus, the image $\mathcal{P}\ ad\ K = K_d$, where K_f is the set of fixpoints of $f : K \to K$, that is $K_f = \{\alpha \in K \mid f\,\alpha = \alpha\}$. Furthermore, $(K_d, +, ;, 0, 1, ad)$ forms a boolean algebra, and therefore, our definitions, notation and results for KATs of Section 3.2 are also available for these algebras.

In antidomain Kleene algebras, we can define *forward modal box* and *diamond* operators $|-]- : K \to K_d \to K_d$ and $|-\rangle - : K \to K_d \to K_d$ such that

$$|\alpha]\,p = ad(\alpha\,;\,ad\,p) \qquad \text{and} \qquad |\alpha\rangle\,p = d\,(\alpha\,;\,p).$$

They also satisfy the *De Morgan duality laws*

$$|\alpha]\,p = \neg\,|\alpha\rangle\,\neg p \qquad \text{and} \qquad |\alpha\rangle\,p = \neg\,|\alpha]\,\neg p,$$

where we have used the boolean negation $\neg$ in replacement of ad.

However, another important duality of Kleene algebras is *opposition*, that is, swapping the factors in multiplications. In fact, the class of Kleene algebras is closed under opposition, meaning that swapping the order of multiplication in a Kleene algebra generates another one. The opposite of the antidomain operation is the *antirange operation* $ar : K \to K$ that represents those states into where a program cannot be executed [35]. For further intuition, in the relational model, the antirange maps a program (relation) to the complement of its range. It is dually axiomatised via

$$\alpha\,;\,ar\ \alpha = 0, \qquad ar\ \alpha + r\,\alpha = 1, \qquad ar\,(\alpha\,;\,\beta) \leq ar\,(r\,\alpha\,;\,\beta),$$

for all $\alpha, \beta \in K$. Again, $r = ar^2 = ar \circ ar$ is the *range operation* that describes those output states into where a program can be executed. As before, Kleene algebras extended with ar produce *antirange Kleene algebras* that allow us to define *backward modal operators*,

$$[\alpha|\,p = ar(ar\,p\,;\,\alpha) \qquad \text{and} \qquad \langle\alpha|\,p = r\,(p\,;\,\alpha),$$

for $\alpha \in K$, $p \in K_d$, and derive their corresponding De Morgan duality laws

$$[\alpha|\,p = \neg\,\langle\alpha|\,\neg p \qquad \text{and} \qquad \langle\alpha|\,p = \neg\,[\alpha|\,\neg p.$$

We can further extend antidomain and antirange Kleene algebras by adding their opposite operators, thus obtaining *modal Kleene algebras* (MKA) with structures given by tuples $(K, +, ;, 0, 1, ^*, ad, ar)$ [35]. Observe that an axiomatisation of MKAs depending only on domain and range would not be as rich because it would lack complementation, thus making K_d a simple distributive lattice. Notice also that at this point, modal Kleene algebras have enough expressive power to reason not only as a Hoare logic but as a dynamic logic. Moreover, the above backward modalities are not available in other work for verification of hybrid systems. Albeit, our interest in them remains as a theoretical fact and we leave their

application in case studies for future work. See the formalisation [54] in the Archive of Formal Proofs that predates our work for a complete development of algebraic structures that starts from domain and antidomain semigroups, passing through semirings, and ending up with MKAs. Said formalisation also contains the calculational properties of those structures and their most important models.

Rich relationships emerge from adding antidomain and antirange operations to Kleene algebras. For starters, $K_d = K_r = \mathcal{P}\ ad\ K = \mathcal{P}\ d\ K = \mathcal{P}\ r\ K = \mathcal{P}\ ar\ K$. Thus, MKAs are closed under opposition. Furthermore, the conjugation laws

$$|\alpha]\, p + q = 1 \leftrightarrow p + [\alpha|\, q = 1 \quad \text{and} \quad |\alpha\rangle\, p\,;\, q = 0 \leftrightarrow p\,;\, \langle\alpha|\, q = 0$$

for $p, q \in K_d$ and $\alpha \in K$ are derivable, as well as the Galois connections

$$\langle\alpha|\, p \leq q \leftrightarrow p \leq |\alpha]\, q \quad \text{and} \quad |\alpha\rangle\, p \leq q \leftrightarrow p \leq [\alpha|\, q.$$

Finally, in MKAs, Hoare triples $\{p\}\,\alpha\,\{q\}$ have another representation equivalent to the other three of KATs, namely, the implications $p \leq |\alpha]\, q$.

Every concept and result introduced in this section is valid for the relational and state transformer models. Opposition, for instance, corresponds to the *converse* relation $(-)^\smile$ defined by $R^\smile = \{(s_2, s_1) \mid s_1\ R\ s_2\}$. The antidomain and antirange operations are then

$$ad_{\mathcal{R}}\, R = \{(s_1, s_1) \mid \neg\, \exists s_2.\ s_1\ R\ s_2\} \quad \text{and} \quad ar_{\mathcal{R}}\, R = ad_{\mathcal{R}}\, R^\smile = \{(s_2, s_2) \mid \neg\, \exists s_1.\ s_1\ R\ s_2\},$$

while $d_{\mathcal{R}}\, R$ and $r_{\mathcal{R}}\, R$ correspond to the domain and range of R respectively. From here, forward diamond and box operators equate to

$$|R]\, P = \{(s_1, s_1) \mid \forall s_2.\ s_1\ R\ s_2 \rightarrow P\, s_2\} \quad \text{and} \quad |R\rangle\, P = \{(s_1, s_1) \mid \exists s_2.\ s_1\ R\ s_2 \wedge P\, s_2\},$$

and their backward versions come from opposition $[R|\, P = |R^\smile]\, P$ and $\langle R|\, P = |R^\smile\rangle\, P$. In summary, the result below holds.

Proposition 3.3.1. *For any set S, the tuple $(\mathcal{P}\,(S \times S), \mathcal{P}\ Id_S, \cup, ;, \emptyset, Id_S, ^*, ad_{\mathcal{R}}, ar_{\mathcal{R}})$ forms a modal Kleene algebra—the* full relation MKA *over S.*

We can derive the state transformer model from our now familiar isomorphism $\mathcal{F}$ and derive analogous results. It is easy to check that $f^{op} = \lambda s_2.\ \{s_1 \mid s_2 \in f\, s_1\}$ satisfies $f^{op} = \mathcal{F}\,(\mathcal{R}\, f)^\smile$. Hence, $ar_{\mathcal{F}}\, f = ad_{\mathcal{F}}\, f^{op}$, where

$$ad_{\mathcal{F}} f\ s = \begin{cases} \{s\}, & \text{if } f\, s = \emptyset, \\ \emptyset, & \text{otherwise.} \end{cases}$$

For state transformers $f : S \rightarrow \mathcal{P}\, S$ below η_S, that is $f \in {\downarrow}(\mathcal{P}\, S)^S\, \eta_S$, the expression $ad_{\mathcal{F}} f$ coincides with $\overline{f}$ of Section 3.2. The forward diamond and box operators follow

$$|f\rangle\, P\, s = \begin{cases} \{s\}, & \text{if } f\, s \cap \{s \mid P\, s\} \neq \emptyset, \\ \emptyset, & \text{otherwise,} \end{cases} \quad \text{and} \quad |f]\, P\, s = \begin{cases} \{s\}, & \text{if } f\, s \subseteq \{s \mid P\, s\}, \\ \emptyset, & \text{otherwise,} \end{cases}$$

while the backward modal operators are $\langle f|\, P = |f^{op}\rangle\, P$ and $[f|\, P = |f^{op}]\, P$ as before. Similar computations can be carried out to describe $d_{\mathcal{F}}$ and $r_{\mathcal{F}}$. Abusing notation, the set of all states satisfying $\langle f|\, P$ is $\langle f|\, P = f^{\dagger}\, P$, where $(-)^{\dagger}$ is the Kleisli extension defined in Section 3.1. This specific equality will be useful as a definition of invariants for systems of ODEs in Section 5.2 and as the cornerstone of an alternative approach for verification components in Sections 6.2 and 6.3. To conclude, the foretold proposition is true.

Proposition 3.3.2. *For any set S, the tuple $((\mathcal{P}\, S)^S, \downarrow(\mathcal{P}\, S)^S\, \eta_S, \cup, ;, \lambda s.\, \emptyset, \eta_S, ^{*_K}, ad_{\mathcal{F}}, ar_{\mathcal{F}})$ forms a modal Kleene algebra—the* full state transformer MKA *over S.*

The main advantage of MKAs over KATs is their modal operators. These modalities are endofunctions $K_d \to K_d$ on the boolean algebra K_d, and therefore, they agree with Jónsson and Tarski's boolean algebras with operators [79]. However, using the isomorphisms $\mathbb{B}^S \cong \mathcal{P}\, S \cong \mathcal{P}\, Id_S \cong \downarrow(\mathcal{P}\, S)^S\, \eta_S$, the modalities $|\alpha]\, -, [\alpha|\, -, |\alpha\rangle\, -, \langle\alpha|\, - : \mathbb{B}^S \to \mathbb{B}^S$ are also functions from predicates to predicates, that is, they are *predicate transformers*. In fact, in a partial correctness setting, the forward box $|-]\, -$ and backward diamond $\langle -|\, -$ of MKA correspond to Dijkstra's *weakest liberal precondition* (*wlp*) and *strongest postcondition* operators respectively. In other words, $|\alpha]\, q$ is the weakest precondition p of the Hoare triple $\{p\}\, \alpha\, \{q\}$ in the sense that every other p implies it, $p \leq |\alpha]\, q$. Similarly, $\langle\alpha|\, p$ implies every other q such that $\{p\}\, \alpha\, \{q\}$. Based on this, verification of the correctness specification $p \leq |\alpha]\, q$ means recursively computing $|\alpha]\, q$ over the structure of α and checking that the result is greater or equal to p. For this purpose, the rules of the *wlp* calculus (without assignments) are derivable for $p, q, i, j, t \in K_d$ and $\alpha, \beta \in K$ [55, 56]

$$|\mathbf{skip}]\, q = q, \tag{wlp-skip}$$
$$|\mathbf{abort}]\, q = 1, \tag{wlp-abort}$$
$$|\alpha\, ;\, \beta]\, q = |\alpha]\, |\beta]\, q, \tag{wlp-seq}$$
$$|\mathbf{if}\ t\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]\, q = (t\, ;\, |\alpha]\, q) + (\neg t\, ;\, |\beta]\, q), \tag{wlp-cond}$$
$$p \leq |\mathbf{while}\ t\ \mathbf{do}\ \alpha\ \mathbf{inv}\ i]\, q \leftarrow p \leq i \wedge i\, ;\, t \leq |\alpha]\, i \wedge i\, ;\, \neg t \leq q, \tag{wlp-while}$$

and as before, MKA also derives properties about the rest of its operations and invariants

$$|\alpha + \beta]\, q = |\alpha]\, q\, ;\, |\beta]\, q, \tag{wlp-choice}$$
$$p \leq i \wedge i \leq |\alpha]\, i \wedge i \leq q \to p \leq |\mathbf{loop}\ \alpha\ \mathbf{inv}\ i]\, q, \tag{wlp-loop}$$
$$i \leq |\alpha]\, i \wedge j \leq |\alpha]\, j \to (i\, ;\, j) \leq |\alpha](i\, ;\, j), \tag{wlp-conj}$$
$$i \leq |\alpha]\, i \wedge j \leq |\alpha]\, j \to (i + j) \leq |\alpha](i + j). \tag{wlp-disj}$$

From these rules, it follows that, with equational reasoning, we can compute $|\alpha]\, q$ for regular programs except for loops, which require annotated invariants. Therefore, in our formalisations we can write simple tactics for verification condition generation. For instance, a tactic could apply these rules and translate the resulting MKA statement without modalities to its relational or state transformer model. Thus, it would make proof obligations become domain-specific problems that are also easily solvable.

Example 3.3.1. Here, we partially compute the weakest precondition for the program `ctrl` of Example 3.2.1. Recall that it is part of the `thermostat` hybrid program, and that it is

$$\begin{aligned}\texttt{ctrl} = {} & (t := 0)\,;(T_0 := T);\\ & \textbf{if } t_1 \textbf{ then } \theta := 1 \textbf{ else}\\ & \textbf{if } t_2 \textbf{ then } \theta := 0 \textbf{ else skip},\end{aligned}$$

where, $t_1 \leftrightarrow (\theta = 0 \wedge T_0 \leq T_m + 1)$ and $t_2 \leftrightarrow (\theta = 1 \wedge T_0 \geq T_M - 1)$. Therefore, we use the *wlp*-laws (wlp-seq) and (wlp-cond) to recursively compute part of its *wlp*.

$$\begin{aligned}|\texttt{ctrl}]\,p &= |t := 0]\,|T_0 := T\,;\textbf{if } t_1 \textbf{ then } \theta := 1 \textbf{ else if } t_2 \textbf{ then } \theta := 0 \textbf{ else skip}]\,p\\ &= |t := 0]\,|T_0 := T]\,|\textbf{if } t_1 \textbf{ then } \theta := 1 \textbf{ else if } t_2 \textbf{ then } \theta := 0 \textbf{ else skip}]\,p\\ &= |t := 0]\,|T_0 := T]\,(t_1\,;|\theta := 1]\,p + \neg t_1\,;|\textbf{if } t_2 \textbf{ then } \theta := 0 \textbf{ else skip}]\,p)\\ &= |t := 0]\,|T_0 := T]\,(t_1\,;|\theta := 1]\,p + \neg t_1\,;(t_2\,;|\theta := 0]\,p + \neg t_2\,;|\textbf{skip}]\,p))\\ &= |t := 0]\,|T_0 := T]\,(t_1\,;|\theta := 1]\,p + \neg t_1\,;t_2\,;|\theta := 0]\,p + \neg t_1\,;\neg t_2\,;p)\end{aligned}$$

Semantically, the argument of the leftmost forward box operations correspond to operations between subidentity relations or state transformers below η_S. The reason for this is that t_1, t_2, $|\theta := 0]\,p$, $|\theta := 1]\,p$, and their negations are elements of the respective boolean algebra. Therefore, we only need to provide a *wlp*-rule for assignments in order to completely obtain the weakest precondition for `ctrl`. We do this in Section 4.3 of next chapter. □

The equational reasoning, the dualities by opposition, the De Morgan, conjugation and Galois connection laws generalise to predicate transformers of type $\mathbb{B}^{S_1} \to \mathbb{B}^{S_2}$. Developing this variant for verification is the topic of Section 6.3. There are many more interesting facts about the modal operators of MKA that deviate from our interest in verification condition generation. For a comprehensive list, the reader can see the AFP entry [54]. Furthermore, the formalisation in Isabelle/HOL of the *wlp* calculus through MKAs precedes our work and is also available in the AFP [56]. However, the formalisation of the invariant rules of MKA and its state transformer model are a contribution from our work [68, 72].

3.4 Algebraic Structures in Isabelle/HOL

In this section, we describe some properties of type classes in Isabelle/HOL. We show their usage as a mechanism to formalise algebraic structures and we explore two ways to make types inherit the lemmas and constants of classes. Achieving this inheritance consists in showing that the type satisfies the axioms of the algebraic structure that the class represents.

Type classes allow us to define constants and operations once. Then, we can use them among many types through type polymorphism, like in functional programming. Each type class can introduce not only constants, but axioms for them too. Classes also have a context delimited with the keywords **begin** and **end** where we can derive consequences from those axioms. The code below formalises mathematical semigroups with a type class.

```
class semigroup =
  fixes mult :: 'a ⇒ 'a ⇒ 'a (infixl · 70)
  assumes mult-assoc: (α · β) · γ = α · (β · γ)
begin

lemma four-assoc: ((α · β) · γ) · δ = α · (β · (γ · δ))
```

by (*simp add*: *mult-assoc*)

end

The name of the class, *semigroup*, must follow the command **class** to indicate Isabelle that we are about to define it. After the equal symbol, we provide the class' constants with the keyword **fixes** and its axioms with **assumes**. The second line states that the operation *mult* is a function of type $'a \Rightarrow 'a \Rightarrow 'a$. The expression in parenthesis uses keyword **infixl** to instruct that infix centred dots $\cdot$ should be parsed as *mult*. The third line introduces axiom *mult-assoc*, which is simply associativity of the binary operation *mult*. Similarly, the command **lemma** introduces the theorem called *four-assoc* that says that for any four elements $\alpha, \beta, \gamma, \delta$ of the semigroup, the equality $((\alpha \cdot \beta) \cdot \gamma \cdot \delta) = \alpha \cdot (\beta \cdot (\gamma \cdot \delta))$ holds. Finally, we use the command **by** to prove the lemma with a call to Isabelle's simplifier augmented with the axiom *mult-assoc* of the *semigroup* class. Classes can also extend other classes. To declare the class of monoids that extend semigroups, the structure is the same, but with the name *semigroup* following the equality symbol.

class *monoid* = *semigroup* +
  fixes $e :: 'a$
  assumes *mult-idr*: $\alpha \cdot e = \alpha$
    and *mult-idl*: $e \cdot \alpha = \alpha$

Classes in Isabelle/HOL allow us to formalise algebraic structures because we can add axioms to their declaration. This is indeed an easy way to introduce inconsistencies inside the proof assistant although it is only limited to consequences of the class' axioms. The rest of the object logic is unaffected by them. As in mathematics, they are made consistent by providing a model to them. This is done with the **instantiation** command. With it, types can inherit the constants of the class together with the lemmas for those constants. This requires users to supply a proof that the type satisfies the axioms of the class via the **instance** command. Previous to that, users also have to indicate which of the type's operations correspond to the class' operations. They can do this with the command **lift-definition**. The code below exemplifies an instantiation of the type of integers (*int*) to the *monoid* class.

instantiation *int*::*monoid*
begin

lift-definition *mult-int* :: *int* $\Rightarrow$ *int* $\Rightarrow$ *int* **is** $(*)$ **.**

lift-definition *e-int* :: *int* **is** *1* **.**

instance — *OFCLASS*(*int*, *semigroup-add-class*)
  apply *intro-classes* — $\bigwedge \alpha\ \beta\ \gamma.\ \alpha \cdot \beta \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$
  apply *transfer* — $\bigwedge \alpha\ \beta\ \gamma.\ \alpha * \beta * \gamma = \alpha * (\beta * \gamma)$
  apply *simp* — $\bigwedge \alpha.\ \alpha \cdot e = \alpha$
  by (*transfer*, *simp*)+

end

Here, we supply the default integer multiplication ($*$) as an argument to **lift-definition** to represent the class multiplication (*mult*), and similarly for 1 and e. For a better explanation of the **instance** proof, we have added next to each line the proof-state that Isabelle displays, indicated after the long dash — . Thus, after the **instance** command, Isabelle asks to prove the instantiation. Then, tactic *intro-classes* transforms this proof obligation into other three, the first being associativity for ($\cdot$) in the integers. After that, applying the *transfer* tactic transforms the first proof obligation to an obligation about integers and their multiplicaton ($*$). This is such an easy obligation, that Isabelle can handle it automatically with its simplifier. Thus, the next class axiom *mult-idr* needs to be proved. Finally, we use a plus symbol + to automatically repeat the previous process for each remaining axiom.

After this instantiation, we can freely use ($\cdot$) with integers in Isabelle. Furthermore, all the lemmas in the class *monoid* are available for this type too. For instance, the following result requires calling the lemma *four-assoc* of our *semigroup* class, as it just restates it.

lemma $(((a::int) \cdot b) \cdot c) \cdot d = a \cdot (b \cdot (c \cdot d))$
using *semigroup-class.four-assoc* **.**

However, the analogous result $(((a::int) * b) * c) * d = a * (b * (c * d))$ has a longer proof that requires the *transfer* tactic to convert from the class operation to the type operation. This is because class instantiations only provide the class' lemmas for their designated constants. To obtain instantiated lemmas for the type operations we require interpretations.

interpretation *int-semigroup*: *semigroup* $(*)$
by *standard simp*

lemma $(((a::int) * b) * c) * d = a * (b * (c * d))$
using *int-semigroup.four-assoc* **.**

In the code above, we use the command **interpretation** followed by the name of the interpretation *int-semigroup*, a colon, the name of the class to interpret and the type operation to instantiate. In this case, the latter two are the *semigroup* class and the multiplication of integers ($*$). Then, after proving that the type satisfies the axioms of the class, the class' lemmas are automatically instantiated. Their names consist of the name of the interpretation followed by a dot and the name of the original class lemma, for instance, *int-semigroup.four-assoc*.

3.5 Kleene Algebras in Isabelle/HOL

Here, we describe the formalisation of the concepts presented in previous sections of this chapter. The objective is to list the rules for verification condition generation as they appear in our AFP submission [68]. Therefore, the material discussed in this section, becomes the

algebraic foundations for developing a hybrid system verification component with the proof assistant. More specifically, for illustration purposes, we present an example formalisation of Kleene algebras in Isabelle/HOL via type classes. However, such example is not the actual content of [10] nor [54] that are the AFP entries that precede our work and on top of which we developed our verification components. Despite not using the light-weight version presented here, it displays the essence of the basis for our work. We also explain in this section our extensions to Isabelle/HOL's libraries for program algebras. In particular, we contribute by implementing rules for verification condition generation that were previously unavailable in existing formalisations.

The code below uses type classes to model Kleene algebras.

class *plus-ord* = *plus* + *ord* +
  assumes *leq-def*: $x \leq y \longleftrightarrow x + y = y$
    and *less-def*: $x < y \longleftrightarrow \neg\ (y \leq x)$

class *dioid* = *comm-monoid-add* + *monoid* + *plus-ord* +
  assumes *add-idem*: $\alpha + \alpha = \alpha$
    and *distribl*: $\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma$
    and *distribr*: $(\alpha + \beta) \cdot \gamma = \alpha \cdot \gamma + \beta \cdot \gamma$
    and *absorpl*: $\alpha \cdot 0 = 0$
    and *absorpr*: $0 \cdot \alpha = 0$

class *kleene-algebra* = *dioid* +
  fixes *kstar* :: $'a \Rightarrow 'a$ $(\text{-}^\star\ [101]\ 101)$
  assumes *unfoldl*: $e + \alpha \cdot \alpha^\star \leq \alpha^\star$
    and *unfoldr*: $e + \alpha^\star \cdot \alpha \leq \alpha^\star$
    and *inductl*: $\gamma + \alpha \cdot \beta \leq \beta \Longrightarrow \alpha^\star \cdot \gamma \leq \beta$
    and *inductr*: $\gamma + \beta \cdot \alpha \leq \beta \Longrightarrow \gamma \cdot \alpha^\star \leq \beta$

The presentation is as discussed in previous sections: in *plus-ord*, we can define a relation ($\leq$) using the nondeterministic choice ($+$). Afterwards, the class *dioid* extends additive and multiplicative monoids as well as *plus-ord* with the idempotency, distributivity and absorption axioms. Finally, the class *kleene-algebra* adds the remaining axioms. The full formalisation of Kleene algebras, KATs and MKAs, their substructures, and an analysis of the consequences for each added axiom are in the AFP [8, 10, 54]. We simply use those libraries to shallowly embed regular programs as elements of Kleene algebras and derive their rules for verification condition generation. Many of these rules are already available from the AFP libraries, for instance, below we restate the formalisation of (wlp-skip), (wlp-abort), (wlp-seq), and (wlp-choice) in the context of the class for antidomain Kleene algebras.

context *antidomain-kleene-algebra*
begin

— Skip
lemma $|1]\ x = d\ x$
using *fbox-one* **.**

— Abort
lemma $|0]\ q = 1$
using *fbox-zero* **.**

— Sequential composition
lemma $|x \cdot y]\ q = |x]\ |y]\ q$
using *fbox-mult* **.**

— Nondeterministic choice
lemma $|x + y]\ q = |x]\ q \cdot |y]\ q$
using *fbox-add2* **.**

end

The reader can compare the above rules with their analogs in previous sections and confirm that they are a faithful formalisation. In contrast with the *wlp*-rules above, we add the corresponding definitions and preliminary lemmas for (wlp-cond), (wlp-while), (wlp-loop), (wlp-conj), and (wlp-disj).

— Conditional statement
definition *aka-cond* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (*if - then - else -* [64,64,64] 63)
where *if p then x else y* $= d\ p \cdot x + ad\ p \cdot y$

lemma *fbox-cond* [*simp*]: $|$*if p then x else y*$]\ q = (ad\ p + |x]\ q) \cdot (d\ p + |y]\ q)$
using *fbox-export1 local.ans-d-def local.fbox-mult*
unfolding *aka-cond-def ads-d-def fbox-def* **by** *auto*

— While loop
definition *whilei* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (*while - do - inv -* [64,64,64] 63)
where *while t do x inv i* $= (d\ t \cdot x)^\star \cdot ad\ t$

lemma *fbox-whilei*:
assumes $d\ p \leq d\ i$ **and** $d\ i \cdot ad\ t \leq d\ q$ **and** $d\ i \cdot d\ t \leq |x]\ i$
shows $d\ p \leq |$*while t do x inv i*$]\ q$
⟨*proof*⟩

— Finite iteration
definition *aka-loopi* :: $'a \Rightarrow 'a \Rightarrow 'a$ (*loop - inv -* [64,64] 63)
where *loop x inv i* $= x^\star$

lemma *fbox-loopi*: $d\ p \leq d\ i \Longrightarrow d\ i \leq |x]\ i \Longrightarrow d\ i \leq d\ q \Longrightarrow d\ p \leq |$*loop x inv i*$]\ q$
unfolding *aka-loopi-def* **by** (*meson dual-order.trans fbox-iso fbox-star-induct-var*)

— Invariants
lemma *plus-inv*: $i \leq |x]\ i \Longrightarrow j \leq |x]\ j \Longrightarrow (i + j) \leq |x]\ (i + j)$
by (*metis ads-d-def dka.dsr5 fbox-simp fbox-subdist join.sup-mono order-trans*)

lemma *mult-inv*: $d\ i \leq |x]\ i \Longrightarrow d\ j \leq |x]\ j \Longrightarrow (d\ i \cdot d\ j) \leq |x]\ (d\ i \cdot d\ j)$
using *fbox-demodalisation3 fbox-frame fbox-simp* **by** *auto*

In the code above, we omit long proofs and replace them with an indicator $\langle proof \rangle$. For conditional statements and while loops, we follow the verification components for regular programs in the AFP [56]. Yet, the formalisation of finite iterations and invariants (last four lemmas) is our contribution. We have done a similar development for KATs and their derived rules for Hoare logic for [44]. However, the KAT construction in Isabelle/HOL via type classes is different from the presentation in this thesis [8]. Type classes only allow mono-typed constants and functions. This rules out the two-sorted approach with boolean algebras. The alternative then adds a mono-typed *antitest* function $n : K \to K$ to the Kleene algebra K. Using this operation, a *test* function $t : K \to K$ can be obtained via $t = n \circ n$. This is similar to our presentation of antidomain and domain operations. Then $K_t = \{\alpha \mid t\,\alpha = \alpha\}$ is the hidden boolean algebra inside the type class for KATs with n as its test complementation. We use this encoding of KATs in our work as well as the libraries of MKAs [54]. Our full formalisation work is also available in the AFP [68]. It covers all the formalisation sections in this thesis except for those in Section 6.4.

We still need to formalise this chapter's propositions 3.1.1, 3.1.2, 3.2.1, 3.2.2, 3.3.1 and 3.3.2, but all these results are analogous. Therefore, we only show here that relations and state transformers form antidomain Kleene algebras. For relations, we use an **interpretation** command whose proof is easy due to Isabelle's automation for sets.

definition *rel-ad* :: *'a rel* ⇒ *'a rel* **where**
rel-ad R = {(*x*,*x*) | *x*. ¬ (∃ *y*. (*x*,*y*) ∈ *R*)}

interpretation *rel-aka*: *antidomain-kleene-algebra rel-ad* (∪) (*O*) *Id* {} (⊆) (⊂) *rtrancl*
by *unfold-locales* (*auto simp*: *rel-ad-def*)

State transformers, on the other hand, are not a default type in Isabelle/HOL. Thus, we use their definition in [129] and do an instantiation proof for them. They are nondeterministic functions, *f* :: *'a nd-fun*. This type precisely corresponds with the set of all functions of type *'a* ⇒*'a set*. Therefore, we heavily use the bijections $f_\bullet$ and $f^\bullet$ between these types in our formalisation. Below, we show a small part of our instantiation proof with omissions marked with vertical dots.

typedef *'a nd-fun* = {*f* :: *'a* ⇒*'a set*. *f* ∈ *UNIV*}
by *simp*

instantiation *nd-fun* :: (*type*) *antidomain-kleene-algebra*
begin

definition *ad f* = (λ*x*. *if* ((*f*$_\bullet$) *x* = {}) *then* {*x*} *else* {})$^\bullet$

⋮

lemma *nd-fun-plus-assoc*[*nd-fun-aka*]: $x + y + z = x + (y + z)$
and *nd-fun-plus-comm*[*nd-fun-aka*]: $x + y = y + x$
and *nd-fun-plus-idem*[*nd-fun-aka*]: $x + x = x$ **for** $x::'a$ *nd-fun*
unfolding *plus-nd-fun-def* **by** (*simp add*: *ksup-assoc*, *simp-all add*: *ksup-comm*)

⋮

instance
apply *intro-classes*
using *nd-fun-aka* **by** *simp-all*

end

This formalisation of state transformer semantics for KATs and MKAs as nondeterministic functions is a contribution from our work.

The interpretation and instantiation proofs depicted in this section are crucial for the modularity of our approach. They not only serve to show that state transformers and relations form models for Kleene algebras, but also formalise the fact that the axioms are sound. It means that we can use the laws of KATs and MKAs with state transformers or relations in Isabelle/HOL. If at any point we prefer one over the other to build verification components, we can quickly and smoothly switch between them. In Section 6.6, we analyse and compare all the verification components that emerge from our formalisations.

At this point, we have formalised verification condition rules for regular and while programs. First, we provided the mathematical definition of Kleene algebras, Kleene algebras with tests, and modal Kleene algebras. Then, we defined in these algebras both regular and while programs. Moreover, we saw that these algebras also allow us to derive rules for verification condition generation. Finally, we formalised all these developments in the general-purpose proof assistant, Isabelle/HOL. Yet, looking at the syntax

$$X \ ::= \ x := e \mid x' = f \,\&\, G \mid ?P \mid X \,;\, X \mid X + X \mid X^*,$$

for hybrid programs, we are still missing rules for assignments and evolution commands. We formalise them within the concrete state transformer and relational semantics in Sections 4.3 and 4.4. However, in order to integrate these into our verification components, we take a slight detour and explore the relationship between dynamical systems, differential equations and state transformers.

Chapter 4

Hybrid Store Semantics

Just like state transformers model programs by associating to initial input states the set of all possible output states, they can also model physical systems by associating to a current state the set of all future states. However, exact analytical descriptions of future states are not always viable, which is why the preferred representations of physical phenomena through mathematical analysis are differential equations.

State transformers are related to differential equations through the intermediate mathematical concept of a *dynamical system*. These are functions $\varphi : T \to S \to S$ that model the time dependency of points in a state space S. To represent time, the set T has a monoid structure and φ is a *monoid action* on S. That is, the equations

$$\varphi\,(t_1 + t_2) = \varphi\,t_1 \circ \varphi\,t_2 \qquad \text{and} \qquad \varphi\,0 = id$$

hold for all $t_1, t_2 \in T$, where $0 \in T$ is the monoid's unit and $id : S \to S$ is the identity function. The first equation represents the inherent determinism in the passage of time, while the second, that our analysis of the system starts at 0. Usual monoids for dynamical systems are $\mathbb{Z}$, $\mathbb{R}$, and their nonnegative variants $\mathbb{N}$ and $\mathbb{R}_+$. If the monoid is at most countable, the function φ is a *discrete dynamical system*. Otherwise and with an underlying topological structure, φ is a *flow* or a *continuous dynamical system*. Flows emerge as solutions to certain systems of differential equations [11, 63, 131].

In Section 4.1, we describe in more detail the connection between state transformers, flows and differential equations. Then, we formalise these results in Section 4.2. At that point, we are ready to add the semantics of the two remaining hybrid programs. Thus, we discuss assignments in Section 4.3 and evolution commands in Section 4.4. Finally, we show the formalisation of the rules for verification condition generation for assignments in Section 4.5, leaving that for evolution commands for Chapter 5.

4.1 Ordinary Differential Equations

We briefly review the mathematical definitions and results for differential equations needed for developing verification components for hybrid systems.

An *ordinary differential equation* (ODE) $F\,(t, x\,t, x'\,t, \ldots, x^{(n)}\,t) = 0$ describes a relationship between a k-continuously differentiable function $x : T \to \mathbb{R}$ and $0 \in \mathbb{R}$

via a continuous function F, where $t \in T \subseteq \mathbb{R}$. A function satisfying the equation is a *solution* to it. By the implicit function theorem, we can locally solve for $x^{(n)}\, t$ and represent ODEs in their more familiar implicit form

$$x^{(n)}\, t = g\,(t, x\, t, x'\, t, \ldots, x^{(n-1)}\, t),$$

where $g : T \times S \to \mathbb{R}$ is continuous and $S \subseteq \mathbb{R}^n$. The number n of the highest derivative is the *order* of the ODE. Furthermore, we can turn any nth-order ODE into a system of n first-order ODEs by introducing new variables x_i with $1 \leq i \leq n-1$ such that

$$x'\, t = x_1\, t, \quad x_1'\, t = x_2\, t, \quad \ldots \quad x_{n-1}'\, t = g\,(t, x\, t, x_1\, t, \ldots, x_{(n-1)}\, t).$$

We can even provide a *time-independent* or *autonomous* related system of $n+1$ ODEs by adding the equation $x_0'\, t = 1$, thus obtaining $x_{n-1}'\, t = g\,(x_0\, t, x\, t, x_1\, t, \ldots, x_{(n-1)}\, t)$.

Given all these possible manipulations, for the rest of this work we will focus on first-order systems of ODEs,

$$\begin{aligned} x_1'\ t &= f_1\ (t, x_1\ t, \ldots, x_n\ t), \\ x_2'\ t &= f_2\ (t, x_1\ t, \ldots, x_n\ t), \\ &\vdots \\ x_n'\ t &= f_n\ (t, x_1\ t, \ldots, x_n\ t), \end{aligned}$$

for continuous functions $f_i : T \times S \to \mathbb{R}$ with $t \in T \subseteq \mathbb{R}$ and $S \subseteq \mathbb{R}^n$. Moreover, we also opt for the vector representation of these systems

$$X'\, t = \begin{pmatrix} x_1'\ t \\ x_2'\ t \\ \vdots \\ x_n'\ t \end{pmatrix} = \begin{pmatrix} f_1\ (t, x_1\ t, \ldots, x_n\ t) \\ f_2\ (t, x_1\ t, \ldots, x_n\ t) \\ \vdots \\ f_n\ (t, x_1\ t, \ldots, x_n\ t) \end{pmatrix} = f\,(t, X\, t),$$

where the function $f : T \times S \to \mathbb{R}^n$ is a *vector field*. That is, it assigns a vector of $\mathbb{R}^n$ to each point in $T \times S$. Therefore, an *initial value problem* (IVP) consists of a vector field $f : T \times S \to \mathbb{R}^n$ and an initial condition $(t_0, s) \in T \times S$. Then, a *solution to the system of ODEs* is a continuously differentiable function $X : T \to S$ such that $X'\, t = f\,(t, X\, t)$ for all $t \in T$. Such a function also *solves the IVP* if it satisfies $X\, t_0 = s$.

Example 4.1.1 (Particles in fluid). To model the three-dimensional motion of particles in a fluid, we can assign a velocity vector to each point in space. For instance, Figure 4.1 shows a graphical representation of the vector field described by the system of differential equations

$$x'\, t = v, \qquad y'\, t = 0, \qquad z'\, t = -\sin\,(x\, t),$$

where $v \in \mathbb{R} \setminus \{0\}$ is a nonzero constant, and the functions x, y, and z model the position of the particles in the three dimensional space.

The corresponding vector field $f : \mathbb{R}^3 \to \mathbb{R}^3$, is therefore

$$f \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} v \\ 0 \\ -\sin x \end{pmatrix}.$$

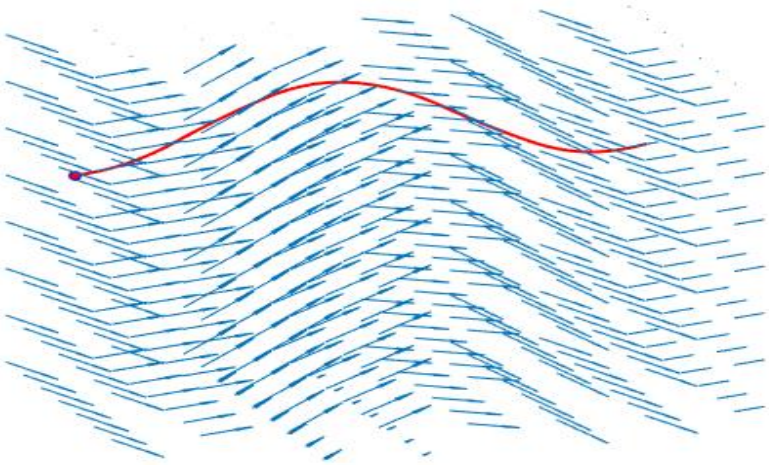


Figure 4.1: Vector field modeling the motion of particles in a fluid

Hence, for initial values $x_0, y_0, z_0 \in \mathbb{R}$ at initial time $t_0 = 0$, the corresponding solution $X : \mathbb{R} \to \mathbb{R}^3$ is the function defined by

$$X\,t = \begin{pmatrix} x_0 \\ y_0 \\ z_0 - \frac{\cos x_0}{v} \end{pmatrix} + \begin{pmatrix} vt \\ 0 \\ \frac{\cos(x_0+vt)}{v} \end{pmatrix}.$$

The fact that X solves the system of ODEs is easily certified with simple applications of well-known derivative rules:

$$X'\,t = \begin{pmatrix} v \\ 0 \\ -\sin(x_0+vt) \end{pmatrix} = f \begin{pmatrix} x_0+vt \\ y_0 \\ z_0 + \frac{\cos(x_0+vt)-\cos x_0}{v} \end{pmatrix} = f\,(X\,t).$$

On the other hand, checking that this function solves the IVP is just a simple substitution.

$$X\,0 = \begin{pmatrix} x_0 \\ y_0 \\ z_0 - \frac{\cos x_0}{v} \end{pmatrix} + \begin{pmatrix} v0 \\ 0 \\ \frac{\cos(x_0+v0)}{v} \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix}.$$

Therefore, the function X represents the trajectory for a given particle starting at the initial point (x_0, y_0, z_0). In Figure 4.1, the red line and dot are a possible depiction of X. □

In practice, solutions $X : T \to S$ to initial value problems $X'\,t = f\,(t, X\,t)$ with $X\,t_0 = s$ represent a possible evolution in time of the system described by the ODEs. Accordingly, the initial condition (t_0, s) describes the initial time and state of the system [63, 131]. However, solutions to IVPs need not be unique, which is an often needed property for the detailed analysis of the corresponding phenomenon.

For its relevance to our formalisation, below we describe a common method for obtaining conditions that guarantee local existence and uniqueness of solutions to IVPs. We start by

regarding IVPs as integral equations via the fundamental theorem of calculus:

$$X\,t - X\,t_0 = \int_{t_0}^{t} f\,(\tau, X\,\tau)d\tau.$$

Rearranging terms and substituting $X\,t_0 = s$, a function satisfies this equation if the operator h, defined by

$$h\,X\,t = s + \int_{t_0}^{t} f\,(\tau, X\,\tau)d\tau,$$

has a fixpoint. Therefore, finding conditions for local existence and uniqueness of solutions to IVPs is just a matter of applying fixpoint theory to h. In particular, h needs to be a contraction on a complete metric space, where completeness means existence of limits of Cauchy sequences and contraction refers to a distance decreasing map. Specifically, we need to choose $\varepsilon, \delta > 0$ so that h is a contraction on the metric space of bounded and continuous functions of type $\overline{B_\varepsilon(t)} \to \overline{B_\delta(s)}$, where $\overline{B_\delta(s)} = \{\tau \mid \|\tau - t\| \leq \delta\}$ is the closed ball of radius δ around s. If that is the case, then we can iterate h on this space, by defining $h^0\,X\,t = s$ and $h^{n+1} = h \circ h^n$, so that

$$X\,t = \lim_{n\to\infty} \left(s + \int_0^t f\,(h^{n-1}\,\tau)d\tau \right) = s + \int_0^t f\,(X\,\tau)d\tau,$$

where the last equality follows by continuity of addition, integration and f.

Moreover, for ensuring that h is a contraction, another requirement is that $f : T \times S \to \mathbb{R}^n$ must be *locally Lipschitz continuous* in S. That is, for each $t \in T$ and each $s \in S$ there must be $\delta > 0$ and $\ell \geq 0$ such that for all $s_1, s_2 \in \overline{B_\delta(s)} \cap S$,

$$\|f\,(t, s_1) - f\,(t, s_2)\| \leq \ell\,\|s_1 - s_2\|\,.$$

Here, $\|s\| = \sqrt{\sum_{i=1}^{n} s_i^2}$ is the euclidean norm of $\mathbb{R}^n$. Thus, we have sketched the proof of the following result.

Theorem 4.1.1 (Picard-Lindelöf)**.** *Let $f : T \times S \to \mathbb{R}^n$ be a vector field defined on a neighbourhood $T \times S$ of (t_0, s) such that f is locally Lipschitz continuous in S and continuous in T, then the IVP $X'\,t = f\,(t, X\,t)$ with $X\,t_0 = s$ has a unique solution for all $t \in T_s$ on some interval $T_s = \overline{B_\varepsilon(t)} \subseteq T$ with $\varepsilon > 0$.*

Flows or continuous dynamical systems $\varphi : T \to S \to S$ arise from the result above as follows. First, we set the time coordinate of the initial condition as $t_0 = 0$ to start imposing the monoid structure on T. Because of this, we can introduce the extra ODE $x_0'\,t = 1$ with initial condition $x_0\,0 = 0$ to our system. Therefore, we obtain a time-independent system of ODEs described by a vector field $f : S \to \mathbb{R}^n$. Picard-Lindelöf theorem for the autonomous case follows from the time-dependent case by using the function $\lambda(t, s).\ f\,s$. Hence, for each $s \in S$, this theorem provides the existence of an interval $T_s \subseteq T$ and a unique local solution or *trajectory* $\varphi_s^f : T_s \to S$ for the IVP, that is, $\varphi_s^{f\prime}\,t = f\,(\varphi_s^f\,t)$ for all $t \in T_s$ and $\varphi_s^f\,0 = s$. Geometrically, the functions φ_s^f are unique curves in S passing through their respective s and always tangential to f like the one depicted in Figure 4.1. Furthermore, their domains are all intervals around 0, which means that we can define a *local flow* function from $\bigcup_{s\in S} T_s \times \{s\}$

to S that maps each (t, s) to $\varphi_s^f\, t$. Alternatively, if, for all $s \in S$, $T_s = T$ is an uncountable monoid with neutral element 0, then $\varphi : T \to S \to S$ such that $\varphi\, t\, s = \varphi_s^f\, t$ is a (global) flow. That is, it satisfies the monoid action identities $\varphi\, 0\, s = s$ and $\varphi\, (t_1 + t_2)\, s = \varphi\, t_1\, (\varphi\, t_2\, s)$ for all $t_1, t_2 \in T$ [131].

Finally, flows provide state transformers by associating a given initial state s to the set of all points in its trajectory. Defining the *orbit* map $\gamma^\varphi : S \to \mathcal{P}\, S$, such that

$$\gamma^\varphi\, s = \mathcal{P}\, \varphi_s^f\, T_s = \{\varphi\, t\, s \mid t \in T_s\},$$

is the first step towards connecting ODEs with the state transformer model for Kleene algebras. As we will see in Section 4.4, this allows us to obtain rules for verification condition generation of hybrid programs.

Example 4.1.2 (Particles in a fluid revisited). A well-known result to check if a function is locally Lipschitz continuous consists in seeing if it is continuously differentiable. That is, it must be differentiable and its derivative, continuous. In the case of the vector field

$$f \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} v \\ 0 \\ -\sin x \end{pmatrix}$$

of Example 4.1.1, the three coordinates satisfy this property. Therefore, it is locally Lipschitz continuous. Furthermore, for each $s = (s_1, s_2, s_3) \in \mathbb{R}^3$, the interval of existence of the unique solution φ_s^f is the entire real line $\mathbb{R}$. Therefore, a global function $\varphi : \mathbb{R} \to \mathbb{R}^3 \to \mathbb{R}^3$ emerges

$$\varphi\, t\, s = \varphi_s^f\, t = \begin{pmatrix} s_1 \\ s_2 \\ s_3 - \frac{\cos s_1}{v} \end{pmatrix} + \begin{pmatrix} vt \\ 0 \\ \frac{\cos\,(s_1 + vt)}{v} \end{pmatrix}.$$

Checking that this function satisfies the additive monoid action law is entirely calculational:

$$\begin{aligned} \varphi\, t_1\, (\varphi\, t_2\, s) &= \begin{pmatrix} s_1 + vt_2 \\ s_2 \\ s_3 - \frac{\cos s_1}{v} + \frac{\cos\,(s_1 + vt_2)}{v} - \frac{\cos(s_1 + vt_2)}{v} \end{pmatrix} + \begin{pmatrix} vt_1 \\ 0 \\ \frac{\cos\,(s_1 + vt_2 + vt_1)}{v} \end{pmatrix} \\ &= \begin{pmatrix} s_1 \\ s_2 \\ s_3 - \frac{\cos s_1}{v} \end{pmatrix} + \begin{pmatrix} v(t_1 + t_2) \\ 0 \\ \frac{\cos\,(s_1 + v(t_1 + t_2))}{v} \end{pmatrix} \\ &= \varphi\, (t_1 + t_2)\, s. \end{aligned}$$

Given this property and the fact that $\varphi\, 0\, s = s$ which we know true from Example 4.1.1, we can conclude that φ is a flow. Thus, for each $s \in \mathbb{R}^3$, its orbit $\gamma^\varphi\, s = \{\varphi\, t\, s \mid t \in \mathbb{R}\}$ collects all the points in space where a particle starting at s will be. □

In summary, ordinary differential equations are the preferred model for describing physical phenomena. We can use vector fields to concisely represent systems of ODEs and, if the vector fields satisfy local Lipschitz continuity, then their associated initial value problems have unique local solutions by Picard-Lindelöf theorem. We can group these solutions or trajectories by patching their domains and defining a local flow. Yet, if the solutions are

global, the flow is a proper continuous dynamical system, that is, a monoid action. In the end, to obtain state transformers from ODEs, we only need the orbit of each initial state of the system. That is, we need the set of all its possible future states.

We can go beyond unique solutions in the description of our program semantics. This is the central concept of Section 5.1. However, in the meantime, we focus on the Isabelle/HOL formalisation of the concepts discussed in this section.

4.2 ODEs in Isabelle/HOL

In contrast with Section 3.5 where we use type classes to formalise Kleene algebras, here we use the related notion of locales for the formalisation of the results in Section 4.1. Their declaration differs only by the command used to call them (**class** and **locale** respectively). Two reasons require us to make this choice. Firstly, classes are mono-typed which does not fit with the two types in a flow's type $\varphi : T \to S \to S$, where $T \subseteq \mathbb{R}$ and S is a possibly different complete metric space. Secondly, a crucial reason for our use of locales is that we base our verification components on Immler and Hölzl's formalisation of Picard-Lindelöf theorem, which itself is done with locales on top of a large Isabelle/HOL library for analysis and ordinary differential equations [67, 73, 74, 76].

In [74], Hölzl and Immler prove existence and uniqueness results for time-dependent vector fields $f :: real \Rightarrow ('a::\{heine\text{-}borel, banach\}) \Rightarrow 'a$ in various scenarios. Specifically, in the context of their locale called *unique-on-bounded-closed*, they have a theorem called *unique-solution* that guarantees uniqueness of solutions within closed intervals of $\mathbb{R}$. Then, they specialise this to the entire real line in their locale *unique-on-strip*, to cartesian products of intervals times closed balls of fixed radii in *unique-on-cylinder*, and to open domains containing the initial condition in *ll-on-open-it*. We base our approach on the latter version as it is the one that resembles the most Theorem 4.1.1. Yet, for convenience we add the condition $t_0 \in T$ and remove intermediate definitions of their formalisation in the introduction of our own locale *picard-lindeloef* shown below.

locale *picard-lindeloef* =
  fixes f::*real* $\Rightarrow$ ($'a$::{*heine-borel*,*banach*}) $\Rightarrow$ $'a$
    and T::*real set*
    and S::$'a$ *set*
    and t_0::*real*
  assumes *open-domain*: *open* T *open* S
    and *interval-time*: *is-interval* T
    and *init-time*: $t_0 \in T$
    and *cont-vec-field*: $\forall\, s \in S.$ *continuous-on* T $(\lambda t.\ f\ t\ s)$
    and *lipschitz-vec-field*: *local-lipschitz* T S f
begin

sublocale *ll-on-open-it* T f S t_0
  by (*unfold-locales*) (*auto simp*: *cont-vec-field lipschitz-vec-field interval-time open-domain*)

lemma *unique-solution*: — proved for a subset of T for general applications
assumes $s \in S$ **and** $t_0 \in U$ **and** $t \in U$
and *is-interval* U **and** $U \subseteq$ *existence-ivl* t_0 s
and *xivp*: $D\ Y_1 = (\lambda t.\ f\ t\ (Y_1\ t))$ *on* U $\quad Y_1\ t_0 = s \quad Y_1 \in U \to S$
and *yivp*: $D\ Y_2 = (\lambda t.\ f\ t\ (Y_2\ t))$ *on* U $\quad Y_2\ t_0 = s \quad Y_2 \in U \to S$
shows $Y_1\ t = Y_2\ t$
⟨*proof*⟩

end

In the locale above, we list the assumptions of Picard-Lindelöf theorem. That is, the time-dependent vector field f is a total function, but continuous in time for each $s \in S$ and locally Lipschitz on the open sets T and S, where T is also an interval that satisfies $t_0 \in T$. In its context we show with the **sublocale** command that its assumptions imply those of *ll-on-open-it*. The latter provides the constant *existence-ivl* t_0 s which corresponds to the largest interval subset of T where solutions exist for initial (t_0, s). We use this constant to prove the *unique-solution* lemma for subsets U of T. The notation $D\ X = (\lambda t.\ f\ t\ (X\ t))$ *on* T is equivalent to $\forall t \in T.\ X'\ t = f\ t\ (X\ t)$ whereas $X \in T \to S$ indicates that the total function X maps elements of T specifically into S.

The uniqueness lemma above describes in its assumptions the formalisation of solutions to initial value problems. Due to their relevance, we define a corresponding set in Isabelle/HOL.

definition *ivp-sols* :: (*real* ⇒ *'a* ⇒ (*'a* :: *real-normed-vector*)) ⇒ (*'a* ⇒ *real set*) ⇒ *'a set* ⇒
real ⇒ *'a* ⇒ (*real* ⇒ *'a*) *set* (*Sols*)
where *Sols f U S* t_0 $s = \{X \in U\ s \to S\ .\ (D\ X = (\lambda t.\ f\ t\ (X\ t))\ on\ U\ s) \wedge X\ t_0 = s \wedge t_0 \in T\ s\}$

In this definition, U is a function to model the dependency of the interval of existence of solutions on the initial state s. Therefore, $X \in$ *Sols f U S* t_0 s if and only if X solves the initial value problem $\forall t \in U\ s.\ X'\ t = f\ t\ (X\ t)$ with $X\ t_0 = s$ and $X \in U\ s \to S$.

Locales such as *picard-lindeloef* bundle the assumptions of a theorem with its name as a predicate. Hence, asserting in Isabelle/HOL *picard-lindeloef f T S* t_0 is the same as asserting that f, T, S, and t_0 satisfy all the assumptions in that locale. If this predicate holds, then all the lemmas in the context of the locale such as *unique-solution* are derivable for f, T, S, and t_0. Based on this feature, we specialise *picard-lindeloef* to the time-independent case where also $t_0 = 0$ in a new locale called *local-flow*. Furthermore, we require a supplied function φ that for each $s \in S$ satisfies the associated IVP on every closed interval $\{0--t\}$, which is the set of all points between 0 and t where t might be greater or lower than 0.

locale *local-flow* = *picard-lindeloef* $(\lambda\ t.\ f)$ *T S 0*
for *f*::*'a*::{*heine-borel*,*banach*} ⇒ *'a*
and *T S L* +
fixes φ :: *real* ⇒ *'a* ⇒ *'a*
assumes *ivp*:
$\bigwedge\ t\ s.\ t \in T \Longrightarrow s \in S \Longrightarrow D\ (\lambda t.\ \varphi\ t\ s) = (\lambda t.\ f\ (\varphi\ t\ s))\ on\ \{0--t\}$
$\bigwedge\ s.\ s \in S \Longrightarrow \varphi\ 0\ s = s$
$\bigwedge\ t\ s.\ t \in T \Longrightarrow s \in S \Longrightarrow (\lambda t.\ \varphi\ t\ s) \in \{0--t\} \to S$

As a consequence of this definition, the set T of the locale is its own maximal interval of existence. This forces users of the locale to provide a correct domain for f from the specification. Other consequences are that $\lambda t.\ \varphi\, t\, s$ solves the systems of ODEs on all of T; it is therefore, for each $s \in S$, a solution of the associated IVP, and every other solution for such an IVP coincides with it on $t \in T$. That is, φ is a local flow for f.

lemma *ex-ivl-eq*: $s \in S \implies$ *existence-ivl* $t_0\ s = T$
⟨*proof*⟩

lemma *has-vderiv-on-domain*: $s \in S \implies D\ (\lambda t.\ \varphi\ t\ s) = (\lambda t.\ f\ (\varphi\ t\ s))$ *on* T
⟨*proof*⟩

lemma *in-ivp-sols*: $s \in S \implies 0 \in U\ s \implies U\ s \subseteq T \implies (\lambda t.\ \varphi\ t\ s) \in Sols\ (\lambda t.\ f)\ U\ S\ 0\ s$
⟨*proof*⟩

lemma *eq-solution*:
assumes $s \in S$
and *is-interval* $(U\ s)$ **and** $U\ s \subseteq T$ **and** $t \in U\ s$
and *xivp*: $X \in Sols\ (\lambda t.\ f)\ U\ S\ 0\ s$
shows $X\ t = \varphi\ t\ s$
⟨*proof*⟩

Finally, if $T = \mathbb{R}$, formalised below as $T = UNIV$ where $UNIV$ is the universal set of a type, then the flow φ is global and therefore a monoid action.

lemma *ivp-sols-collapse*: $T = UNIV \implies s \in S \implies Sols\ (\lambda t.\ f)\ (\lambda s.\ T)\ S\ 0\ s = \{(\lambda t.\ \varphi\ t\ s)\}$
⟨*proof*⟩

lemma *is-monoid-action*:
assumes $s \in S$ **and** $T = UNIV$
shows $\varphi\ 0\ s = s$
and $\varphi\ (t_1 + t_2)\ s = \varphi\ t_1\ (\varphi\ t_2\ s)$
⟨*proof*⟩

Finally, for certifying derivatives, we have created a tactic in Isabelle/HOL that consists in collecting derivative rules and feeding them to the *auto* tactic. That is, we have created a list of theorems named *poly-derivatives*.

named-theorems *poly-derivatives compilation of optimised miscellaneous derivative rules.*

Then, we add derivative rules to this list, for instance, rules for sines, cosines and exponentials.

lemma *vderiv-cosI*[*poly-derivatives*]:
assumes $D\ (f{::}real \Rightarrow real) = f'$ *on* T **and** $g = (\lambda t.\ -\ (f'\ t) * sin\ (f\ t))$
shows $D\ (\lambda t.\ cos\ (f\ t)) = g$ *on* T
⟨*proof*⟩

lemma *vderiv-sinI*[*poly-derivatives*]:
assumes $D\ (f{::}real \Rightarrow real) = f'\ on\ T$ **and** $g = (\lambda t.\ (f'\ t) * cos\ (f\ t))$
shows $D\ (\lambda t.\ sin\ (f\ t)) = g\ on\ T$
⟨*proof*⟩

lemma *vderiv-expI*[*poly-derivatives*]:
assumes $D\ (f{::}real \Rightarrow real) = f'\ on\ T$ **and** $g = (\lambda t.\ (f'\ t) * exp\ (f\ t))$
shows $D\ (\lambda t.\ exp\ (f\ t)) = g\ on\ T$
⟨*proof*⟩

Other rules that we have added include rules for addition, multiplication, division and exponentiation to a natural power. This allows *auto* to apply the rule that fits best automatically. The following is a derivative certification involving exponentials, cosines and divisions.

lemma $c \neq 0 \Longrightarrow D\ (\lambda t.\ a5 * t\hat{}5 + a3 * (t\hat{}3\ /\ c) - a2 * exp\ (t\hat{}2) + a1 * cos\ t + a0) =$
$(\lambda t.\ 5 * a5 * t\hat{}4 + 3 * a3 * (t\hat{}2\ /\ c) - 2 * a2 * t * exp\ (t\hat{}2) - a1 * sin\ t)\ on\ T$
by(*auto intro*!: *poly-derivatives simp*: *power2-eq-square*)

As we will see in Sections 5.4 and 5.5, this improvement in automation contributes to faster completion of two verification procedures described in Section 4.4 and Section 5.2 respectively.

In summary, the *picard-lindeloef* locale provides conditions for existence and uniqueness of solutions for time-independent IVPs in Isabelle/HOL. On the other hand, *local-flow* is the autonomous special case of *picard-lindeloef* with 0 being the initial time and extended with a variable φ for specifying a characterisation of the flow. They form the basic infrastructure of dynamical systems for building verification components in the proof assistant. We also added the tactic *poly-derivatives* to certify derivatives automatically.

4.3 Semantics for Assignments

Recall that we wish to provide semantics of hybrid programs defined by the grammar

$$X \ ::= \ x := e \mid x' = f\,\&\,G \mid ?P \mid X\,;X \mid X + X \mid X^*,$$

and derive verification conditions for each constructor. In the case of the last four, we explained in Chapter 3 how to obtain these from state transformers $f : S \to \mathcal{P}\,S$ or relations $R \subseteq S \times S$ via Kleene algebras. Here we focus on defining semantics for assignments $x := e$.

We start by fixing a finite set of program variables V. Then, states $s \in S$ of hybrid programs correspond to functions from V to $\mathbb{R}$, that is $S \subseteq \mathbb{R}^V$. Intuitively, for a state $s : V \to \mathbb{R}$ and a variable $x \in V$, the real number $s\,x$ is the value of x at state s. Section 6.4 discusses an alternative approach that abstracts from this concrete implementation to more complex state spaces.

Given an input state $s : V \to \mathbb{R}$, the output state of an assignment $x := e$ should coincide with s everywhere but in x, where expression e dictates the value of $s\,x$. Furthermore, the right hand side in an assignment $x := e$ may depend on the value of the program

variables at the previous state. This means that making e a fixed value would not align with our intuition. Instead, we make it a function $e : S \to \mathbb{R}$. Also, in our shallow embedding approach, assignments should correspond to either a state transformer or to a relation. Hence, we define them in the state transformer model as functions that map states to singletons of updated states, that is

$$(x :=_{\mathcal{F}} e)\, s = \{s[x \mapsto e\, s]\},$$

where we use update functions $-[- \mapsto -] : B^A \to A \to B \to B^A$ such that $f[a \mapsto b]\, a = b$ and $f[a \mapsto b]\, x = f\, x$ for any function $f : A \to B$ and for $x \neq a$. Using our notation from the powerset monad, our definition of assignments is just lifting update functions to state transformers $S \to \mathcal{P}\, S$ via $\eta_S \circ -[x \mapsto e\, s]$.

Similarly to other hybrid programs, the isomorphism $\mathcal{R}$ between relations and state transformers allows us to compute the relational version of the semantics for assignments

$$(x :=_{\mathcal{R}} e) = \mathcal{R}\,(x :=_{\mathcal{F}} e) = \{(s, s[x \mapsto e\, s]) \mid s \in S\}.$$

We can also start with the relational version and define $(x :=_{\mathcal{F}} e) = \mathcal{F}\,(x :=_{\mathcal{R}} e)$. Nevertheless, in both semantics, the *wlp* or forward box predicate transformer, computes the same value for assignments

$$|x := e]\, Q\, s = Q\, s[x \mapsto e\, s], \tag{wlp-assign}$$

which justifies dropping the subscripts $\mathcal{F}$ and $\mathcal{R}$. In the equation above, we also adhere to our indistinction of predicates $Q : S \to \mathbb{B}$ with their respective boolean algebras of each semantics. Similarly, the Hoare triple for assignments is also derivable

$$\{\lambda s.\ Q\, s[x \mapsto e\, s]\}\, x := e\, \{Q\}. \tag{h-assign}$$

The following couple of examples show how to use these two rules of verification condition generation for assignments.

Example 4.3.1 (`thermostat`'s control revisited). In Example 3.2.1, we derived the leafs of a proof tree to verify the correctness specification

$$\{\lambda s.\ T_m \leq s\, T \leq T_M\}\, \texttt{ctrl}\, \{\lambda s.\ T_m \leq s\, T \leq T_M\},$$

where the hybrid program

$$\begin{aligned} \texttt{ctrl} = {} & (t := \lambda s.\ 0)\,;(T_0 := \lambda s.\ s\, T);\\ & \textbf{if } \lambda s.\ s\, \theta = 0 \wedge s\, T_0 \leq T_m + 1 \textbf{ then } \theta := \lambda s.\ 1 \textbf{ else}\\ & \textbf{if } \lambda s.\ s\, \theta = 1 \wedge s\, T_0 \geq T_M - 1 \textbf{ then } \theta := \lambda s.\ 0 \textbf{ else skip} \end{aligned}$$

models a simple control for a thermostat turning a heater θ on or off to regulate the temperature T of a room. The specification then requires that, after the discrete intervention of the thermostat, the room's temperature remains between T_m and T_M. Notice that, in contrast with our previous depiction of `ctrl`, we now make explicit the types $S \to \mathbb{R}$ and $S \to \mathbb{B}$ in assignments and tests respectively.

In Example 3.2.1, we conclude that each leaf of the tree requires a proof of a Hoare triple involving an assignment. Two of them are

$$\{\lambda s.\ T_m \le s\,T \le T_M\}\,(t := \lambda s.\ 0)\,\{\lambda s.\ T_m \le s\,T \le T_M\},$$
$$\{\lambda s.\ T_m \le s\,T \le T_M\}\,(T_0 := \lambda s.\ s\,T)\,\{\lambda s.\ T_m \le s\,T \le T_M\}.$$

With our recently defined semantics for assignments, we can validate the conclusion of Example 3.2.1. Indeed, we just need to observe that

$$(\lambda s.\ (\lambda \varsigma.\ T_m \le \varsigma\,T \le T_M)\,s[t \mapsto 0]) = (\lambda s.\ T_m \le s\,T \le T_M) \text{ and}$$
$$(\lambda s.\ (\lambda \varsigma.\ T_m \le \varsigma\,T \le T_M)\,s[T_0 \mapsto s\,T]) = (\lambda s.\ T_m \le s\,T \le T_M),$$

because the update does not modify $s\,T$. Therefore, we can rewrite the above Hoare triples in the (h-assign) form and discharge them because they are simply true.

The rest of the leafs of the proof tree involve assignments for θ, either $(\theta := \lambda s.\ 1)$ or $(\theta := \lambda s.\ 0)$ with the same postcondition $(\lambda s.\ T_m \le s\,T \le T_M)$. The only difference is that the precondition includes an extra conjunct. For instance,

$$\{\lambda s.\ s\,\theta = 0 \wedge T_m \le s\,T \le T_M\}\,(t := \lambda s.\ 1)\,\{\lambda s.\ T_m \le s\,T \le T_M\}.$$

In those cases, we only need to weaken the precondition with an application of (h-cons) to obtain the proof obligation

$$\{\lambda s.\ T_m \le s\,T \le T_M\}\,(\theta := \lambda s.\ 1)\,\{\lambda s.\ T_m \le s\,T \le T_M\},$$

and proceed as in the previous branches.

Therefore, due to the fact that we can derive a true statement for each of the leafs in the proof tree, the results presented here and in Example 3.2.1 verify that $\lambda s.\ T_m \le s\,T \le T_M$ is an invariant for `ctrl`. □

Example 4.3.2 (*wlp* of `thermostat`'s control). Similarly, in Example 3.3.1 we partially obtained a weakest liberal precondition of `ctrl` which we copy here.

$$|\texttt{ctrl}]\,P = |t := 0]\,|T_0 := T]\,(t_1\,;\,|\theta := 1]\,P + \neg t_1\,;\,t_2\,;\,|\theta := 0]\,P + \neg t_1\,;\,\neg t_2\,;\,P)\,,$$

where $t_1 = (\theta = 0 \wedge T_0 \le T_m + 1)$ and $t_2 = (\theta = 1 \wedge T_0 \ge T_M - 1)$.

Making explicit the semantic types of assignments and tests, the *wlp* becomes

$$\begin{aligned}
|\texttt{ctrl}]\,P\,s = |t := \lambda \varsigma.\ 0]\,|T_0 := \lambda \varsigma.\ \varsigma\,T](Q_1 \vee Q_2 \vee Q_3)\,s, \text{ where}\\
Q_1\,s = (t_1\,s \wedge |\theta := \lambda \varsigma.\ 1]\,P\,s),\\
Q_2\,s = (\neg t_1\,s \wedge t_2\,s \wedge |\theta := \lambda \varsigma.\ 0]\,P\,s),\\
Q_3\,s = (\neg t_1\,s \wedge \neg t_2\,s \wedge P\,s).
\end{aligned}$$

Thus, it remains for us to apply (wlp-assign) to complete the calculation. The right hand side then becomes $(Q_1 \vee Q_2 \vee Q_3)\,s[t \mapsto 0][T_0 \mapsto s\,T]$. In particular,

$$\begin{aligned}
Q_1\,s[t \mapsto 0][T_0 \mapsto s\,T] &= s\,\theta = 0 \wedge s\,T \le T_m + 1 \wedge P\,s[t \mapsto 0][T_0 \mapsto s\,T][\theta \mapsto 1],\\
Q_2\,s[t \mapsto 0][T_0 \mapsto s\,T] &= \neg\,(s\,\theta = 0 \wedge s\,T \le T_m + 1) \wedge (s\,\theta = 1 \wedge s\,T \ge T_M - 1)\\
&\qquad \wedge P\,s[t \mapsto 0][T_0 \mapsto s\,T][\theta \mapsto 0], \text{ and}\\
Q_3\,s[t \mapsto 0][T_0 \mapsto s\,T] &= \neg\,(s\,\theta = 0 \wedge s\,T \le T_m + 1) \wedge \neg\,(s\,\theta = 1 \wedge s\,T \ge T_M - 1)\\
&\qquad \wedge P\,s[t \mapsto 0][T_0 \mapsto s\,T].
\end{aligned}$$

Intuitively, each Q_i with $i \in \{1,2,3\}$ represents a region of the thermostat's behaviour. Q_1 corresponds to the moment when the temperature gets close to the lower bound T_m. Similarly, Q_2 represents a region where the temperature is close to T_M. Finally, Q_3 considers all the remaining regions where T might not be close to any of the boundaries of the comfortable zone. Therefore, the *wlp* of `ctrl`, states that P holds after its execution if and only if any of the regions is true with $P\,s[t \mapsto 0][T_0 \mapsto s\,T][\theta \mapsto 1]$ holding at Q_1, accordingly $P\,s[t \mapsto 0][T_0 \mapsto s\,T][\theta \mapsto 0]$ doing so at Q_2, and $P\,s[t \mapsto 0][T_0 \mapsto s\,T]$ at Q_3. In particular, observe that if $P = (\lambda s.\ s\,t = 0)$, then

$$P\,s[t \mapsto 0][T_0 \mapsto s\,T][\theta \mapsto 1] = P\,s[t \mapsto 0][T_0 \mapsto s\,T][\theta \mapsto 0] = P\,s[t \mapsto 0][T_0 \mapsto s\,T] = \top.$$

Therefore, $\lambda s.\ s\,t = 0$ holds after the execution of `ctrl`. □

Adding (h-assign) to the rules obtained from KATs completes the derivation of Hoare logic for while programs as in [9]. Accordingly, adding (wlp-assign) to the forward box equations of MKAs like in [55] generates a calculus for computing weakest liberal preconditions of while programs. Therefore, to obtain a hybrid version of these calculi, it remains to provide semantics and rules for verification condition generation of evolution commands $x' = f\,\&\,G$. In the sequel we pursue this goal.

4.4 Semantics for Evolution Commands

In this section, we explain one of the most important conceptual contributions from our work in [44, 72]. Specifically, we proceed to define the state transformer and relational semantics of evolution commands $x' = f\,\&\,G$. These are hybrid programs that consist of a system of differential equations, supplied by the vector field f, and a predicate G which models boundary conditions. The predicate G is called an *evolution domain constraint* or simply a *guard*. The remaining variable x' and the equality symbol $=$ are just syntactic addenda to resemble ODEs. Intuitively, "executing" an evolution command means that the system follows the flow of the differential equation while respecting the guard for an unspecified amount of time. In what follows, we make this intuition mathematically concrete.

At the end of Section 4.1, we observed that the orbit $\gamma^\varphi\,s = \{\varphi_s^f\,t \mid t \in T_s\}$ for a trajectory $\varphi_s^f : T_s \to S$, with $T_s \subseteq \mathbb{R}$ and $s \in S \subseteq \mathbb{R}^V$, is a state transformer $\gamma^\varphi : S \to \mathcal{P}\,S$. Therefore, it is a good initial candidate for a semantics of evolution commands. Due to the isomorphism between the finite dimensional vector space $\mathbb{R}^n$ and $\mathbb{R}^V$, where V is the set of n possible program variables, this observation still holds in our state space $S \subseteq \mathbb{R}^V$. However, we can generalise orbits in two ways. Firstly, we do not need trajectories to describe evolutions in time. In fact, we only need functions $X : T \to S$ mapping times to states. Secondly, because of the evolution domain restriction in evolution commands, we prefer to work with a G-guarded version of the orbit. Hence, we define the *G-guarded orbit map* as the function $\gamma : (T \to S) \to (S \to \mathbb{B}) \to \mathcal{P}\,T \to \mathcal{P}\,S$ such that

$$\gamma\,X\,G\,U = \bigcup\{\mathcal{P}\,X\,(\downarrow U\,t) \mid \mathcal{P}\,X\,(\downarrow U\,t) \subseteq G\},$$

where we abuse notation and treat predicates as sets. Recall that $\downarrow U\,t = \{\tau \in U \mid \tau \leq t\}$, and observe that, by its type, $U \subseteq T$. Therefore, the set $\gamma\,X\,G\,U$ patches the image under

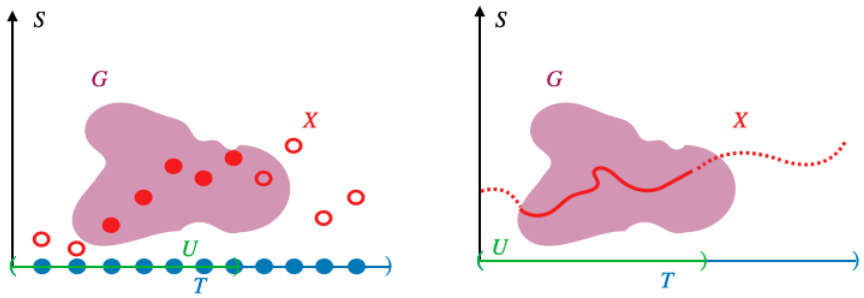


Figure 4.2: A discrete (left) and a continuous (right) guarded orbit

X of all the points of U below t that respect G for each $t \in T$. Figure 4.2 schematises two guarded orbits, one with a discrete time domain and another with continuous time. It shows points in the discrete guarded orbit as filled dots and those in the continuous one as part of an uninterrupted line. The guarded orbit are all those points in the image of X that satisfy G within U. Notice that U is mainly there to restrict our domain of interest for the guarded orbit even though X is defined in all of T. We can see the resemblance of this definition with that of traditional orbits in the following result.

Lemma 4.4.1. *Let S be a set, $U \subseteq T \subseteq \mathbb{R}$, $X : T \to S$ and $G : S \to \mathbb{B}$, then*

$$\gamma\, X\, G\, U = \{X\, t \mid t \in U \wedge \forall \tau \in {\downarrow} U\, t.\ G\, (X\, \tau)\}.$$

Geometrically, if U is an interval, this means that G-guarded orbits collect all the points of the longest initial segment of the curve X that also satisfies G. Moreover, as indicated in the lemma below, $\top$-guarded orbits coincide with traditional orbits γ^{φ}, where $\top$ is the constant true predicate.

Lemma 4.4.2. *Suppose φ_s^f is the unique solution on T_s to the IVP given by the locally Lipschitz continuous vector field $f : S \to S$ and the initial condition $(0, s)$, then*

$$\gamma\, \varphi_s^f\, \top\, T_s = \gamma^{\varphi}\, s.$$

Given these results, we are ready to define the state transformer semantics for evolution commands. For a state $s \in S \subseteq \mathbb{R}^V$, an autonomous locally Lipschitz continuous vector field $f : S \to S$, and a predicate $G : S \to \mathbb{B}$, let $\varphi_s^f : T_s \to S$ be the unique longest trajectory for the respective IVP such that $\varphi_s^f\, 0 = s$ with $T_s \subseteq \mathbb{R}$. If U is an interval satisfying $0 \in U \subseteq T_s$, then the state transformer semantics of evolution commands is the G-guarded orbit

$$(x' =_{\mathcal{F}} f \,\&\, G)_U\, s = \gamma\, \varphi_s^f\, G\, U = \{\varphi_s^f\, t \mid t \in U \wedge \forall \tau \in {\downarrow} U\, t.\ G\, (\varphi_s^f\, \tau)\}.$$

On the other hand, applying the isomorphism $\mathcal{R}$ between relations and state transformers yields the relational semantics

$$(x' =_{\mathcal{R}} f \,\&\, G)_U = \mathcal{R}\, (x' =_{\mathcal{F}} f \,\&\, G)_U = \{(s, \varphi_s^f\, t) \mid t \in U \wedge \forall \tau \in {\downarrow} U\, t.\ G\, (\varphi_s^f\, \tau)\}.$$

Just like with assignments, we could start with this equation and obtain our original definition via $(x' =_{\mathcal{F}} f \,\&\, G)_U = \mathcal{F}\, (x' =_{\mathcal{R}} f \,\&\, G)_U$. Furthermore, recall that the interval T_s guaranteed

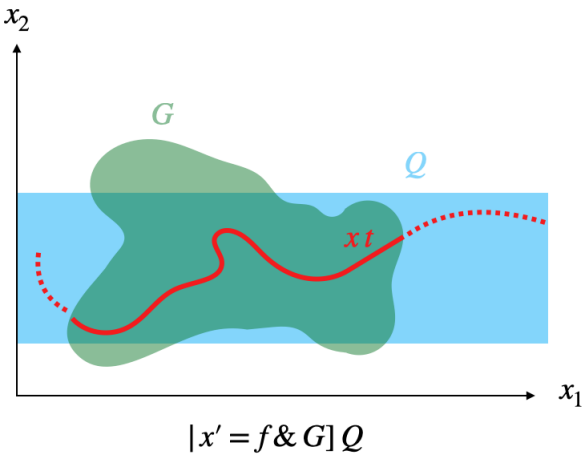


Figure 4.3: The continuous line illustrates where the formula $|(x' = f \,\&\, G)_U]\, Q$ holds.

by Picard-Lindelöf theorem is a neighbourhood of 0, thus, some part of the unique solution exists on the positive reals. Based on this, if the interval $U \subseteq T_s$ is the longest initial segment of the nonnegative reals $\mathbb{R}_+$ in T_s, we get the standard relational semantics of $\mathsf{d}\mathcal{L}$ [113].

It only remains to derive rules for verification condition generation of evolution commands. Recall that in the MKA semantics $P\,;\alpha \leq \alpha\,;Q$ of *wlp*s and Hoare triples, the sequential composition on the left-hand side guarantees P before α. Similarly, Q holds after α by the right hand side. Contrastingly, in the case of evolution commands and consistently with $\mathsf{d}\mathcal{L}$, all the points of the orbit satisfy Q (see also Figure 4.3):

$$|(x' = f \,\&\, G)_U]\, Q\, s = \forall t \in U.\ (\forall \tau \in {\downarrow}U\, t.\ G\,(\varphi_s^f\, \tau)) \to Q\,(\varphi_s^f\, t). \tag{wlp-evol}$$

As expected, in both semantics the weakest liberal preconditions coincide, which enables us to omit subscripts $\mathcal{F}$ and $\mathcal{R}$. Similarly, a Hoare triple for evolution commands is also available

$$\{\lambda\, s.\ \forall t \in U.\ (\forall \tau \in {\downarrow}U\, t.\ G\,(\varphi_s^f\, \tau)) \to Q\,(\varphi_s^f\, t)\}\ (x'{=}f \,\&\, G)_U\ \{Q\}. \tag{h-evol}$$

The assumptions on our definition of the semantics for evolution commands are essential. The results derived so far hold only for autonomous Lipschitz continuous vector fields with initial time 0. As a consequence, verifying $x' = f \,\&\, G$ in this setting requires the following procedure

1. Show that $f : S \to S$ satisfies the assumptions of Picard-Lindelöf theorem, that is, f is locally Lipschitz continuous and $S \subseteq \mathbb{R}^V$ is open;
2. Supply a (local) flow φ for f with a subset $0 \in U$ of the interval of existence T;
3. Certify that it is indeed a (local) flow, that is, show that for all $s \in S$ and all $t \in T$, $\varphi_s^{f'}\, t = f\,(\varphi_s^f\, t)$ and $\varphi_s^f\, 0 = s$.
4. If all previous steps succeed, apply either (h-evol) or (wlp-evol) depending on the required specification.

As with other deductive verification approaches, the procedure above is as successful as the user is knowledgable and skillful with mathematical results. This is because finding a Lipschitz constant or a solution to a differential equation ranges from easy to impossible. In many cases, working with analytical solutions is not possible which is why we discuss in Section 5.2 alternative ways to reason about ODEs. In $\mathtt{d}\mathcal{L}$, the first step is not required because all terms in its language generate Lipschitz continuous vector fields. Nevertheless, in many applications, we can remove this requirement or use an alternative one (see Sections 5.3 and 6.5). Similarly, in practice, users can delegate the computation of solutions to systems of differential equations in step 2 to computer algebra systems. We explore further simplifications to the procedure above in Sections 5.2 and 5.3. In the meantime, we exemplify its use below.

Example 4.4.1 (`thermostat`'s verification). Recall from Figure 2.5, and the running Examples 3.2.1, 3.3.1, 4.3.1 and 4.3.2 that $\mathtt{thermostat} = \mathbf{loop}\ (\mathtt{ctrl}\,;\mathtt{dyn})$ where

$$\mathtt{dyn} = \mathbf{if}\ \lambda s.\ s\,\theta = 0\ \mathbf{then}\ x' = f_0 \,\&\, G_0\ \mathbf{else}\ x' = f_{T_L} \,\&\, G_{T_L},$$

and `ctrl` is as in Example 4.3.1. The vector field f_c models the heating and cooling dynamics with its defining equations $f_c\,s\,T = -a(s\,T - c)$, $f_c\,s\,t = 1$, $f_c\,s\,\theta = 0$, and $f_c\,s\,T_0 = 0$. Essentially, this means that f_c makes the room temperature decrease or grow exponentially towards $c \in \{0, T_L\}$ at a rate of $a > 0$, where $0 \leq T_m$ and $T_M \leq T_L$. As the derivatives equate to 0, it leaves the program variables T_0 and θ unchanged. Finally, it also includes a coordinate to model the passing of time. The guards in the dynamics are

$$G_c\,s \leftrightarrow \left(s\,t \leq -\frac{1}{a}\ln\left(\frac{c - L_c}{c - s\,T_0}\right)\right),$$

where L_c is T_m if $c = 0$ or T_M if $c = T_L$. They restrict the duration of the evolution command to never go beyond T_m or T_M.

The material presented so far suffices to verify the entire hybrid program. However, for simplicity, we will limit ourselves to obtain the *wlp* for the first branch of the continuous dynamics, that is $x' = f_0 \,\&\, G_0$. For a full verification of `thermostat`, we refer the reader to the formalisation in Section 5.4.

According to the above procedure, to obtain $|(x'{=}f_0 \,\&\, G_0)_U]\,P$, we first need to check that f_0 is locally Lipschitz continuous. Simply put, we must find a constant ℓ such that $\|f_0\,s_1 - f_0\,s_2\| \leq \ell\,\|s_1 - s_2\|$ in a suitable region. Given that for variables t, T_0 and θ the vector field is constant, the difference $f_0\,s_1 - f_0\,s_2$ becomes 0 at those coordinates. Thus,

$$\|f_0\,s_1 - f_0\,s_2\| = |f_0\,s_1\,T - f_0\,s_2\,T| = a\,|s_1\,T - s_2\,T| \leq a\,\|s_1 - s_2\|.$$

Hence, f_0 is Lipschitz continuous in all of $\mathbb{R}^V$, and the equations $\varphi_s^{f_0}\,\tau\,T = s\,T\exp(-a\tau)$, $\varphi_s^{f_0}\,\tau\,t = s\,t + \tau$, $\varphi_s^{f_0}\,\tau\,T_0 = s\,T_0$, and $\varphi_s^{f_0}\,\tau\,\theta = s\,\theta$ describe the unique solution to the associated initial value problem for $s = (s_1, s_2, s_3)^\top \in \mathbb{R}^V$.

To simplify our result, we fix $U\,s = \mathbb{R}_+$. Based on this, applying (wlp-evol) gives us

$$\begin{aligned}|(x' = f_0 \,\&\, G_0)_{\mathbb{R}_+}]\,Q\,s &= \forall r \geq 0.\ (\forall \tau \in [0, r].\ G\,(\varphi_s^{f_0}\tau)) \to Q\,(\varphi_s^{f_0}\,r)\\ &= \forall r \geq 0.\ \left(\forall \tau \in [0, r].\ \varphi_s^{f_0}\,\tau\,t \leq -\frac{1}{a}\ln\left(\frac{T_m}{s\,T_0}\right)\right) \to Q\,(\varphi_s^{f_0}\,r).\end{aligned}$$

In particular, we know from Example 4.3.1 that $\lambda s.\ s\,t = 0$ holds after the execution of `ctrl`. Therefore, $\varphi_s^{f_0}\,\tau\,t = \tau$ for all $\tau \in [0, r]$ during the execution of this branch of `dyn`. Thus, in that case, the inequality in the *wlp* above implies $0 \geq -a\tau \geq \ln(T_m/s\,T_0)$. If we can guarantee that $\varphi_s^{f_0}\,0\,T = s\,T = s\,T_0$, we could prove that $\varphi_s^{f_0}\,\tau\,T = s\,T\exp(-a\tau) \geq T_m$. This is the reason why `ctrl` includes the assignment $(T_0 := \lambda s.\ s\,T)$. See lemma *thermostat* in Section 5.4 for a full verification of `thermostat`. □

Despite the requirements listed in the procedure above, our G-guarded orbits apply to more functions than just those satisfying Picard-Lindelöf theorem. This suggests the possibility of alternative generalised semantics of evolution commands which we explore in more detail in Sections 5.1 and 5.3.

Adding assignments and evolution commands to the regular programs from Kleene algebras generates hybrid programs. Thus, our work in this section completes our derivation of rules for each of them. Namely, by adding (h-evol) and (h-assign) to the rules of Hoare logic of Section 3.2, we get *differential Hoare logic* $\mathsf{d}\mathcal{H}$. This is a minimalistic logic for verification of hybrid systems. Analogously, (wlp-evol) and (wlp-assign) complete our work in this chapter for extending the *wlp*-calculus of Section 3.3 to reason about hybrid programs. As usual, we provide one verification rule for each hybrid program. Then, applying these rules recursively generates proof obligations which are specific to the concrete relational or state transformer semantics on $S \subseteq \mathbb{R}^V$.

4.5 Hybrid Stores in Isabelle/HOL

In this section, we describe the instantiation of the state transformer and relational semantics to our concrete hybrid store in Isabelle/HOL. We explain the connection between our previous Sections 3.5 and 4.2 that formalise Kleene algebras and ODEs respectively. Then we build upon them to define assignments as in Section 4.3 and obtain rules for verification condition generation for this hybrid program.

As explained in Section 4.3, our hybrid stores $s \in \mathbb{R}^V$ correspond to functions from a finite set of program variables to the real numbers. In Isabelle, we model the set V with a finite type $'n :: \mathit{finite}$. The state space $\mathbb{R}^V$ then corresponds to the type of real valued vectors $(\mathit{real}, 'n)\ \mathit{vec}$ which we abbreviate with $\mathit{real}\hat{}'n$. By definition, this is a subtype of the type of functions from $'n$ to *real*. This means that, by default, Isabelle/HOL provides an isomorphism between $\mathit{real}\hat{}'n$ and $'n \Rightarrow \mathit{real}$. This isomorphism uses infix notation \$. Operationally, $s\$i$ is the ith coordinate of s, or rather, the value of store s at variable i. Its inverse uses a χ-binder replacing λ-abstraction. Hence, $(\chi i.\ s\$i) = s$ for all $s :: \mathit{real}\hat{}'n$ and $(\chi\ i.\ x)\$i = x$ for $x :: \mathit{real}$

In Isabelle there is an instantiation proof showing that vectors with finite dimension over a field such as $\mathit{real}\hat{}'n$ form a Banach space where the Heine-Borel theorem applies. That is, $\mathit{real}\hat{}'n$ is a complete metric space in the sense of Section 4.1 whose compact sets are precisely its closed and bounded sets. This enables us to use the assumptions of the locale *picard-lindeloef* on vector fields $f :: \mathit{real} \Rightarrow \mathit{real}\hat{}'n \Rightarrow \mathit{real}\hat{}'n$.

Similarly, recall that we have two models for Kleene algebras in Isabelle/HOL because of our interpretation of relations as antidomain Kleene algebras and a similar instantiation proof for state transformers of Section 3.5. This means that we can apply our rules for forward box

operators on relations of type $(real\hat{}'n \times real\hat{}'n)$ *set* and nondeterministic functions $(real\hat{}'n)$ *nd-fun*. Thus, we only show the formalisation with the relational model below because the process for state transformers is analogous. However, as shown below, we change notation at the level of these concrete semantics to resemble that of regular programs. We also simplify the verification laws by removing antidomain and antirange operations as follows.

notation *Id* (*skip*)
  and *relcomp* (**infixl** ; *70*)
  and *zero-class.zero* (*0*)
  and *rel-aka.fbox* (*wp*)

With command **notation**, we can use *skip* instead of the identity relation, semicolon ; instead of relational composition, and *wp* instead of the forward box operator. Afterwards, below we introduce the isomorphism between predicates and subidentity relations and add simplification rules about this operator via the lemma *p2r-simps*. The last two lines of this lemma, in particular, indicate that the antidomain operation on a lifted predicate corresponds to the lifting of its negation, and that the domain operation leaves them unchanged.

definition *p2r* :: *'a pred* ⇒ *'a rel* ((*1* ⌈-⌉)) **where**
  $\lceil P \rceil = \{(s,s) \mid s.\ P\ s\}$

lemma *p2r-simps*[*simp*]:
  $\lceil P \rceil \leq \lceil Q \rceil = (\forall\, s.\ P\ s \longrightarrow Q\ s)$
  $(\lceil P \rceil = \lceil Q \rceil) = (\forall\, s.\ P\ s = Q\ s)$
  $(\lceil P \rceil\ ;\ \lceil Q \rceil) = \lceil \lambda\ s.\ P\ s \wedge Q\ s \rceil$
  $(\lceil P \rceil \cup \lceil Q \rceil) = \lceil \lambda\ s.\ P\ s \vee Q\ s \rceil$
  rel-ad $\lceil P \rceil = \lceil \lambda s.\ \neg\ P\ s \rceil$
  rel-aka.ads-d $\lceil P \rceil = \lceil P \rceil$
  unfolding *p2r-def rel-ad-def rel-aka.ads-d-def* **by** *auto*

Then, we show the relational equivalent to the forward box operation with lemma *wp-rel*. Finally, we introduce upper case abbreviations for the hybrid programs of Sections 3.2 and 3.5. The corresponding rules for verification condition generation, just call their Kleene algebra versions of Section 3.5 and the *auto* tactic.

lemma *wp-rel*: *wp R* $\lceil P \rceil = \lceil \lambda\ x.\ \forall\ y.\ (x,y) \in R \longrightarrow P\ y \rceil$
  unfolding *rel-aka.fbox-def p2r-def rel-ad-def* **by** *auto*

abbreviation *cond-sugar* :: *'a pred* ⇒ *'a rel* ⇒ *'a rel* ⇒ *'a rel* (*IF - THEN - ELSE - [64,64] 63*)
  where *IF P THEN X ELSE Y* ≡ *rel-aka.aka-cond* $\lceil P \rceil$ *X Y*

abbreviation *loopi-sugar* :: *'a rel* ⇒ *'a pred* ⇒ *'a rel* (*LOOP - INV - [64,64] 63*)
  where *LOOP R INV I* ≡ *rel-aka.aka-loopi R* $\lceil I \rceil$

lemma *wp-loopI*:
  $\lceil P \rceil \leq \lceil I \rceil \Longrightarrow \lceil I \rceil \leq \lceil Q \rceil \Longrightarrow \lceil I \rceil \leq wp\ R\ \lceil I \rceil \Longrightarrow \lceil P \rceil \leq wp\ (LOOP\ R\ INV\ I)\ \lceil Q \rceil$
  using *rel-aka.fbox-loopi*[*of* $\lceil P \rceil$] **by** *auto*

At this point, we only need to add assignments and evolution commands to our Isabelle/HOL verification components. We leave the formalisation of the latter for Section 5.4 because for their Isabelle implementation we use a generalisation described in Section 5.1. Therefore, we focus on assignments here. The first step is to define updates for the type *real^'n*. We do this via the inverse χ of the isomorphism \$ between vectors and functions with finite domain, and with Isabelle's function updates denoted $f(a := b)$ instead of $f[a \mapsto b]$. We show that our definition behaves as expected in lemma *vec-upd-eq* below.

definition *vec-upd* :: $('a\hat{}'b) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'a\hat{}'b$
where *vec-upd s i a* = $(\chi\ j.\ ((\$)\ s)(i := a))\ j)$

lemma *vec-upd-eq*: *vec-upd s i a* = $(\chi\ j.$ *if* $j = i$ *then a else* $s\$j)$
by (*simp add*: *vec-upd-def*)

Based on this, the definition of assignments is as in Section 4.3. Its rule for verification condition generation just requires to unfold definitions and call lemma *wp-rel*.

definition *assign* :: $'b \Rightarrow ('a\hat{}'b \Rightarrow 'a) \Rightarrow ('a\hat{}'b)$ *rel* ((*2*- ::= -) [*70*, *65*] *61*)
where $(x ::= e) = \{(s,$ *vec-upd s x* $(e\ s))|\ s.$ *True*$\}$

lemma *wp-assign* [*simp*]: *wp* $(x ::= e)\ \lceil Q \rceil = \lceil \lambda s.\ Q\ (\chi\ j.\ (((\$)\ s)(x := (e\ s)))\ j) \rceil$
unfolding *wp-rel vec-upd-def assign-def* **by** (*auto simp*: *fun-upd-def*)

The construction of hybrid programs and the derivation of their rules of inference that we have described so far is a relatively fast way to build verification components for hybrid systems in a proof assistant. There are two main obstacles to achieving their implementation. The first consists in deriving all the necessary properties in Kleene algebras to derive the rules of Hoare logic or the weakest liberal precondition calculus. In Isabelle/HOL however, this is trivial as such derivations are available in AFP entries that precede our work [8,54]. Secondly, the shallow embedding of the verification components also requires a sufficient formalisation of relations, functions and ordinary differential equations. Yet, once attained, it simplifies the construction of verification tools for hybrid systems. For instance, function updates and their properties replace implementation nuances about assignments and substitutions. The approach also benefits from the compositionality of Kleene algebras, as new models can be quickly instantiated. Looking at the syntax

$$X \ ::= \ x := e \mid x' = f \,\&\, G \mid ?P \mid X\,;X \mid X + X \mid X^*,$$

this means that formalisation of the Kleene algebras provide us with four out of six hybrid programs. In the following chapters, we explore variations to this implementation to make them faster, more expressive or more flexible.

Chapter 5

Verification Components

So far, we have shown a mathematical roadmap for deriving a minimal logic for verification of hybrid programs, $\mathsf{d}\mathcal{H}$, and a hybrid *wlp*-calculus. However, we have not finished its development in Isabelle/HOL nor verified any hybrid system with the resulting verification components. In this chapter, we tackle those missing discussions. Specifically, we give further mathematical background to implement the components in a general manner that also allows us to reproduce alternative verification procedures from $\mathsf{d}\mathcal{L}$. Specifically, those involving differential invariants [107]. Then, we formalise our approach in Isabelle/HOL and derive the most important rules of $\mathsf{d}\mathcal{L}$ with it.

As stated before, in order to formalise invariants shallowly in a proof assistant, we generalise our semantics for evolution commands in Section 5.1. Then, in Section 5.2 we use this semantics to generalise invariant sets for dynamical systems and implement verification with differential invariants as in $\mathsf{d}\mathcal{L}$. In Section 5.3, we present another way to verify hybrid systems based on dynamical systems rather than using differential equations. In Sections 5.4 and 5.5, we formalise all these developments in Isabelle/HOL which finishes our construction of verification components for hybrid systems. Finally, we use Section 5.6 to derive the most important axioms and rules of inference of $\mathsf{d}\mathcal{L}$ with our components. This guarantees that, in our framework, verification is feasible not only via differential Hoare logic $\mathsf{d}\mathcal{H}$ and our hybrid *wlp*-calculus, but also via our derived rules in the style of $\mathsf{d}\mathcal{L}$.

5.1 Generalised Semantics for Evolution Commands

The verification procedure presented in Section 4.4 requires certifying solutions to systems of ODEs. However, there are many systems of ODEs for which it is impossible to supply analytic solutions. In theory, given that our work is inside a general purpose proof assistant, we could formalise non-analytic functions and still supply them for the verification. Nevertheless, in practice said approach would take too much time and manipulations of such complicated functions are non-trivial. Yet, there are alternative mathematical methods for solving the verification problem. For instance, one can try to deduce properties about the vector field before even trying to find a solution for it. A well-studied method in $\mathsf{d}\mathcal{L}$, involves analysing the properties of the vector field in the form of invariants of the system [107]. Therefore, one can obtain partial correctness specifications for evolution commands without generating

the solutions to the ODEs. This is feasible because $\mathsf{d}\mathcal{L}$ provides in terms and rules to reason about its invariants. In our case, we have two options: provide a deep embedding of $\mathsf{d}\mathcal{L}$'s syntax for invariants or develop the approach semantically. We opt for the second one as it opens new avenues for research and fits better with our shallow embedding. Therefore, this section presents a generalisation of evolution commands that is useful to implement invariant reasoning in a general-purpose proof assistant.

The semantics for evolution commands developed in the previous chapter are G-guarded orbits specialised to the trajectories of a locally Lipschitz continuous vector field. However, we can generalise beyond flows and trajectories with definitions that we have already provided. We start with the following set introduced in Section 4.2.

$$\mathsf{Sols}\, f\, U\, S\, t_0\, s = \{X : U\, s \to S \mid (\forall t \in U\, s.\ X'\, t = f\, t (X\, t)) \wedge X\, t_0 = s \wedge t_0 \in U\, s\}.$$

A few remarks about this definition apply to the rest of the chapter. Firstly, we do not impose local Lipschitz continuity, nor even continuity for the vector field $f : T \to S \to S$. Therefore, uniqueness of the solutions to associated IVPs do not necessarily applies. Secondly, instead of using the intervals of existence $T_s \subseteq T$, as provided by Picard-Lindelöf's theorem, we use a function $U : S \to \mathcal{P}\,\mathbb{R}$ mapping each state $s \in S \subseteq \mathbb{R}^V$ to a subset of $\mathbb{R}$. Finally, if $t_0 \in U\, s$, then $\mathsf{Sols}\, f\, U\, S\, t_0\, s$ collects all the local solutions $U\, s \to S$ to the IVP given by f and (t_0, s). Otherwise, $\mathsf{Sols}\, f\, U\, S\, t_0\, s$ is empty.

As a consequence, for each $X \in \mathsf{Sols}\, f\, U\, S\, t_0\, s$ and predicate $G : S \to \mathbb{B}$, we get one G-guarded orbit, namely, $\gamma\, X\, G\, (U\, s)$. To obtain the set of all possible future states of the system described by f that satisfy G, we then patch all these G-guarded orbits in a *G-guarded orbital* via the function $\gamma^f_G : S \to \mathcal{P}\, S$ as

$$\gamma^f_G\, s = \bigcup \{\gamma\, X\, G\, (U\, s) \mid X \in \mathsf{Sols}\, f\, U\, S\, t_0\, s\}.$$

By their type, G-guarded oribtals are state transformers. To see that they can replace our former semantics for evolution commands, the following results suffice.

Proposition 5.1.1. *Let $f : T \to S \to S$ be a vector field, $G : S \to \mathbb{B}$ a guard, and $U : S \to \mathcal{P}\, T$ with $t_0 \in T \subseteq \mathbb{R}$ and $S \subseteq \mathbb{R}^V$. Then*

1. *if $X \in \mathsf{Sols}\, f\, U\, S\, t_0\, s$, then $\gamma\, X\, G\, (U\, s) \subseteq \gamma^f_G\, s$.*
2. *$\gamma^f_G\, s = \{X t \mid t \in U\, s \wedge (\forall \tau \in \downarrow (U\, s)\, t.\ G\, (X\, \tau)) \wedge X \in \mathsf{Sols}\, f\, U\, S\, t_0\, s\}$.*

Furthermore, if f is locally Lipschitz continuous on S and continuous on T, and if $U\, s$ is a subinterval of the longest interval of existence $T_s \subseteq T$ of the unique solution $\varphi^f_s : T_s \to S$ such that $t_0 \in U\, s$, then

$$\gamma^f_G\, s = \gamma\, \varphi^f_s\, G\, (U\, s) = \{\varphi^f_s\, t \mid t \in U\, s \wedge \forall \tau \in \downarrow (U\, s)\, t.\ G\, (\varphi^f_s\, \tau)\}.$$

Hence, we introduce a variant of evolution commands $(x' =_{\mathcal{F}} f\, \&\, G \text{ on } U\, S \text{ at } t_0)$ given by these G-guarded orbitals. As before, the corresponding relational semantics is

$$(x' =_{\mathcal{R}} f\, \&\, G \text{ on } U\, S \text{ at } t_0) = \{(s, X\, t) \mid t \in U\, s \wedge \forall \tau \in \downarrow U\, t.\ G\, (X\, \tau) \wedge X \in \mathsf{Sols}\, f\, U\, S\, t_0\, s\}$$

Once again, rules for computing weakest liberal preconditions or forward box operators are available for this generalisation

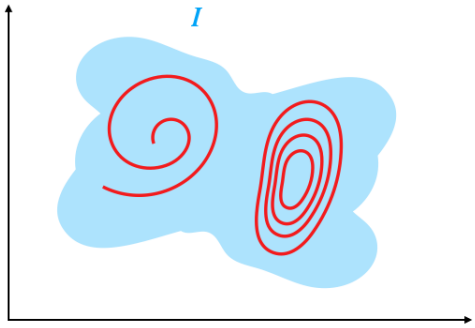


Figure 5.1: Invariants for ODEs contain every orbit that starts inside them

Theorem 5.1.2 (wlp-g-orbital)**.** *Let $f : T \to S \to S$ be a vector field, $G : S \to \mathbb{B}$ a guard, and $U : S \to \mathcal{P}\, T$ with $t_0 \in T \subseteq \mathbb{R}$ and $S \subseteq \mathbb{R}^V$. Then*

$$[x' = f \,\&\, G \text{ on } U\ S \text{ at } t_0]\, Q\, s = \forall X \in \mathsf{Sols}\, f\, U\, S\, t_0\, s. \forall t \in U\, s.\ \mathcal{P}\, X\, (\downarrow U\, t) \subseteq G \to Q\, (X\, t).$$

Similarly, we obtain a Hoare triple for G-guarded orbitals

Theorem 5.1.3 (h-g-orbital)**.** *Let $f : T \to S \to S$ be a vector field, $G : S \to \mathbb{B}$ a guard, and $U : S \to \mathcal{P}\, T$ with $t_0 \in T \subseteq \mathbb{R}$ and $S \subseteq \mathbb{R}^V$. Then*

$$\{\lambda s.\ \forall X{\in}\,\mathsf{Sols}\, f\, U\, S\, t_0\, s. \forall t{\in}U\, s.\ \mathcal{P}\, X\, (\downarrow U\, t) \subseteq G \to Q\, (X\, t)\}\, x' = f \,\&\, G \text{ on } U\ S \text{ at } t_0\, \{Q\}.$$

If an IVP has two solutions, then it has infinite solutions, and models with infinite evolutions in time are unlikely to be of use in applications. However, a benefit from this generalisation is that it enables us to reason with differential invariants. This is the topic of the following section.

5.2 Invariants for Evolution Commands

Recall from Section 3.2 that an invariant for a (hybrid) program α is a property I such that $\{I\}\, \alpha\, \{I\}$. The *differential invariants* of $\mathsf{d}\mathcal{L}$ are exactly the invariants from Section 3.2 specialised to evolution commands. They are inspired by *invariant sets* of dynamical systems theory or (semi)group theory. Namely, for a (semi)group action $\varphi : T \to S \to S$, these are sets $I \subseteq S$ such that $T \cdot I \subseteq I$, where $T \cdot I = \{\varphi\, t\, s \mid t \in T \wedge s \in I\}$. In terms of flows, this means that invariant sets contain all of their orbits $\gamma^\varphi\, s$, that is $\gamma^\varphi\, s = \mathcal{P}\, \varphi^f_s\, T \subseteq I$ for all $s \in I$ [131]. In other words and as depicted in Figure 5.1, every trajectory that starts in the region defined by the invariant remains in the invariant. In this section, we make this idea mathematically precise in terms of our generalisation of guarded orbits.

Therefore, we define an *invariant* for a vector field $f : T \to S \to S$ and guard $G : S \to \mathbb{B}$ along $U : S \to \mathcal{P}\, T$ as a predicate $I : S \to \mathbb{B}$ such that

$$(\gamma^f_G)^\dagger\, I \subseteq I,$$

where we use the Kleisli extension $g^\dagger = \mu \circ \mathcal{P}\, g$ and $\mu_S\, X = \bigcup X$ from the powerset monad as in Section 3.1. Apart from manipulating predicates as sets, we also use $I_1 \wedge I_2$ instead of $(\lambda s.\ I_1\, s \wedge I_2\, s)$ and similar simplifications of notation. With the following result, we provide a relationship between our invariants for vector fields and differential invariants of $\mathsf{d}\mathcal{L}$.

Proposition 5.2.1. *Let $f : T \to S \to S$ be a vector field, $G : S \to \mathbb{B}$ a guard, and $U : S \to \mathcal{P}\, T$ with $t_0 \in T \subseteq \mathbb{R}$ and $S \subseteq \mathbb{R}^V$. Then the following are equivalent.*

1. *I is an invariant for f and G along U.*
2. *$\langle x' = f\,\&\,G \text{ on } U\ S \text{ at } t_0 |\, I \leq I$*
3. *$I \leq |x' = f\,\&\,G \text{ on } U\ S \text{ at } t_0]\, I$*
4. *$\{I\}\, x' = f\,\&\,G \text{ on } U\ S \text{ at } t_0\, \{I\}$*

As observed in Section 3.3, the first two are equivalent because they both are the Kleisli extension of γ^f_G on I. Then, (2) and (3) are equivalent because of the Galois connection between backward diamonds and forward boxes. The equivalence between the last two was also stated in Section 3.3.

When doing verification condition generation, statements (3) and (4) are the most frequently found. By the previous result, we can turn them into the first statement and prove it instead. For this purpose, we list more results about our invariants. In particular, they generalise invariant sets of dynamical systems.

Proposition 5.2.2. *Let $f : T \to S \to S$ be a vector field, $G : S \to \mathbb{B}$ a guard, and $U : S \to \mathcal{P}\, T$ with $t_0 \in T \subseteq \mathbb{R}$ and $S \subseteq \mathbb{R}^V$. Then the following are equivalent*

1. *I is an invariant for f and G along U,*
2. *$(\gamma^f_G\, s) \subseteq I$, for all $s \in I$, and*
3. *for all $s \in I$, $X \in \mathsf{Sols}\, f\, U\, S\, t_0\, s$ and $t \in U\, s$, if $\mathcal{P}\, X\, (\downarrow U\, t) \subseteq G$, then $I\, (X\, t)$.*

Under local Lipschitz continuity and other conditions of Proposition 4.4.1, the second statement becomes the definition of an invariant set. The third statement is a direct application of (wlp-g-orbital) of Theorem 5.1.2 on $I \leq |x' = f\,\&\,G \text{ on } U\ S \text{ at } t_0]\, I$ of Proposition 5.2.1.

The relationship between guards and invariants is also useful. In general, guards do not contribute much in the proof of invariance, but they are required for guaranteeing postconditions. Fortunately, we can take guards out of the evolution command into the postcondition. The following result makes these assertions mathematically precise.

Lemma 5.2.3. *Let $f : T \to S \to S$ be a vector field, $G : S \to \mathbb{B}$ a guard, and $U : S \to \mathcal{P}\, T$ with $t_0 \in T \subseteq \mathbb{R}$ and $S \subseteq \mathbb{R}^V$. Then*

1. *$I \leq |x' = f\,\&\,\top \text{ on } U\ S \text{ at } t_0]\, I \to I \leq |x' = f\,\&\,G \text{ on } U\ S \text{ at } t_0]\, I$, and*
2. *$|x' = f\,\&\,G \text{ on } U\ S \text{ at } t_0]\, Q = |x' = f\,\&\,G \text{ on } U\ S \text{ at } t_0](G \wedge Q)$.*

The first statement follows by antitonicity of forward boxes over the set containment $\gamma_G^f\, s \subseteq \gamma_\top^f\, s$ for all $s \in S$. The second statement holds because, as explained in Section 4.4, both guards and postconditions hold along the entire evolution.

Therefore, in order to prove $P \leq |x' = f \,\&\, G \text{ on } U\, S \text{ at } t_0]\, Q$, we may use an invariant $I \leq |x' = f \,\&\, G \text{ on } U\, S \text{ at } t_0](G \wedge I)$ such that $P \leq I$ and $(G \wedge I) \leq Q$ by (h-cons). That is, the following rule is derivable.

Theorem 5.2.4 (h-inv/wlp-inv)**.** *Let $f : T \to S \to S$ be a vector field, $G : S \to \mathbb{B}$ a guard, and $U : S \to \mathcal{P}\, T$ with $t_0 \in T \subseteq \mathbb{R}$ and $S \subseteq \mathbb{R}^V$. Then*

$$P \leq I \wedge I \leq |x' = f \,\&\, G \text{ on } U\, S \text{ at } t_0]\, G;I \wedge (G;I) \leq Q \;\to P \leq |x' = f \,\&\, G \text{ on } U\, S \text{ at } t_0]\, Q.$$

For a procedure to determine the second conjunct in the antecedent of the above implication without actually solving the system, we rely on the next lemma.

Lemma 5.2.5. *Let $f : T \to S \to S$ be a vector field, $G : S \to \mathbb{B}$ a guard, and $U : S \to \mathcal{P}\, T$ a function that maps states of $S \subseteq \mathbb{R}^V$ to intervals $U\, s$ such that $t_0 \in U\, s \subseteq T \subseteq \mathbb{R}$. If $\mu, \nu : S \to \mathbb{R}$ are differentiable, then*

1. *$\mu = \nu$ is an invariant for f along U if $(\mu \circ X)' = (\nu \circ X)'$ for all X such that $X'\, t = f\, t\, (X\, t)$ for $t \in U\, (X\, t_0)$,*
2. *both $\mu \geq \nu$ and $\mu > \nu$ are invariants for f along U if $(\mu \circ X)'\, t \geq (\nu \circ X)'\, t$ when $t > 0$, and $(\mu \circ X)'\, t \leq (\nu \circ X)'\, t$ when $t < 0$ for all X such that $X'\, t = f\, t\, (X\, t)$ and $G\, (X\, t)$ for $t \in U\, (X\, t_0)$,*
3. *$\neg(\mu \geq \nu)$ is an invariant for f along U if and only if $\mu < \nu$ is also an invariant for f along U,*
4. *$\mu \neq \nu$ is an invariant for f along U if both $\mu > \nu$ and $\nu > \mu$ are (and conversely if t_0 is below every $U\, s$ with $s \in S$).*

Each of these results relies heavily on the mean value theorem. As an example, we use Proposition 5.2.2 (3) for an intuitive argument that explains the first statement. Suppose s satisfies $\mu\, s = \nu\, s$ and let $X \in \mathsf{Sols}\, f\, U\, S\, t_0\, s$, then $X\, t_0 = s$. By the mean value theorem, if two functions coincide at the initial time $\mu\, (X\, t_0) = \nu\, (X\, t_0)$ and their derivatives are the same (by hypothesis), then they remain equal. That is, $\mu\, (X\, t) = \nu\, (X\, t)$ for all $t \in U\, (X\, t_0)$.

As a consequence of these results, the following procedure for proving partial correctness specifications $P \leq |x' = f \,\&\, G \text{ on } U\, S \text{ at } t_0 \textbf{ inv } I]\, Q$ emerges.

1. Check that I is an invariant for f along U
   (a) Transform I into negation normal form.
   (b) If applicable, use (wlp-conj), (wlp-disj) and Lemma 5.2.5 (3) and (4) to obtain positive atomic correctness specifications $I_i \leq |x' = f \,\&\, G \text{ on } U\, S \text{ at } t_0]\, I_i$.
   (c) For each atomic $I_i \leq |x' = f \,\&\, G \text{ on } U\, S \text{ at } t_0]\, I_i$ apply Lemma 5.2.5 (1) and (2).
2. If step 1 is successful, we only need to show $P \leq I$ and $(G \wedge I) \leq Q$ by (wlp-inv).

Just like solutions to systems of ODEs, invariants can also be supplied by external tools [115, 127]. We do not pursue such extensions to our components in this thesis precisely because, scientifically, we know its possible. Instead, we focus on more conceptual contributions detailed in the upcoming chapters.

The example below shows an application of the procedure above. In particular, it evidences the simplicity of its use and its resemblance to mathematical practice. We show the corresponding Isabelle formalisation in Section 5.5.

Example 5.2.1 (Invariant for free fall). A simple differential equation to model the free fall of an object is $x'' t = g$, where $g \in \mathbb{R}$ is the acceleration due to gravity. The equations $f_g\, s\, y = s\, v$ and $f_g\, s\, v = g$ define the corresponding vector field f for variables $y, v \in V$ and $s \in \mathbb{R}^V$. We obtain it by converting $x'' t = g$ to a system of first order ODEs as explained in Section 4.1. In this example, we show that the law of conservation of energy, $I\, s$ defined by

$$I\, s \leftrightarrow \left(-\frac{1}{2} m (s\, v)^2 = mg(h - s\, y)\right),$$

is an invariant for f_g and any guard G along $\mathbb{R}$ for all $s \in \mathbb{R}^V$. Here, m is the mass of the object and h is the height from where it started falling.

Steps $1(a)$ and $1(b)$ do not apply. For, $1(c)$ we should apply Lemma 5.2.5.1. That is, cancelling m above, we need to show that

$$\left(-\frac{1}{2}(X\, t\, v)^2\right)' = (g(h - X\, t\, y))',$$

for all X that solves the system of ODEs and for all $t \in \mathbb{R}$. In particular, from this assumption it follows that $X'\, t\, y = f_g\, (X\, t)\, y = X\, t\, v$ and $X'\, t\, v = f_g\, (X\, t)\, v = g$. The equation above then holds as shown below

$$\left(-\frac{1}{2}(X\, t\, v)^2\right)' = -(X\, t\, v)(X'\, t\, v) = -g(X\, t\, v) = -g(X'\, t\, y) = (g(h - X\, t\, y))'.$$

Therefore, I is an invariant for the system and, by Proposition 5.2.1.3, it holds that

$$I \leq |x' = f_g \,\&\, G \text{ on } \mathbb{R}\, \mathbb{R}^V \text{ at } t_0]\, I.$$

□

As a last remark, we specialise previous results about invariants to the evolution commands of Section 4.4 guarded by the constant true predicate $\top$. In this setting, invariants are strengthened to equalities.

Corollary 5.2.6. *Let $f : T \to S \to S$ be a locally Lipschitz continuous vector field on $S \subseteq \mathbb{R}^V$ and continuous on $T \subseteq \mathbb{R}$, and U a subinterval of the longest interval of existence $T_s \subseteq T$ of the unique solution $\varphi_s^f : T_s \to S$ such that $0 \in U$. Then the following are equivalent.*

1. *I is an invariant for f and $\top$ along U,*
2. *$\langle (x' = f \,\&\, \top)_U |\, I = I$,*

3. $I = |(x' = f \,\&\, \top)_U]\, I$,

4. $\{I\}\,(x' = f \,\&\, \top)_U\,\{I\}$, *and*

5. $(\gamma^\varphi\, s) \subseteq I$, *for all* $s \in I$.

Just like in Section 4.4, we can add (h-inv) of Theorem 5.2.4 to the rules of differential Hoare logic. These allows $\mathsf{d}\mathcal{H}$ to reason about systems of ODEs without solving them. The same applies by adding the respective (wlp-inv) for the weakest liberal precondition calculus. The results presented in this section, generalise our previous work in [72].

5.3 Components Based on Dynamical Systems

So far we have presented mathematical background to develop verification components for hybrid systems based on solutions or invariants for systems of ODEs. In both cases, users supply the solution or the invariant and they need to certify them with the proof assistant. Specifically for the solution based approach, checking Lipschitz continuity and certifying the derivation could represent formalising more than one really needs. In those cases, one could introduce the solution as an invariant but then one needs to certify invariance. That is, for some users, these procedures might seem too pessimistic as they do not trust user-input. Therefore, an alternative approach introduces the dynamics of the physical system from the specification and trusts the correctness of this input. This section adds an innovative variation of evolution commands that adheres to this more optimistic viewpoint. In this setting, correctness of solutions is entirely the responsibility of users or the software they use to describe the dynamics of the system.

The implementation itself is easy as we have already defined all the preliminary concepts in previous sections. We start with a couple of functions $\varphi : T \to S \to S$ and $U : S \to \mathcal{P}\, T$ that, by their types, resemble flows of Section 4.1 and our interval functions of Section 5.1 respectively. Yet, we impose no algebraic or topological structure for T and S to maximise the range of applications for our definitions in this section. Then we just use these functions as arguments of G-guarded orbits to obtain our desired variation of evolution commands,

$$(\mathbf{evol}\,\varphi\, G\, U)\; s = \gamma\,(\lambda t.\; \varphi\, t\, s)\, G\,(U\; s).$$

Hence, φ models the evolution of a system in time. It takes a time $t \in T$ and a state $s \in S$ and outputs the corresponding state $\varphi\, t\, s$. Analogously, $U\, s$ represents a choice of a subdomain of T where φ can evolve. Therefore, $\mathbf{evol}\,\varphi\, G\, U$ represents all the possible states of the system as constrained by G and U. By definition, time T needs to at least be a preorder. However, if T is a monoid, we can require φ to be a monoid action on S, and thus, a dynamical system. As the action can be either discrete or continuous, this new hybrid program generalises evolution commands and it also opens the possibility of integrating discrete systems in correctness specifications. With Isabelle, we can add all these additional constraints via type classes or locales.

Based on our definition, we can compute the *wlp* for these modified evolution commands.

$$|\mathbf{evol}\,\varphi\, G\, U]\, Q\, s = \forall t \in U\; s.\; (\forall \tau \in {\downarrow}(U\; s)\; t.\; G\,(\varphi\, \tau\; s)) \to Q\,(\varphi\, t\, s). \qquad \text{(wlp-g-dyn)}$$

This is of course the same for both the relational and the state transformer semantics. We list the corresponding Hoare rule for a complete presentation.

$$\{\lambda s.\ \forall t \in U\, s.\ (\forall \tau \in {\downarrow}(U\, s)\, t.\ G\,(\varphi\, \tau\, s)) \to Q\,(\varphi\, t\, s)\}\ \mathbf{evol}\ \varphi\, G\, U\, \{Q\}. \tag{h-g-dyn}$$

In those cases where optimistic users can use this alternative approach, most of the procedure of Section 4.4 becomes redundant and they only need to apply rule (wlp-g-dyn) or (h-g-dyn). If a user wishes to certify the solution, they can still quickly check that $(\varphi\, t\, s)' = f\, t\, (\varphi\, t\, s)$ for $t \in U\, s$. Yet, certification of solutions is usually not required in other approaches [39], which simplifies their verification process. If instead, users prefer to certify not only the solution but the fact that it is unique, they can use the procedure of Section 4.4. However, it is not always the case that an analytic solution is available. For these scenarios, our procedure with invariants is still an option. The hybrid program presented in this section is not only a variation of evolution commands, but it is also evidence that our modular approach enables us to quickly add new hybrid programs targeted for specific verification problems. It also shows that our framework provides a good place to experiment by developing different hybrid programs.

5.4 Evolution Commands in Isabelle/HOL

Here we describe our formalisation of our verification components based on certification of solutions. That is, we define guarded orbitals and orbits in Isabelle/HOL and use them to derive their respective rules of $\mathsf{d}\mathcal{H}$ and the hybrid *wlp*-calculus. In summary, this section formalises the concepts and results of Sections 4.4 and 5.1.

As in Section 4.4, we start with the definition of G-guarded orbits and its resemblance to traditional orbits of Lemma 4.4.1, where *down U t* formalises $\downarrow U\, t$.

definition *g-orbit* :: $((\prime a{::}ord) \Rightarrow \prime b) \Rightarrow (\prime b \Rightarrow bool) \Rightarrow \prime a\ set \Rightarrow \prime b\ set\ (\gamma)$
where $\gamma\ X\ G\ U = \bigcup \{\mathcal{P}\ X\ (down\ U\ t)\ |t.\ \mathcal{P}\ X\ (down\ U\ t) \subseteq \{s.\ G\ s\}\}$

lemma *g-orbit-eq*: $\gamma\ X\ G\ U = \{X\ t\ |t.\ t \in U \wedge (\forall \tau{\in}down\ U\ t.\ G\ (X\ \tau))\}$
unfolding *g-orbit-def* **using** *order-trans* **by** *auto blast*

The proof just requires unfolding the definition and calling the *auto* tactic extended with order-transitivity.

For G-guarded orbitals, we define them as in Equation 5.1 and provide their extension of Proposition 5.1.1.2. As before, the proof just calls *auto* and unfolds definitions.

definition *g-orbital* :: $(real \Rightarrow \prime a \Rightarrow \prime a) \Rightarrow (\prime a \Rightarrow bool) \Rightarrow (\prime a \Rightarrow real\ set) \Rightarrow \prime a\ set \Rightarrow real \Rightarrow$
$(\prime a{::}real\text{-}normed\text{-}vector) \Rightarrow \prime a\ set$
where $g\text{-}orbital\ f\ G\ U\ S\ t_0\ s = \bigcup \{\gamma\ X\ G\ (U\ s)\ |X.\ X \in Sols\ f\ U\ S\ t_0\ s\}$

lemma *g-orbital-eq*: $g\text{-}orbital\ f\ G\ U\ S\ t_0\ s =$
$\{X\ t\ |t\ X.\ t \in U\ s \wedge \mathcal{P}\ X\ (down\ (U\ s)\ t) \subseteq \{s.\ G\ s\} \wedge X \in Sols\ f\ U\ S\ t_0\ s\ \}$
unfolding *g-orbital-def ivp-sols-def g-orbit-eq* **by** *auto*

We also formalise the equivalence between orbitals and orbits (part 3 of Proposition 5.1.1) within the context of the locale *picard-lindeloef*.

lemma *g-orbital-orbit*:
  assumes $s \in S$
    and *ivl*: *is-interval* $(U\ s)$
    and *ivp*: $Y \in Sols\ f\ U\ S\ t_0\ s$
    and $U\ s \subseteq T$
  shows *g-orbital* $f\ G\ U\ S\ t_0\ s = g\text{-}orbit\ Y\ G\ (U\ s)$
  ⟨*proof*⟩

In *local-flow*, we provide their full denotation by instantiating to the locale's parameters and we show that guarded orbitals and orbits generalise their traditional counterparts.

context *local-flow*
begin

lemma *g-orbital-collapses*:
  assumes $s \in S$ **and** *is-interval* $(U\ s)$ **and** $U\ s \subseteq T$ **and** $0 \in U\ s$
  shows *g-orbital* $(\lambda t.\ f)\ G\ U\ S\ 0\ s = \{\varphi\ t\ s |\ t.\ t \in U\ s \wedge (\forall \tau \in down\ (U\ s)\ t.\ G\ (\varphi\ \tau\ s))\}$
  apply (*subst g-orbital-orbit*[*of* - - $\lambda t.\ \varphi\ t\ s$], *simp-all add*: *assms g-orbit-eq*)
  by (*rule in-ivp-sols*, *simp-all add*: *assms*)

lemma *orbit-eq*: $s \in S \Longrightarrow \gamma^{\varphi}\ s = \{\varphi\ t\ s |\ t.\ t \in T\}$
  by(*unfold orbit-def*, *subst g-orbital-collapses*, *simp-all add*: *assms init-time interval-time*)

lemma *true-g-orbit-eq*: $s \in S \Longrightarrow g\text{-}orbit\ (\lambda t.\ \varphi\ t\ s)\ (\lambda s.\ True)\ T = \gamma^{\varphi}\ s$
  unfolding *g-orbit-eq orbit-eq*[*OF assms*] **by** *simp*

end

It only remains to derive the rules for verification condition generation of evolution commands. We start by introducing the notation of Section 5.1. The *wlp*-law (wlp-g-orbital) of Theorem 5.1.2 follows immediately by using lemma *wp-rel* of Section 4.5 and unfolding definitions.

definition *g-ode* :: $(real \Rightarrow ('a::banach) \Rightarrow 'a) \Rightarrow 'a\ pred \Rightarrow ('a \Rightarrow real\ set) \Rightarrow 'a\ set \Rightarrow real \Rightarrow$
  $'a\ rel$ $((1x´=- \&\ -\ on\ -\ -\ @\ -))$
  where $(x´= f\ \&\ G\ on\ U\ S\ @\ t_0) = \{(s,s')\ |s\ s'.\ s' \in g\text{-}orbital\ f\ G\ U\ S\ t_0\ s\}$

lemma *wp-g-orbital*: $wp\ (x´= f\ \&\ G\ on\ U\ S\ @\ t_0)\ \lceil Q \rceil =$
  $\lceil \lambda s.\ \forall X \in Sols\ f\ U\ S\ t_0\ s.\ \forall t \in U\ s.\ (\forall \tau \in down\ (U\ s)\ t.\ G\ (X\ \tau)) \longrightarrow Q\ (X\ t) \rceil$
  unfolding *g-orbital-eq wp-rel ivp-sols-def g-ode-def* **by** *auto*

We do this same procedure for guarded orbits to obtain (wlp-g-dyn).

definition *g-evol* :: $(('a::ord) \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b\ pred \Rightarrow ('b \Rightarrow 'a\ set) \Rightarrow 'b\ rel$ (*EVOL*)
where $EVOL\ \varphi\ G\ U = \{(s,s')\ |s\ s'.\ s' \in g\text{-}orbit\ (\lambda t.\ \varphi\ t\ s)\ G\ (U\ s)\}$

lemma *wp-g-dyn*[*simp*]:
fixes $\varphi :: ('a::preorder) \Rightarrow 'b \Rightarrow 'b$
shows $wp\ (EVOL\ \varphi\ G\ U)\ \lceil Q \rceil = \lceil \lambda s.\ \forall\, t{\in}U\ s.\ (\forall\, \tau{\in}down\ (U\ s)\ t.\ G\ (\varphi\ \tau\ s)) \longrightarrow Q\ (\varphi\ t\ s) \rceil$
unfolding *wp-rel g-evol-def g-orbit-eq* **by** *auto*

For (wlp-evol), we instantiate guarded orbitals to the parameters of the *local-flow*.

context *local-flow*
begin

lemma *wp-g-ode-subset*:
assumes $\bigwedge s.\ s \in S \Longrightarrow 0 \in U\ s \wedge is\text{-}interval\ (U\ s) \wedge U\ s \subseteq T$
shows $wp\ (x' = (\lambda t.\ f)\ \&\ G\ on\ U\ S\ @\ 0)\ \lceil Q \rceil =$
$\lceil \lambda s.\ s \in S \longrightarrow (\forall\, t{\in}(U\ s).\ (\forall\, \tau{\in}down\ (U\ s)\ t.\ G\ (\varphi\ \tau\ s)) \longrightarrow Q\ (\varphi\ t\ s)) \rceil$
⟨*proof*⟩

lemma *wp-g-ode*: $wp\ (x' = (\lambda t.\ f)\ \&\ G\ on\ (\lambda s.\ T)\ S\ @\ 0)\ \lceil Q \rceil =$
$\lceil \lambda s.\ s \in S \longrightarrow (\forall\, t{\in}T.\ (\forall\, \tau{\in}down\ T\ t.\ G\ (\varphi\ \tau\ s)) \longrightarrow Q\ (\varphi\ t\ s)) \rceil$
by (*subst wp-g-ode-subset*, *simp-all add*: *init-time interval-time*)

end

Given that lemmas *wp-g-ode-subset* and *wp-g-ode* are inside the *local-flow* locale, to use them in proofs it is necessary to show previously that the conditions of the locale hold. This implies checking that f, T and S satisfy, for instance local Lipschitz continuity, and that φ solves the associated IVP for intervals $[0, t]$ with $t \in T$.

In the formalisations above, it is evident that Isabelle handles theorems about sets automatically. The only lemmas where we need considerable user interaction, because of their long structured proofs, are inside the locales *picard-lindeloef* and *local-flow*. In these cases, we often need to use lemma *unique-solution* of Section 4.2 which requires us to show more analytical properties like existence of derivatives on a subset of the interval T. Nevertheless, our proofs highly resemble those of traditional mathematical textbooks in length and form.

This completes the construction of our verification components for hybrid systems in Isabelle/HOL. In order to verify a hybrid program, users should apply the *wlp*-laws recursively to obtain domain specific proof obligations about predicates and relations or state transformers. In case that the hybrid program involves evolution commands, users can apply either lemma *wp-g-ode-subset* or *wp-g-ode*. This will require them to supply a function φ that must satisfy the conditions of the *local-flow* locale. As explained in the procedure of Section 4.4, they need to prove that f is locally Lipschitz continuous and that φ solves its IVPs. Because of our interest on the mathematical concepts that the framework provides and our interest in exploring its qualitative features, we have delegated the automation of these procedures for future work.

We culminate this section with a formalisation of the hybrid program `thermostat` from

Figure 2.5 and the running Examples 3.2.1, 3.3.1, 4.3.1, 4.3.2, and 4.4.1. As the number of variables is 4, we use Isabelle's finite type of four elements. We fix $1::4$ for variable T, $2::4$ for t, $3::4$ for T_0, and $4::4$ for θ. Then, we formalise the vector field, the guard and the unique solution as in Example 4.4.1 with the χ abstraction to denote vectors.

abbreviation *temp-vec-field* :: *real* ⇒ *real* ⇒ *real^4* ⇒ *real^4* (*f*)
where *f a L s* ≡ (χ *i*. *if i* = *2 then 1 else* (*if i* = *1 then* − *a* ∗ (*s*$*1* − *L*) *else 0*))

abbreviation *therm-guard* :: *real* ⇒ *real* ⇒ *real* ⇒ *real* ⇒ *real^4* ⇒ *bool* (*G*)
where *G Tmin Tmax a L s* ≡ (*s*$*2* ≤ − (*ln* ((*L*−(*if L*=*0 then Tmin else Tmax*))/(*L*−*s*$*3*)))/*a*)

abbreviation *temp-flow* :: *real* ⇒ *real* ⇒ *real* ⇒ *real^4* ⇒ *real^4* (φ)
where φ *a L t s* ≡ (χ *i*. *if i* = *1 then* − *exp*(−*a* ∗ *t*) ∗ (*L* − *s*$*1*) + *L else*
(*if i* = *2 then t* + *s*$*2 else s*$*i*))

The proof for local Lipschitz continuity of Example 4.4.1 requires relatively longer scripting proofs in mathematical style. This is because Isabelle lacks proof automation for arithmetic with real numbers. In Sections 6.5 and 7.6, we discuss ways to avoid this certifications. In the meantime, we have omitted the proofs below and used an identifier ⟨*proof*⟩ for them instead.

lemma *norm-diff-temp-dyn*: *0* < *a* ⟹ ‖*f a L s*$_1$ − *f a L s*$_2$‖ = |*a*| ∗ |*s*$_1$$*1* − *s*$_2$$*1*|
⟨*proof*⟩

lemma *local-lipschitz-temp-dyn*:
assumes *0* < (*a*::*real*)
shows *local-lipschitz UNIV UNIV* (λ*t*::*real*. *f a L*)
⟨*proof*⟩

In contrast, the proof that the vector field and the unique solution satisfy the assumptions of our *local-flow* is simpler.

lemma *local-flow-temp*: *a* > *0* ⟹ *local-flow* (*f a L*) *UNIV UNIV* (φ *a L*)
by (*unfold-locales*, *auto intro*!: *poly-derivatives local-lipschitz-temp-dyn simp*: *forall-4 vec-eq-iff*)

The formalisation of `thermostat` highly resembles Figure 2.5. Below we provide a proof of its correct behaviour. That is, the temperature *s*$*1* remains within the comfortable range.

lemma *thermostat*:
assumes *a* > *0* **and** *0* ≤ *t* **and** *0* < *Tmin* **and** *Tmax* < *L*
shows ⌈λ*s*. *Tmin* ≤ *s*$*1* ∧ *s*$*1* ≤ *Tmax* ∧ *s*$*4* = *0*⌉ ≤ *wp*
(*LOOP* (
— control
((*2* ::= (λ*s*. *0*));(*3* ::= (λ*s*. *s*$*1*));
(*IF* (λ*s*. *s*$*4* = *0* ∧ *s*$*3* ≤ *Tmin* + *1*) *THEN* (*4* ::= (λ*s*.*1*)) *ELSE*
(*IF* (λ*s*. *s*$*4* = *1* ∧ *s*$*3* ≥ *Tmax* − *1*) *THEN* (*4* ::= (λ*s*.*0*)) *ELSE skip*));

```
  — dynamics
  (IF (λs. s$4 = 0) THEN (x´=(λt. f a 0) & G Tmin Tmax a 0 on (λs. {0..t}) UNIV @ 0)
  ELSE (x´=(λt. f a L) & G Tmin Tmax a L on (λs. {0..t}) UNIV @ 0)))
 ) INV (λs. Tmin ≤s$1 ∧ s$1 ≤ Tmax ∧ (s$4 = 0 ∨ s$4 = 1)))
 ⌈λs. Tmin ≤ s$1 ∧ s$1 ≤ Tmax⌉
apply(rule wp-loopI, simp-all add: fbox-temp-dyn[OF assms(1,2)])
 using temp-dyn-up-real-arith[OF assms(1) - - assms(4), of Tmin]
  and temp-dyn-down-real-arith[OF assms(1,3), of - Tmax] by auto
```

The first line in the lemma above lists the assumptions of the verification problem. The second to last line describes the loop invariant. It consists of the postcondition and the fact that the heater is either turned on or off. In the first line of the proof, this loop invariant is picked automatically using the rule (wlp-loop). Using the *wlp* of evolution commands instantiated to the thermostat's vector field in *fbox-temp-dyn*, the structure of the hybrid program is also tackled in this line. This leaves only proof obligations about real numbers, which we have proved separately and named *temp-dyn-up-real-arith* and *temp-dyn-down-real-arith*. We use them together with *auto* to complete the proof.

5.5 Differential Invariants in Isabelle/HOL

We still need to formalise invariants for vector fields along a collection of intervals. Here we describe the addition of these invariants to the verification components. This allows users to prove partial correctness specifications without solving the system of ODEs as explained in Section 5.2. We also certify various results from that section. We start by writing their definition with the Kleisli extension already unfolded.

```
definition diff-invariant :: ('a ⇒ bool) ⇒ (real ⇒ ('a::real-normed-vector) ⇒ 'a) ⇒
 ('a ⇒ real set) ⇒ 'a set ⇒ real ⇒ ('a ⇒ bool) ⇒ bool
 where diff-invariant I f U S t0 G ≡ (⋃ ∘ (𝒫 (g-orbital f G U S t0))) {s. I s} ⊆ {s. I s}
```

As before, proofs about sets are easy to discharge by unfolding definitions. Thus, we immediately obtain the equivalences of Proposition 5.2.2.

```
lemma diff-invariant-eq: diff-invariant I f U S t0 G =
 (∀ s. I s ⟶ (∀ X∈Sols f U S t0 s. (∀ t∈U s.(∀ τ∈(down (U s) t). G (X τ)) ⟶ I (X t))))
 unfolding diff-invariant-def g-orbital-eq image-le-pred by auto


lemma diff-inv-eq-inv-set:
 diff-invariant I f U S t0 G = (∀ s. I s ⟶ (g-orbital f G U S t0 s) ⊆ {s. I s})
 unfolding diff-invariant-eq g-orbital-eq image-le-pred by auto
```

For Proposition 5.2.1, we focus on the equivalence between invariants and *wlp*-specifications. The corresponding Hoare triple and backward diamond are merely notational changes.

```
lemma wp-diff-inv[simp]: (⌈I⌉ ≤ wp (x´= f & G on U S @ t0) ⌈I⌉) = diff-invariant I f U S t0 G
```

unfolding *diff-invariant-eq wp-g-orbital* **by**(*auto simp*: *p2r-def*)

Working towards the procedure of Section 5.2, Lemma 5.2.3 also follows by definition.

lemma *diff-inv-guard-ignore*:
assumes $\lceil I \rceil \leq wp\ (x´= f\ \&\ (\lambda s.\ True)\ on\ U\ S\ @\ t_0)\ \lceil I \rceil$
shows $\lceil I \rceil \leq wp\ (x´= f\ \&\ G\ on\ U\ S\ @\ t_0)\ \lceil I \rceil$
using *assms* **unfolding** *wp-diff-inv diff-invariant-eq* **by** *auto*

lemma *wp-g-orbital-guard*:
assumes $H = (\lambda s.\ G\ s \wedge Q\ s)$
shows $wp\ (x´= f\ \&\ G\ on\ U\ S\ @\ t_0)\ \lceil Q \rceil = wp\ (x´= f\ \&\ G\ on\ U\ S\ @\ t_0)\ \lceil H \rceil$
unfolding *wp-g-orbital* **using** *assms* **by** *auto*

Therefore, we can derive rule (wlp-inv) of Theorem 5.2.4.

lemma *wp-g-odei*:
$\lceil P \rceil \leq \lceil I \rceil \Longrightarrow \lceil I \rceil \leq wp\ (x´= f\ \&\ G\ on\ U\ S\ @\ t_0)\ \lceil I \rceil \Longrightarrow \lceil \lambda s.\ I\ s \wedge G\ s \rceil \leq \lceil Q \rceil \Longrightarrow$
$\lceil P \rceil \leq wp\ (x´= f\ \&\ G\ on\ U\ S\ @\ t_0\ DINV\ I)\ \lceil Q \rceil$
⟨*proof*⟩

For this result, *DINV* is syntactic sugar to add invariants to specifications. Operationally it is the same as our evolution commands from the previous section. That is, $(-)DINV\ I$ is the Isabelle notation for $(-)$ **inv** I of Section 3.2 specific for evolution commands.

Finally, we have proved each one of the clauses of Lemma 5.2.5 as its own theorem in Isabelle/HOL. Just like the tactic for derivatives using the list *poly-derivatives*, we group these clauses in a list called *diff-invariant-rules*. Below we only show the clause for equalities.

named-theorems *diff-invariant-rules rules for certifying differential invariants*.

lemma *diff-invariant-eq-rule* [*diff-invariant-rules*]:
assumes *Uhyp*: $\bigwedge s.\ s \in S \Longrightarrow is\text{-}interval\ (U\ s)$
$\bigwedge X.\ (D\ X = (\lambda \tau.\ f\ \tau\ (X\ \tau))\ on\ U(X\ t_0)) \Longrightarrow (D\ (\lambda \tau.\ \mu(X\ \tau) - \nu(X\ \tau)) = ((*_R)\ 0)\ on\ U(X\ t_0))$
shows *diff-invariant* $(\lambda s.\ \mu\ s = \nu\ s)\ f\ U\ S\ t_0\ G$
⟨*proof*⟩

This completes the procedure of Section 5.2. To verify a correctness specification of the form $\lceil P \rceil \leq wp\ (x´= f\ \&\ G\ on\ U\ S\ @\ t_0\ DINV\ I)\ \lceil Q \rceil$, users need only apply rule *wp-g-odei*. This will generate three proof obligations, one of which is a proof for differential invariance. They can discharge this one by a repeated application of our *diff-invariant-rules*. In combination with *auto* and the list *poly-derivatives*, this process is almost automatically, except when the invariants involve inequalities. In these cases, users need to provide the derivatives of the functions μ and ν of Lemma 5.2.5. To exemplify this, we provide below the formalisation of the automatic proof of invariance in Example 5.2.1 with *s$1* representing variable y and *s$2* doing the same for v.

lemma *diff-invariant* $(\lambda s\ .\ 2 * g * s\$1 - 2 * g * h - s\$2 * s\$2 = 0)\ (f\ g)\ (\lambda s\ .\ UNIV)\ S\ t_0\ G$
by (*auto intro!*: *poly-derivatives diff-invariant-rules*)

For a complete formalisation of Section 5.2, we also include Corollary 5.2.6 below.

context *local-flow*
begin

lemma *wp-diff-inv-eq*:
assumes $\bigwedge s.\ s \in S \Longrightarrow 0 \in U\ s \wedge$ *is-interval* $(U\ s) \wedge U\ s \subseteq T$
shows *diff-invariant* $I\ (\lambda t.\ f)\ U\ S\ 0\ (\lambda s.\ True) =$
$(\lceil \lambda s.\ s \in S \longrightarrow I\ s \rceil = wp\ (x' = (\lambda t.\ f)\ \&\ (\lambda s.\ True)\ on\ U\ S\ @\ 0)\ \lceil \lambda s.\ s \in S \longrightarrow I\ s \rceil)$
⟨*proof*⟩

lemma *diff-inv-eq-inv-set*:
diff-invariant $I\ (\lambda t.\ f)\ (\lambda s.\ T)\ S\ 0\ (\lambda s.\ True) = (\forall s.\ I\ s \longrightarrow \gamma^{\varphi}\ s \subseteq \{s.\ I\ s\})$
unfolding *diff-inv-eq-inv-set orbit-def* **by** (*auto simp*: *p2r-def*)

end

The formalisations presented in the last two sections are an improvement to our 2019 submission to Isabelle's Archive of Fromal Proofs [68]. They extend our entry with the dependency of the interval U on the initial state s and generalise accordingly the results of [72]. In future chapters we explore the benefits of these modifications.

5.6 Derivation of the Axioms of dL

The first implementation of modal reasoning for hybrid programs was differential dynamic logic (dL) [108]. Among other innovations, this approach pioneered the use of differential invariants for proving correctness specifications. Furthermore, numerous case studies evidence its effectiveness and that of its domain-specific proof assistant KeYmaera X [50,77,81,89,96,116]. Therefore, in order to evidence the capabilities of our components, we can emulate dL-style verification. Thus, in this section we derive some axioms and rules of inference of dL that allow us to verify hybrid systems as one would do with this logic. As the rules of dynamic logic come from the axioms of MKA, we focus specifically on the axioms that involve evolution commands.

In general, the rules of inference involving evolution commands in dL are reformulations of the axioms. However, there are different presentations in the literature for them. Hence, we take [112] and KeYmaera X's *cheat sheet* [20] as templates for our derivations in this section. Furthermore, the interval of existence of solutions for the axioms of dL are not explicit in the logic's syntax. For our formalisations, we prefer to leave this parameter general, except when we intend closer resemblence to the dL rule. In those cases, we opt to use $U\ s = \mathbb{R}_{\geq 0}$ and $S = \mathbb{R}^V$. Thus, we include an Isabelle/HOL abbreviation for them:

abbreviation *g-dl-ode* ::(('a::*banach*)⇒'a) ⇒ 'a *pred* ⇒ 'a *rel* ((1*x´*=- & -))
where (*x´*=*f* & *G*) ≡ (*x´*= (λ*t*. *f*) & *G* *on* (λ*s*. {*t*. *t* ≥ 0}) *UNIV* @ 0)

abbreviation *g-dl-ode-inv* :: (('a::*banach*)⇒'a) ⇒ 'a *pred* ⇒ 'a *pred* ⇒ 'a *rel*
where (*x´*= *f* & *G* *DINV* *I*) ≡ (*x´*= (λ*t*. *f*) & *G* *on* (λ*s*. {*t*. *t* ≥ 0}) *UNIV* @ 0 *DINV* *I*)

Our first derivation is axiom DS, formalised below. It has two versions, one with a general vector field f and the other with the constant vector field mapping everything to c. Both cases follow from a simple application of (wlp-evol) or, in Isabelle/HOL, lemma *wp-g-ode-subset*.

lemma *diff-solve-axiom1*:
assumes *local-flow* *f* *UNIV* *UNIV* φ
shows *wp* (*x´*= *f* & *G*) ⌈*Q*⌉ =
⌈λ*s*. ∀ *t*≥0. (∀ τ∈{0..*t*}. *G* (φ τ *s*)) ⟶ *Q* (φ *t* *s*)⌉
by (*subst* *local-flow.wp-g-ode-subset*[*OF* *assms*], *auto*)

lemma *diff-solve-axiom2*:
fixes *c*::'a::{*heine-borel*, *banach*}
shows *wp* (*x´*=(λ*s*. *c*) & *G*) ⌈*Q*⌉ =
⌈λ*s*. ∀ *t*≥0. (∀ τ∈{0..*t*}. *G* (*s* + τ ∗$_R$ *c*)) ⟶ *Q* (*s* + *t* ∗$_R$ *c*)⌉
by (*subst* *local-flow.wp-g-ode-subset*[*OF* *line-is-local-flow*,*of* *UNIV*], *auto*)

The corresponding rule of inference is just one direction of the equivalence in axiom DS but with added preconditions as in Hoare rules.

lemma *diff-solve-rule*:
assumes *local-flow* *f* *UNIV* *UNIV* φ
and ∀ *s*. *P* *s* ⟶ (∀ *t*≥0. (∀ τ∈{0..*t*}. *G* (φ τ *s*)) ⟶ *Q* (φ *t* *s*))
shows ⌈*P*⌉ ≤ *wp* (*x´*= *f* & *G*) ⌈*Q*⌉
using *assms* **by** (*subst* *local-flow.wp-g-ode-subset*[*OF* *assms*(1)], *auto*)

Alternatively, for working with invariants, we derive three axioms of dL with their respective rules of inference. Applied in a backward style, the axiom and rule for *differential cuts* (DC) serve to introduce invariants inside guards.

lemma *diff-cut-axiom*:
assumes *wp* (*x´*= *f* & *G* *on* *U* *S* @ t_0) ⌈*C*⌉ = *Id*
shows *wp* (*x´*= *f* & *G* *on* *U* *S* @ t_0) ⌈*Q*⌉ = *wp* (*x´*= *f* & (λ*s*. *G* *s* ∧ *C* *s*) *on* *U* *S* @ t_0) ⌈*Q*⌉
⟨*proof*⟩

lemma *diff-cut-rule*:
assumes *wp-C*: ⌈*P*⌉ ≤ *wp* (*x´*= *f* & *G* *on* *U* *S* @ t_0) ⌈*C*⌉
and *wp-Q*: ⌈*P*⌉ ≤ *wp* (*x´*= *f* & (λ*s*. *G* *s* ∧ *C* *s*) *on* *U* *S* @ t_0) ⌈*Q*⌉
shows ⌈*P*⌉ ≤ *wp* (*x´*= *f* & *G* *on* *U* *S* @ t_0) ⌈*Q*⌉
⟨*proof*⟩

Recall from Section 4.4 that specifically for evolution commands, postconditions hold

along the entire orbit. Hence, intuitively, the DC axiom above states that if the cut C is satisfied along the evolution, then it is safe to annotate it in the guard. Similarly, in the assumptions of its respective rule, G, C and Q hold along the evolution if starting in a state satisfying P, hence Q trivially holds too.

Observe that the DC rule generates two proof obligations labelled *wp-C* and *wp-Q*. If C is an invariant for the system, the *differential induction* (DI) rule discharges the *wp-C* branch. Hence, we formalise it and its axioms below.

lemma *diff-inv-axiom1*:
assumes $G\ s \longrightarrow I\ s$ **and** *diff-invariant* $I\ (\lambda t.\ f)\ (\lambda s.\ \{t.\ t \geq 0\})\ UNIV\ 0\ G$
shows $(s,s) \in wp\ (x´= f\ \&\ G)\ \lceil I \rceil$
using *assms* **unfolding** *wp-g-orbital diff-invariant-eq* **apply** *clarsimp*
by (*erule-tac x=s* **in** *allE*, *frule ivp-solsD*(*2*), *clarsimp*)

lemma *diff-inv-axiom2*:
assumes *picard-lindeloef* $(\lambda t.\ f)\ UNIV\ UNIV\ 0$
and $\bigwedge s.\ \{t::real.\ t \geq 0\} \subseteq picard\text{-}lindeloef.ex\text{-}ivl\ (\lambda t.\ f)\ UNIV\ UNIV\ 0\ s$
and *diff-invariant* $I\ (\lambda t.\ f)\ (\lambda s.\ \{t::real.\ t \geq 0\})\ UNIV\ 0\ G$
shows $wp\ (x´= f\ \&\ G)\ \lceil I \rceil = wp\ \lceil G \rceil\ \lceil I \rceil$
⟨*proof*⟩

lemma *diff-inv-rule*:
assumes $\lceil P \rceil \leq \lceil I \rceil$ **and** *diff-invariant* $I\ f\ U\ S\ t_0\ G$ **and** $\lceil I \rceil \leq \lceil Q \rceil$
shows $\lceil P \rceil \leq wp\ (x´= f\ \&\ G\ on\ U\ S\ @\ t_0)\ \lceil Q \rceil$
apply(*rule wp-g-orbital-inv*[*OF assms*(*1*) - *assms*(*3*)])
unfolding *wp-diff-inv* **using** *assms*(*2*) **.**

The first axiom says that if I is an invariant of the system and a state s satisfies the implication $G \rightarrow I$, then the state also satisfies the forward box of I after the evolution command. The reason is that the implication guarantees that I holds at the beginning of the evolution. This and invariance guarantees I for the rest of the orbit.

The formalisation of the second axiom requires us to make an assumption of $\mathsf{d}\mathcal{L}$ explicit. Namely, that the vector field satisfies Picard-Lindelöf's theorem. This holds in $\mathsf{d}\mathcal{L}$ because its vector fields correspond to terms of the language of real arithmetic which satisfy this hypothesis [108]. Due to the fact that we fix $U\ s = \mathbb{R}_{\geq 0}$, we have to add the second assumption not listed in the typical presentation of $\mathsf{d}\mathcal{L}$-axioms. The hypothesis states that the nonnegative real numbers are a subset of the interval of existence. The axiom says that if I is a differential invariant, its satisfaction as a postcondition of evolution commands is equivalent to showing that I follows after G. The corresponding rule of DI is an alternative version of our (wlp-inv) of Theorem 5.2.4.

To discharge the *wp-Q* branch of rule DC, $\mathsf{d}\mathcal{L}$ takes advantage of the richer guard. Thus it uses a *differential weakening* axiom that essentially says that if the guard implies the postcondition, then the postcondition holds.

lemma *diff-weak-axiom1*: $Id \subseteq wp\ (x´= f\ \&\ G\ on\ U\ S\ @\ t_0)\ \lceil G \rceil$
unfolding *wp-rel g-ode-def g-orbital-eq* **by** *auto*

lemma *diff-weak-axiom2*:
wp (*x´= f & G on T S @ t*$_0$) ⌈*Q*⌉ = *wp* (*x´= f & G on T S @ t*$_0$) ⌈λ *s*. *G s* ⟶ *Q s*⌉
unfolding *wp-g-orbital image-def* **by** *force*

lemma *diff-weak-rule*:
assumes ⌈*G*⌉ ≤ ⌈*Q*⌉
shows ⌈*P*⌉ ≤ *wp* (*x´= f & G on U S @ t*$_0$) ⌈*Q*⌉
using *assms* **by** (*auto simp*: *wp-g-orbital*)

The rules derived so far provide evidence of the fact that our components serve as a verification tool for hybrid programs. But they also complement our previous derivations and procedures in terms of weakest liberal preconditions or Hoare rules. It remains to prove *differential effect* (DE) and *differential ghost* (DG) rules of dL [22]. These rules serve as tools to reason about complicated solutions of differential equations like exponentials, logarithms, sines and cosines. They are necessary for dL because its syntax resides on a first order setting for real numbers. However, our components lie in the higher order logic of Isabelle, which means that we can explicitly write and reason about those complicated solutions in the proof assistant. In some cases, that approach may be undesirable which is why, in Section 7.6, we discuss alternative ways to tackle problems that require these axioms. This is why we refrain from formalising DE and DG. However, we still intend to develop a general semantic approach that subsumes them.

Finally, a different hybrid program often used in dL corresponds to nondeterministic assignments. This is an assignment of a variable to an undetermined variable. Because of the shallowness of our constructions in Isabelle/HOL, we can quickly formalise it and provide a rule for verification condition generation.

definition *nondet-assign* :: *'b* ⇒ (*'a^'b*) *rel* ((*2- ::= ? *) [*70*] *61*)
where (*x ::= ?*) = {(*s*,*vec-upd s x k*)|*s k*. *True*}

lemma *wp-nondet-assign*[*simp*]: *wp* (*x ::= ?*) ⌈*P*⌉ = ⌈λ*s*. ∀ *k*. *P* (χ *j*. ((($) *s*)(*x := k*)) *j*)⌉
unfolding *wp-rel nondet-assign-def vec-upd-eq* **apply**(*clarsimp*, *safe*)
by (*erule-tac x*=(χ *j*. *if j* = *x then k else s* $ *j*) **in** *allE*, *auto*)

As easily as we added this hybrid program, we can also extend or modify our components. In the following chapter, we explore the extensibility and modularity of our approach.

This concludes the formalisations of this chapter. Including preliminary lemmas to derive the main results displayed here, they amount to approximately 20 pages of proofs in Isabelle/HOL. Furthermore, we have not only derived a weakest liberal precondition calculus and a Hoare logic for verification of hybrid programs, but we have also shown that our components can simulate the reasoning style of dL. Furthermore, we have provided a generalisation for evolution commands that removes the need for intermediate certifications required in the procedure of Section 4.4. The content of this chapter is an extension to our previous work in [71] and [72].

Chapter 6

Extensions

So far, we have described a generic method to build verification components for hybrid systems within the proof assistant Isabelle/HOL. We start with an algebraic abstraction of hybrid programs and then provide a semantics to it. Then we instantiate this semantics to a concrete program store where we derive the rules of verification condition generation for each hybrid program. We summarise this process in Figure 6.1.

Furthermore, Figure 6.1 also describes the contents of this chapter, that is, we explore in more detail the modularity and extensibility of our approach for building verification components. Specifically, in Section 6.1 we show how extending Kleene algebras with tests allows us to derive a refinement calculus for hybrid systems. Secondly, in Sections 6.2 and 6.3, we replace the approach based on Kleene algebras with direct encodings of forward box operators as predicate transformers. Section 6.4 varies the program store model and uses lenses instead of our vectors $s \in \mathbb{R}^V$. Section 6.5 explains a formalisation of affine systems of differential equations in Isabelle/HOL. It evidences that improvements to the libraries of the proof assistant reverberate in the verification components. Finally, we culminate the chapter with a review of all the available verification components, their use cases and features in Section 6.6.

6.1 Differential Refinement Calculi

Our first modification to our components consists in adding a refinement operation $[-,-] : B \times B \to K$ to our Kleene algebras with tests, where B is the subjacent boolean algebra. The refinement must satisfy for all $\alpha \in K$ and $p, q \in B$,

$$\{p\}\,\alpha\,\{q\} \leftrightarrow \alpha \leq [p,q].$$

If this holds, we get a *refinement Kleene algebra with tests* (rKAT) [9]. Easy consequences of this definition include $\{p\}\,[p,q]\,\{q\}$ and $\{p\}\,\alpha\,\{q\} \to \alpha \leq [p,q]$. That is, $[p,q]$ is the greatest element α in K satisfying the Hoare triple $\{p\}\,\alpha\,\{q\}$. Hence, $[p,q]$ works as a partial correctness version of Morgan's *specification statement* [99] which justifies our notation.

Furthermore, just like KAT provides us with rules of propositional Hoare logic, we can derive in rKAT variants to Carroll Morgan's propositional refinement laws [9]. These are

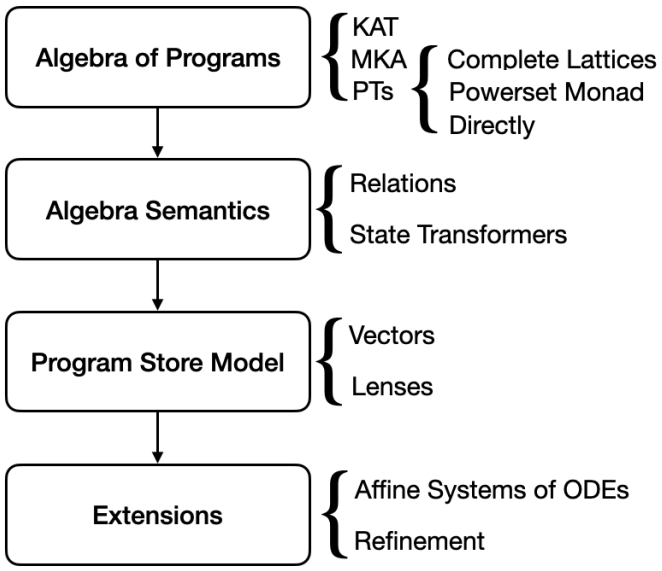


Figure 6.1: Alternatives in the construction of verification components in Isabelle/HOL

consequences of the definition of the refinement operation and the rules of Section 3.2.

$$\mathbf{skip} \leq [p, p], \tag{r-skip}$$
$$\mathbf{abort} \leq [p, q], \tag{r-abort}$$
$$[p', q'] \leq [p, q], \quad \text{if } p \leq p' \text{ and } q' \leq q, \tag{r-cons}$$
$$[p, r] \,;\, [r, q] \leq [p, q], \tag{r-seq}$$
$$[p, q] + [p, q] \leq [p, q], \tag{r-choice}$$
$$\mathbf{if}\ t\ \mathbf{then}\ [t \,;\, p, q]\ \mathbf{else}\ [\neg t \,;\, p, q] \leq [p, q], \tag{r-cond}$$
$$\mathbf{while}\ t\ \mathbf{do}\ [t \,;\, p, p] \leq [p, \neg t \,;\, p], \tag{r-while}$$
$$\mathbf{loop}\ [p, p] \leq [p, p]. \tag{r-loop}$$

We can also obtain modified versions of our invariant rules and, specifically for refinement, we derive monotonic rules for some hybrid programs.

$$\mathbf{if}\ t\ \mathbf{then}\ \alpha_1\ \mathbf{else}\ \beta_1 \leq \mathbf{if}\ t\ \mathbf{then}\ \alpha_2\ \mathbf{else}\ \beta_2, \quad \text{if } \alpha_1 \leq \alpha_2 \text{ and } \beta_1 \leq \beta_2; \tag{r-cond-mono}$$
$$\mathbf{while}\ t\ \mathbf{do}\ \alpha_1 \leq \mathbf{while}\ t\ \mathbf{do}\ \alpha_2, \quad \text{if } \alpha_1 \leq \alpha_2; \tag{r-while-mono}$$
$$\mathbf{loop}\ \alpha_1 \leq \mathbf{loop}\ \alpha_2, \quad \text{if } \alpha_1 \leq \alpha_2; \tag{r-loop-mono}$$
$$\mathbf{while}\ t\ \mathbf{do}\ \alpha\ \mathbf{inv}\ i \leq [p, q] \quad \text{if } p \leq i \,;\, t \text{ and } \alpha \leq [i, i] \text{ and } \neg t \,;\, i \leq q; \tag{h-while-inv}$$
$$\mathbf{loop}\ \alpha\ \mathbf{inv}\ i \leq [p, q] \quad \text{if } p \leq i \text{ and } \alpha \leq [i, i] \text{ and } i \leq q. \tag{h-loop-inv}$$

Finally, despite not using them in proofs, we get $\alpha \leq [\mathbf{abort}, \mathbf{skip}]$ and $[\mathbf{abort}, \mathbf{abort}] \leq \alpha$.

In both, our relational and state transformer semantics for hybrid programs based on states $s \in S \subseteq \mathbb{R}^V$, the refinement operation corresponds to its algebraic intuition of the greatest element satisfying the Hoare triple. That is,

$$[P, Q]_{\mathcal{R}} = \bigcup \{R \subseteq S \times S \mid \{P\}\, R\, \{Q\}\}\,, \quad \text{and} \quad [P, Q]_{\mathcal{F}}\, s = \bigcup \{f\, s \in \mathcal{P}\, S \mid \{P\}\, f\, \{Q\}\}\,.$$

Alternatively, one can simply regard them as a program that relates each input satisfying P to all outputs where Q holds.

Therefore, instantiating to these concrete semantics we also obtain refinement laws for assignments as well as their *leading* and *following* laws that introduce assignments before or after another hybrid program [9].

$$(x := e) \leq [\lambda s.\ Q[x \mapsto e\,s], Q]\,, \tag{r-assgn}$$
$$(x := e)\,;[Q,Q] \leq [\lambda s.\ Q[x \mapsto e\,s], Q], \tag{r-assgnl}$$
$$[Q, \lambda s.\ Q[x \mapsto e\,s]]\,;(x := e) \leq [Q,Q]. \tag{r-assgnf}$$

We can do similar derivations for each of our versions for evolution commands. We start with the version based on vector fields and unique solutions of Section 4.4.

Lemma 6.1.1. *Let $U : S \to \mathcal{P}\,T$ and G, Q be predicates. Suppose that $f : T \to S \to S$ is a locally Lipschitz continuous vector field on $S \subseteq \mathbb{R}^V$ and continuous on $T \subseteq \mathbb{R}$. If for each $s \in S$, $U\,s$ is a subinterval of the longest interval of existence $T_s \subseteq T$ of the unique solution $\varphi_s^f : T_s \to S$ with $0 \in U\,s$, then*

$$(x' = f\,\&\,G)_U \leq [\lambda s \in S. \forall t \in U\,s.\ (\forall \tau \in {\downarrow}(U\,s)\,t.\ G\,(\varphi_s^f\,\tau)) \to Q\,(\varphi_s^f\,t), Q], \tag{r-evol}$$
$$(x' = f\,\&\,G)_U\,;[Q,Q] \leq [\lambda s \in S. \forall t \in U\,s.\ (\forall \tau \in {\downarrow}U\,t.\ G\,(\varphi_s^f\,\tau)) \to Q\,(\varphi_s^f\,t), Q], \tag{r-evoll}$$
$$\big[Q, \lambda s \in S. \forall t \in U\,s.\ (\forall \tau \in {\downarrow}U\,t.\ G\,(\varphi_s^f\,\tau)) \to Q\,(\varphi_s^f\,t)\big]\,;(x' = f\,\&\,G)_U \leq [Q,Q]. \tag{r-evolr}$$

For guarded orbitals and orbits of Sections 5.1 and 5.3, similar derivations with leading and following laws are available. Here, we just write those for the variants of evolution commands where we directly write the dynamics $\varphi : T \to S \to S$ in specifications.

$$\mathbf{evol}\,\varphi\,G\,U \leq [\lambda s. \forall t \in U\,s.\ (\forall \tau \in {\downarrow}(U\,s)\,t.\ G\,(\varphi\,\tau\,s)) \to Q\,(\varphi\,t\,s), Q], \tag{r-g-dyn}$$
$$(\mathbf{evol}\,\varphi\,G\,U)\,;[Q,Q] \leq [\lambda s. \forall t \in U\,s.\ (\forall \tau \in {\downarrow}(U\,s)\,t.\ G\,(\varphi\,\tau\,s)) \to Q\,(\varphi\,t\,s), Q], \tag{r-g-dynl}$$
$$[Q, \lambda s. \forall t \in U\,s.\ (\forall \tau \in {\downarrow}(U\,s)\,t.\ G\,(\varphi\,\tau\,s)) \to Q\,(\varphi\,t\,s)]\,;(\mathbf{evol}\,\varphi\,G) \leq [Q,Q]. \tag{r-g-dynr}$$

Again, these latter variants avoid the need to certify solutions or prove uniqueness. Therefore, they simplify the procedure of Section 4.4.

Together, the laws stated in this section form our differential refinement calculus $\mathtt{d}\mathcal{R}$. If applied step by step, they work incrementally and compositionally to construct a hybrid program that abides to a given initial specification $[p, q]$. The following example shows the formalised approach at hand.

Example 6.1.1. In Section 5.4, we verified the correct behaviour of the `thermostat` hybrid program of Figure 2.5. An alternative procedure uses our $\mathtt{d}\mathcal{R}$ laws to incrementally build it. We start by providing an abbreviation for the loop invariant saying that the temperature *s$1* is within the comfortable range and that the heater *s$4* is either turned on or off.

abbreviation *therm-loop-inv* :: *real* ⇒ *real* ⇒ *real^4* ⇒ *bool* (*I*)
where *I Tmin Tmax s* ≡ *Tmin* ≤ *s$1* ∧ *s$1* ≤ *Tmax* ∧ (*s$4* = *0* ∨ *s$4* = *1*)

Then we refine the two possible continuous dynamics for `thermostat`, when the temperature of the room decreases, and when it rises.

lemma *R-therm-dyn-down*:
assumes $a > 0$ **and** $0 \leq \tau$ **and** $0 < Tmin$ **and** $Tmax < L$
shows *rel-R* $\lceil \lambda s.\ s\$4 = 0 \wedge I\ Tmin\ Tmax\ s \wedge s\$2 = 0 \wedge s\$3 = s\$1 \rceil\ \lceil I\ Tmin\ Tmax \rceil \geq$
$(x' = (\lambda t.\ f\ a\ 0)\ \&\ G\ Tmin\ Tmax\ a\ 0\ on\ (\lambda s.\ \{0..\tau\})\ UNIV\ @\ 0)$
apply(*rule local-flow.R-g-ode-ivl*[*OF local-flow-therm*])
using *assms therm-dyn-down-real-arith*[*OF assms*(1,3), *of* - *Tmax*] **by** *auto*

lemma *R-therm-dyn-up*:
assumes $a > 0$ **and** $0 \leq \tau$ **and** $0 < Tmin$ **and** $Tmax < L$
shows *rel-R* $\lceil \lambda s.\ s\$4 \neq 0 \wedge I\ Tmin\ Tmax\ s \wedge s\$2 = 0 \wedge s\$3 = s\$1 \rceil\ \lceil I\ Tmin\ Tmax \rceil \geq$
$(x' = (\lambda t.\ f\ a\ L)\ \&\ G\ Tmin\ Tmax\ a\ L\ on\ (\lambda s.\ \{0..\tau\})\ UNIV\ @\ 0)$
apply(*rule local-flow.R-g-ode-ivl*[*OF local-flow-therm*])
using *assms therm-dyn-up-real-arith*[*OF assms*(1) - - *assms*(4), *of Tmin*] **by** *auto*

The first formalisation states that the dynamics when the heater is off refine the precondition-postcondition pair where the loop invariant is preserved, but the heater is initially turned off and the thermostat has already measured the room's temperature. The second is similar but with the heater initially turned on. Their proofs consist of a simple application of (r-evol) and automatically discharging the emerging arithmetic obligations. Then, we can integrate these branches with (r-cond-mono) to obtain the lemma below.

lemma *R-therm-dyn*:
assumes $a > 0$ **and** $0 \leq \tau$ **and** $0 < Tmin$ **and** $Tmax < L$
shows *rel-R* $\lceil \lambda s.\ I\ Tmin\ Tmax\ s \wedge s\$2 = 0 \wedge s\$3 = s\$1 \rceil\ \lceil I\ Tmin\ Tmax \rceil \geq$
$(IF\ (\lambda s.\ s\$4 = 0)\ THEN\ (x' = (\lambda t.\ f\ a\ 0)\ \&\ G\ Tmin\ Tmax\ a\ 0\ on\ (\lambda s.\ \{0..\tau\})\ UNIV\ @\ 0)$
$ELSE\ (x' = (\lambda t.\ f\ a\ L)\ \&\ G\ Tmin\ Tmax\ a\ L\ on\ (\lambda s.\ \{0..\tau\})\ UNIV\ @\ 0))$
⟨*proof*⟩

We can do a similar process for the discrete part of `thermostat`.

lemma *R-therm-assign1*:
rel-R $\lceil I\ Tmin\ Tmax \rceil\ \lceil \lambda s.\ I\ Tmin\ Tmax\ s \wedge s\$2 = 0 \rceil \geq (2 ::= (\lambda s.\ 0))$
by (*auto simp*: *R-assign-rule*)

lemma *R-therm-assign2*:
rel-R $\lceil \lambda s.\ I\ Tmin\ Tmax\ s \wedge s\$2 = 0 \rceil\ \lceil \lambda s.\ I\ Tmin\ Tmax\ s \wedge s\$2 = 0 \wedge s\$3 = s\$1 \rceil \geq$
$(3 ::= (\lambda s.\ s\$1))$
by (*auto simp*: *R-assign-rule*)

lemma *R-therm-ctrl*:
rel-R $\lceil I\ Tmin\ Tmax \rceil\ \lceil \lambda s.\ I\ Tmin\ Tmax\ s \wedge s\$2 = 0 \wedge s\$3 = s\$1 \rceil \geq$
$(2 ::= (\lambda s.\ 0)); (3 ::= (\lambda s.\ s\$1));$
$(IF\ (\lambda s.\ s\$4 = 0 \wedge s\$3 \leq Tmin + 1)\ THEN\ (4 ::= (\lambda s.1))\ ELSE$
$IF\ (\lambda s.\ s\$4 = 1 \wedge s\$3 \geq Tmax - 1)\ THEN\ (4 ::= (\lambda s.0))\ ELSE\ skip)$

⟨*proof*⟩

Finally, we can bring all of these partial results together to obtain our desired specification.

lemma *R-therm-loop*: *rel-R* ⌈*I Tmin Tmax*⌉ ⌈*I Tmin Tmax*⌉ ≥
(*LOOP*
rel-R ⌈*I Tmin Tmax*⌉ ⌈λ*s*. *I Tmin Tmax s* ∧ *s*$*2* = *0* ∧ *s*$*3* = *s*$*1*⌉;
rel-R ⌈λ*s*. *I Tmin Tmax s* ∧ *s*$*2* = *0* ∧ *s*$*3* = *s*$*1*⌉ ⌈*I Tmin Tmax*⌉
INV I Tmin Tmax)
by (*intro R-loop R-seq*, *simp-all*)

lemma *R-thermostat-flow*:
assumes *a* > *0* **and** *0* ≤ τ **and** *0* < *Tmin* **and** *Tmax* < *L*
shows *rel-R* ⌈*I Tmin Tmax*⌉ ⌈*I Tmin Tmax*⌉ ≥
(*LOOP* (
— control
((*2* ::= (λ*s*. *0*));(*3* ::= (λ*s*. *s*$*1*));
(*IF* (λ*s*. *s*$*4* = *0* ∧ *s*$*3* ≤ *Tmin* + *1*) *THEN* (*4* ::= (λ*s*.*1*)) *ELSE*
(*IF* (λ*s*. *s*$*4* = *1* ∧ *s*$*3* ≥ *Tmax* − *1*) *THEN* (*4* ::= (λ*s*.*0*)) *ELSE skip*));
— dynamics
(*IF* (λ*s*. *s*$*4* = *0*) *THEN* (*x´*=(λ*t*. *f a 0*) & *G Tmin Tmax a 0 on* (λ*s*. {*0*..*t*}) *UNIV* @ *0*)
ELSE (*x´*=(λ*t*. *f a L*) & *G Tmin Tmax a L on* (λ*s*. {*0*..*t*}) *UNIV* @ *0*)))
) *INV I Tmin Tmax*)
by (*intro order-trans*[*OF - R-therm-loop*] *R-loop-mono*
R-seq-mono R-therm-ctrl R-therm-dyn[*OF assms*]

In *R-therm-loop*, we show that the specifications for the control and the dynamics in a loop refine the loop invariant. Thus, in *R-thermostat-flow* by transitivity, the original program also preserves it. □

In Section 7.5, instead of the compositional style presented here, we show an alternative proof-style to refine a hybrid program with a linear sequence of refinements. As before, changing semantics is very easy. This is why we have not only derived our laws of the differential refinement calculus in the relational setting, but also with the state transformer semantics. The extension is such a straightforward addendum to our previous formalisations, that we only need four pages of proofs per semantics, giving a total of eight pages which corresponds roughly to 400 lines of code.

Due to the fact that MKAs subsume KATs, we can also derive a refinement component in that setting. This would follow Back and von Wright's approach with predicate transformers [12] which we explain further in the next couple of sections. However, due to time constrains, we did not pursue this construction in our submissions to the Archive of Formal Proofs [68]. Furthermore, there is a previous refinement variant for verification of hybrid systems based on d$\mathcal{L}$, namely differential refinement logic [88]. Its focus is on local incremental refinement relations to already verified hybrid programs in order to safely augment them. In contrast, our d$\mathcal{R}$ resembles more the standard Morgan-style approach [99] with global refinement relations to safely build programs from specifications. Yet, their refinement logic has the same expressivity as d$\mathcal{L}$ [88]. Hence, we only need to prove the

soundness of their rules in the MKA-based components as we did in Section 5.6 to attain similar capabilities. The limitations of such a development remain to be explored. In the sequel, we focus instead on alternative algebraic foundations for the development of our verification components.

6.2 Predicate Transformers à la Back and von Wright

As shown in Figure 6.1, the process for developing verification components in Isabelle/HOL is compositional. Thus, instead of our approach with program algebras such as MKAs and KATs, we can use different encodings of predicate transformers. In this section, we focus on Back and von Wright's model as functions between complete lattices [12] as they provide different levels of generality with respect to modal Kleene algebras.

Initially, we consider an alternative algebraic development of Kleene algebras. It begins with *near-quantales* $(S, \leq, \cdot)$, which is a complete lattice with an associative binary composition operation $\cdot$ that preserves suprema in its left argument. In other words, it is a partially ordered set $(S, \leq)$ with suprema $\bigsqcup A$ and infima $\bigsqcap A$ for every subset $A \subseteq S$, such that

$$\left(\bigsqcup A\right) \cdot \beta = \bigsqcup_{\alpha \in A} \alpha \cdot \beta.$$

From this assumption, we can show that composition $\cdot$ preserves the lattice order in its first argument, that is, $\alpha \leq \beta \rightarrow \alpha \cdot \gamma \leq \beta \cdot \gamma$. If in a near-quantale, this also holds true for its second argument, then it is a *prequantale*. On the other hand, if the composition of the near-quantale preserves sups not only in the left argument but in its right argument, then we get a *quantale*. Thus, we can do a similar derivation for order preservation on the second argument to show that quantales are prequantales. Finally, all these structures are *unital* if their composition has a unit 1.

From near-quantales, we get most of the dioid axioms from Section 3.1 by defining as usual $0 = \bigsqcup \emptyset$. We are only missing the identity laws for 1, right distributivity and the annihilation $\alpha \cdot 0 = 0$. But we obtain the identity laws with unital near-quantales, and the other two we get from quantales. Therefore, to obtain Kleene algebras we only need a Kleene star, which is definable in unital near-quantales via $\alpha^* = \bigsqcup_{i \in \mathbb{N}} \alpha^i$. Its left and right unfold and induction axioms then follow from the axioms of quantales and Kleene's fixpoint theorem. Therefore, the following holds.

Proposition 6.2.1. *Every quantale is a Kleene algebra.*

For a detailed derivation of these algebraic facts with further intermediate structures and their consequences see the Isabelle/HOL formalisation precluding our work in [129].

The next step is to regard predicate transformers as functions $f : L_1 \rightarrow L_2$ between complete lattices $(L_1, \leq_1)$ and $(L_2, \leq_2)$. These are *order preserving* if $\alpha \leq_1 \beta \rightarrow f\,\alpha \leq_2 f\,\beta$, *sup-preserving* if $f \circ \bigsqcup = \bigsqcup \circ \mathcal{P}\, f$ and *inf-preserving* if $f \circ \bigsqcap = \bigsqcap \circ \mathcal{P}\, f$. Notice that either of the latter, sup- or inf-, imply the former. Fixing a complete lattice L, we write $\mathcal{T}(L)$, $\mathcal{T}_{\leq}(L)$,$\mathcal{T}_{\sqcup}(L)$, and $\mathcal{T}_{\sqcap}(L)$, for the set of transformers, order-, sup-, and inf-preserving transformers respectively. In the opposite lattice, sup- and inf- preserving transformers reverse their properties, that is $\mathcal{T}_{\sqcap}(L) = \mathcal{T}_{\sqcup}(L^{op})$. Furthermore, the following result holds [12,53].

Proposition 6.2.2. *For a set X and a complete lattice L, the functions in L^X form a complete lattice with order and sups extended pointwise.*

From this we can define infima, least and greatest elements from sups on L^X in the standard way. Moreover, in L^L we obtain a unit id_L and a composition $\circ$ that preserves sups and infs in its first argument but not always in the second one. In summary, we get the next proposition.

Proposition 6.2.3. *Given a complete lattice L,*

1. *$\mathcal{T}(L)$ and $\mathcal{T}(L^{op})$ are unital near-quantales;*

2. *$\mathcal{T}_{\leq}(L)$ and $\mathcal{T}_{\leq}(L^{op})$ are unital prequantales;*

3. *$\mathcal{T}_{\sqcup}(L)$ and $\mathcal{T}_{\sqcap}(L)$ are unital quantales.*

With this result attained, the predicate transformers $f \in \mathcal{T}_{\sqcup}(L)$ define the modalities of MKA. To see this, recall from Section 3.1 that an alternative characterization of $\langle f|\, P$ is $f^{\dagger}\, P$. Translated to quantale notation this corresponds to sup-preserving transformers, that is, $(\bigsqcup \circ \mathcal{P}\, f)\, P$. In turn, the forward box operator is an inf-preserving transformer in the opposite quantale with the lattice dualised and composition swapped. Explicitly, this is

$$|f]\, P = \left(\bigsqcap \circ \mathcal{P}\, (L \setminus f^{op})\right)\, (\neg\, P).$$

Similar equivalences hold for the remaining modal operators. In comparison with MKA, using this approach over $\mathcal{T}_{\sqcup}(L)$ is less general due to the fact that it implies the finite sups and infs of MKA. Accordingly, this approach over $\mathcal{T}_{\leq}(L)$ is more general as finite sup- and inf-preservation of MKA implies order preservation.

Finally, for tests we can assume that the lattice is a complete boolean algebra. Then we can introduce the predicate transformer $|p](-)$ such that $|p]\, q = p \rightarrow q$, which gives us in $\mathcal{T}_{\sqcap}(L)$ the standard definitions [12],

$$\textbf{if } p \textbf{ then } f \textbf{ else } g = |p] \circ f \sqcap |\neg p] \circ g \qquad \text{and} \qquad \textbf{while } p \textbf{ do } f = (|p] \circ f)^* \circ |\neg p].$$

From the constructions on this section we can derive the *wlp*-rules of Section 3.3, thus obtaining verification components for regular programs. For hybrid program verification we only need to instantiate this approach to the relational or the state transformer model and proceed as in Sections 4.3 and 4.4. There is a proof of the fact that state transformers form quantales in [129]. Yet, the approach generalises to functions in $L_2^{L_1}$ and hence, to categories [12]. Although the formalisation of quantales is available in the AFP [129], we have opted for developing in Isabelle/HOL the categorical approach concretised to predicate transformers from the powerset monad because it is more general. We explain these ideas in the following section.

6.3 Predicate Transformers from the Powerset Monad

Another approach for encoding modal operators of Kleene algebras as predicate transformers uses monads [90]. It can be seen as a direct encoding of the hybrid programs over the relational or state transformer model. To obtain it, we return to the powerset monad $(\mathcal{P}, \eta, \mu)$ of Section 3.1, where $\eta : \mathbf{1}_{\mathbf{Set}} \to \mathcal{P}$ and $\mu : \mathcal{P}^2 \to \mathcal{P}$ such that $\eta_S\, s = \{s\}$ and $\mu_S\, X = \bigcup X$ are natural transformations. Moreover, in contrast to previous chapters and to shorten the length of this section, we include the Isabelle/HOL formalisation after the introduction of each mathematical concept.

As stated before, each monad (T, η, μ) with $T : \mathcal{C} \to \mathcal{C}$ gives rise to its Kleisli category $\mathcal{C}_T$ whose objects are those of $\mathcal{C}$ and whose morphisms are Kleisli arrows $\mathcal{C}_T(X, Y) = \mathcal{C}(X, T\,Y)$ whose composition is $g \circ_{\mathcal{C}_T} f = g^\dagger \circ_{\mathcal{C}} f$, where $(-)^\dagger$ is the Kleisli extension $g^\dagger = \mu \circ_{\mathcal{C}} (T\,g)$. In the case of the powerset monad, $\mathbf{Set}_{\mathcal{P}}$ provides state transformers, and our functors $\mathcal{R}$ and $\mathcal{F}$ make this isomorphic to the category of sets and binary relations **Rel**.

To obtain more general predicate transformers $\langle - | \in \mathbf{Set}(\mathcal{P}\,X, \mathcal{P}\,Y)$, we use the Kleisli extension as a functor that maps the set $X \in \mathbf{Set}_{\mathcal{P}}$ to the pair $(\mathcal{P}\,X, \bigcup) \in \mathbf{Set}^{\mathcal{P}}$ and $f \in \mathbf{Set}_{\mathcal{P}}(X, Y) = \mathbf{Set}(X, \mathcal{P}\,Y)$ to $f^\dagger \in \mathbf{Set}^{\mathcal{P}}(X^\dagger, Y^\dagger) = \mathbf{Set}(\mathcal{P}\,X, \mathcal{P}\,Y)$. Here, $\mathbf{Set}^{\mathcal{P}}$ is the category of *Eilenberg-Moore algebras* of the powerset monad whose objects are pairs $(X, \bigsqcup)$ or complete lattices and its morphisms are sup-preserving functions. Notice that its morphisms coincide with the predicate transformers of Section 6.2. The Isabelle formalisation using $bd_{\mathcal{F}}$ for "backward diamond" is analogous $bd_{\mathcal{F}}\, f = \mu \circ \mathcal{P}\, f$. Moreover, the embedding $\langle - | = (-)^\dagger$ maps to complete boolean algebras $(\mathcal{P}\,X, \bigcup)$. In the context of verification, these correspond to the boolean algebras of MKAs or the complete lattices in the Back and von Wright approach of the previous section. The embedding also has an inverse defined on morphisms by $\langle - |^{-1} = (-) \circ \eta$. The emerging isomorphism therefore preserves compositions, units, and sups, thus respecting our purpose for hybrid program verification. In summary, we get $\mathbf{Set}^{\mathcal{P}} \cong \mathbf{Set}_{\mathcal{P}} \cong \mathbf{Rel}$, where the first isomorphism holds by $\langle - |$, and the second by $\mathcal{F}$.

Similarly, forward boxes are a contravariant functor $|-] : \mathbf{Set}(X, \mathcal{P}\,Y) \to \mathbf{Set}(\mathcal{P}\,Y, \mathcal{P}\,X)$ that maps state transformers into inf-preserving functions on complete lattices. In Isabelle, it is formalised in [130] with a lattice dualisation function ∂ that for boolean algebras is De Morgan's duality, and in this case, set complementation $\partial\ X = UNIV - X$. It allows us to define the functor $\partial_F\, f = \partial \circ f \circ \partial$ which is naturally isomorphic to the identity functor via ∂. Then, the weakest precondition predicate transformer is $|-] = \partial_F \circ \langle - | \circ (-)^{op}$, which unfolds to its expected $|f]\, P = \{s \mid f\, s \subseteq P\}$. In Isabelle, this is automatic by unfolding definitions.

lemma *fb*$_{\mathcal{F}}$ *F S* = {*s*. *F s* ⊆ *S*}
by(*auto simp*: *ffb-def kop-def klift-def map-dual-def dual-set-def f2r-def r2f-def*)

The functor has an inverse $|-]^{-1}$ such that $|g]^{-1}\, x = \bigcap\{P \mid x \in g\, P\}$. The emerging isomorphism thus preserves infs of powerset lattices, its greatest elements, but not always sups and least elements. In turn, the isomorphism to relations is as before with $\mathcal{F}$ and $\mathcal{R}$.

In [130], the formalisation of forward diamonds and backward boxes is as in Section 3.3: $fd_{\mathcal{F}} = bd_{\mathcal{F}} \circ op_K$ and $bb_{\mathcal{F}} = fb_{\mathcal{F}} \circ op_K$. Their inverses similarly are $|-\rangle^{-1} = (-)^{op} \circ \langle - |^{-1}$ and $[-|^{-1} = (-)^{op} \circ |-]^{-1}$. In a mono-typed setting, these operators are coherent with the isomorphisms between relations and state transformers: $|f\rangle = |\mathcal{R}\, f\rangle$, $|R\rangle = |\mathcal{F}\, R\rangle$,

$|f] = |\mathcal{R}\, f]$, $|R] = |\mathcal{F}\, R]$, and, dually, $\langle f| = \langle \mathcal{R}\, f|$, $\langle R| = \langle \mathcal{F}\, R|$, $[f| = [\mathcal{R}\, f|$ and $[R| = [\mathcal{F}\, R|$.

As with predicate transformers from quantales, we can derive the *wlp*-rules of previous Sections. For the complete derivations, see our formalisation in [68].

lemma *ffb-assign*[*simp*]: $ffb_{\mathcal{F}}\ (x ::= e)\ Q = \{s.\ (\chi\ j.\ (((\$)\ s)(x := (e\ s)))\ j) \in Q\}$
unfolding *vec-upd-def assign-def* **by** (*subst ffb-eq*) *simp*

lemma *ffb-kcomp*[*simp*]: $ffb_{\mathcal{F}}\ (G\ ;\ F)\ P = ffb_{\mathcal{F}}\ G\ (ffb_{\mathcal{F}}\ F\ P)$
unfolding *ffb-eq* **by** (*auto simp*: *kcomp-def*)

lemma *ffb-if-then-else*[*simp*]: $ffb_{\mathcal{F}}\ (IF\ T\ THEN\ X\ ELSE\ Y)\ Q =$
$\{s.\ T\ s \longrightarrow s \in ffb_{\mathcal{F}}\ X\ Q\} \cap \{s.\ \neg\ T\ s \longrightarrow s \in ffb_{\mathcal{F}}\ Y\ Q\}$
unfolding *ffb-eq ifthenesle-def* **by** *auto*

lemma *ffb-loopI*: $P \leq \{s.\ I\ s\} \Longrightarrow \{s.\ I\ s\} \leq Q \Longrightarrow \{s.\ I\ s\} \leq ffb_{\mathcal{F}}\ F\ \{s.\ I\ s\}$
$\Longrightarrow P \leq ffb_{\mathcal{F}}\ (LOOP\ F\ INV\ I)\ Q$
unfolding *loopi-def* **using** *ffb-kstarI*[*of P*] **by** *simp*

lemma *fbox-g-evol*[*simp*]:
fixes $\varphi :: ('a{::}preorder) \Rightarrow 'b \Rightarrow 'b$
shows $fb_{\mathcal{F}}\ (EVOL\ \varphi\ G\ U)\ Q = \{s.\ (\forall\, t{\in}U\ s.\ (\forall\, \tau{\in}down\ (U\ s)\ t.\ G\ (\varphi\ \tau\ s)) \longrightarrow (\varphi\ t\ s) \in Q)\}$
unfolding *g-evol-def g-orbit-eq ffb-eq* **by** *auto*

lemma *ffb-g-ode-subset*:
assumes $\bigwedge s.\ s \in S \Longrightarrow 0 \in U\ s \wedge is\text{-}interval\ (U\ s) \wedge U\ s \subseteq T$
shows $fb_{\mathcal{F}}\ (x' = (\lambda t.\ f)\ \&\ G\ on\ U\ S\ @\ 0)\ Q =$
$\{s.\ s \in S \longrightarrow (\forall\, t{\in}(U\ s).\ (\forall\, \tau{\in}down\ (U\ s)\ t.\ G\ (\varphi\ \tau\ s)) \longrightarrow (\varphi\ t\ s) \in Q)\}$
⟨*proof*⟩

Knowing the categorical representation of forward box operators as predicate transformers allows us to go one step further and obtain verification components with a direct encoding. Therefore, instead of relying on previous formalisations of MKAs in [54] or predicate transformers through the powerset monad in [130], we have created a lightweight verification component by directly defining weakest preconditions.

definition *fbox* :: $('a \Rightarrow 'b\ set) \Rightarrow 'b\ pred \Rightarrow 'a\ pred$ (|-] - [61,81] 82)
where $|F]\ P = (\lambda s.\ (\forall\, s'.\ s' \in F\ s \longrightarrow P\ s'))$

By their type, these are indeed predicate transformers. Furthermore, Isabelle's ability to prove properties about them highly increases, as it mostly relies on the representation on the right hand side as a higher order logic formula. The proofs of the *wlp*-rules are the same as those presented above in this section up to renaming of the lemmas.

Both versions of the verification components, the spartan version and the one based on predicate transformers of the powerset monad are a contribution from our work. Each of them consists of approximately 400 lines of code which covers 4 pages of proofs. To build them, we relied on knowledge of the categorical approach to predicate transformers due at least to Manes [92]. However, given the restrictive type system of Isabelle to encode categories, the

formalisations of predicate transformers used here consist only on the particular instance of the powerset monad. There is an alternative formalisation of predicate transformers à la Back and von Wright by Preoteasa [119,120]. However, we used the formalisation in [129,130] for its emphasis on quantales and verification. It includes, for example, a forward box operator for the relational model $fb_{\mathcal{R}}$ that we ended up not using. As the approach is compositional, we can potentially vary the store model like in next section or add refinements like in Section 6.1.

6.4 Lenses

As Figure 6.1 shows, apart form modifying the underlying algebraic structure for the components, we can also modify the model for the program store. This illustrates another section where our framework is modular but also opens the possibility for a generalisation of our state space. Repeating our previous approach with algebras for programs, we can use an algebraic entity to obtain various store models at once. Thus, in this section we use lenses [47] to provide the verification components with the potential to use state space models other than vectors in $\mathbb{R}^V$.

Lenses are an abstract algebraic representation of the program store with access get and mutation put functions to manipulate it. Formally, the four-tuple $x = (A, S, \mathit{get}_x, \mathit{put}_x)$ is a lens [47], denoted $x : A \Longrightarrow S$, if the laws

$$\mathit{get}_x\ (\mathit{put}_x\ s\ v) = v, \qquad \mathit{put}_x\ (\mathit{put}_x\ s\ u)\ v = \mathit{put}_x\ s\ v, \qquad \mathit{put}_x\ s\ (\mathit{get}_x\ s) = s,$$

hold for all $s \in S$ and $v, u \in A$, where $\mathit{get}_x : S \to A$ and $\mathit{put}_x : S \to A \to S$. Intuitively and for our purposes, the lens $x : A \Longrightarrow S$ represents a variable. The view A models the type of the variable which comes from the source S that corresponds to the state space or program store. Modelling the accessing and mutating behaviour with these equations goes back at least to Back and Von Wright [12]. The axioms express that get_x queries the value of x while put_x updates it. Thus, the first axiom states that after updating x's value to be v, querying it will give v. The second axiom says that updating the value of x twice, is the same as updating it once to the second value. Finally, the third equation says that updating the value of x with its current value is the same as doing nothing. For $x \in V$, with V a finite set, the tuple $\mathit{vec\text{-}lens}_x^V = (\mathbb{R}, \mathbb{R}^V, \lambda s.\ s\,x, \lambda s\,t.\ s[x \mapsto t])$ is a lens. Therefore, the set $\{\mathit{vec\text{-}lens}_x^V \mid x \in V\}$ corresponds to the finite dimensional vector model $\mathbb{R}^V$ of the program store that we have used in previous sections.

Two lenses x and y on the same state space S can be independent of each other, represented with the symmetric irreflexive relation $x \bowtie y$. It indicates that they correspond to different regions of S which is formalised with the equations [47]

$$\mathit{get}_x\ (\mathit{put}_y\ s\ v) = \mathit{get}_x\ s, \quad \mathit{get}_y\ (\mathit{put}_x\ s\ u) = \mathit{get}_y\ s, \quad \text{and} \quad \mathit{put}_x\ (\mathit{put}_y\ s\ v)\ u = \mathit{put}_y\ (\mathit{put}_x\ s\ u)\ v.$$

Intuitively, if x and y are independent, updating y on state s has no effect on querying x and vice versa. The commutativity of the third equation means that we can update two independent lenses in any order without affecting the overall result. In the case of our hybrid programs, every variable is independent of the other.

Similarly, given a function f and a lens x, we say that x is unrestricted in f, denoted $x \sharp f$ if for each $s \in S$ and $v \in A$, $f\ (\mathit{put}_x\ s\ v) = f\ s$. Unrestriction serves to formalise the common

provisos in logic and programming for substitutions saying that f does not depend on x [47]. We can use these notions to prove properties about assignments with lenses. To do that, we first define state updates after a function $f : S \to S$ with the equation

$$f(x \mapsto e)\, s = \mathit{put}_x\, (f\, s)\, (e\, s),$$

where $x : A \Longrightarrow S$ and $e : S \to A$. That is, it updates the value of $f\, s$ to the value of the expression e evaluated on s. As in previous sections, e is a function to model dependency on previous states, for instance, to update to $x + y + 1$ for variables x and y, we would use $e\, s = \mathit{get}_x\, s + \mathit{get}_y\, s + 1$. Simultaneous updates for $n \geq 2$ variables then correspond to

$$[x_1 \mapsto_s e_1, x_2 \mapsto_s e_2, \cdots, x_n \mapsto_s e_n] = id(x_1 \mapsto e_1)(x_2 \mapsto e_2) \cdots (x_n \mapsto e_n),$$

because $[x \mapsto_s e, y \mapsto_s f] = [y \mapsto_s f, x \mapsto_s e]$ when $x \bowtie y$, and $[x \mapsto_s e, x \mapsto_s f] = id(x \mapsto e)$ by the second law of lenses. In particular, this notation is useful for defining vector fields $f : S \to S$. The semantics for assignments as state transformers are therefore

$$(x := e)\, s = \{id(x \mapsto e)\, s\} = \{\mathit{put}_x\, s\, (e\, s)\}.$$

Notice that the program algebra semantics remains, but the underlying store semantics has changed. Thus, we can use different lenses as our program store model like records and functions [47]. In Isabelle/HOL, for instance, there is an alternative axiomatisation of vector spaces $\mathbb{V}$ and their ordered bases via type classes [67,76]. Foreseeing applications with code generation, this formalisation includes a list representation $\ell\, s$ of vectors $s \in \mathbb{V}$. Functions $l\,!\,n$ and *list-update* l n v output the nth element of the list l and the update of the nth element of l to the value v respectively. Therefore, the lenses

$$\begin{aligned} \mathit{eucl\text{-}lens}_k^n &= (\mathbb{R}, U, \lambda s.\ (\ell\, s)\,!\,k, \lambda s\, v.\ \mathit{list\text{-}update}\ (\ell\, s)\ k\ v), \qquad \text{for } k < n, \text{ and} \\ \mathit{fun\text{-}lens}_x^{(A,B)} &= (B, A \to B, \lambda f.\ f\, x, \lambda f\, v.\ f[x \mapsto v]), \end{aligned}$$

are possible alternatives in the development of verification components. In particular, lenses $\mathit{eucl\text{-}lens}_k^n$ might be useful for code generation of verified programs. We use them below for an alternative formalisation of the `thermostat` hybrid program of Figure 2.5. It is still unknown if the lenses $\mathit{fun\text{-}lens}_x^{(\mathbb{N},\mathbb{R})}$ are an infinite dimensional alternative for our components.

Another benefit of lenses comes from the extensive formalisation of Isabelle/UTP [45,46]. It provides a rich collection of commands such as **utp-lift-notation** or **U** (below) that enable us to avoid lambdas λ in our formalisation. Thus, instead of using numbers $s\$1$, $s\$2$, $s\$3$, and $s\$4$ as in Example 6.1.1, we can have a cleaner presentation of `thermostat` with lenses.

abbreviation $T :: \mathit{real} \Longrightarrow \mathit{real}\hat{}4$ **where** $T \equiv \Pi[0]$
abbreviation $t :: \mathit{real} \Longrightarrow \mathit{real}\hat{}4$ **where** $t \equiv \Pi[1]$
abbreviation $T_0 :: \mathit{real} \Longrightarrow \mathit{real}\hat{}4$ **where** $T_0 \equiv \Pi[2]$
abbreviation $\vartheta :: \mathit{real} \Longrightarrow \mathit{real}\hat{}4$ **where** $\vartheta \equiv \Pi[3]$

abbreviation *ftherm* :: *real* ⇒ *real* ⇒ (*real*, 4) *vec* ⇒ (*real*, 4) *vec* (*f*)
where $f\ a\ c \equiv [T \mapsto_s -(a * (T - c)),\ T_0 \mapsto_s 0,\ \vartheta \mapsto_s 0,\ t \mapsto_s 1]$

abbreviation *therm-guard* :: *real* ⇒ *real* ⇒ *real* ⇒ *real* ⇒ (*real^4*) *upred* (*G*)
where $G\ T_l\ T_h\ a\ L \equiv \mathbf{U}(t \leq -\ (ln\ ((L-(if\ L=0\ then\ T_l\ else\ T_h))/(L-T_0)))/a)$

abbreviation *therm-loop-inv* :: *real* ⇒ *real* ⇒ (*real^4*) *upred* (*I*)
where $I\ T_l\ T_h \equiv \mathbf{U}(T_l \leq T \wedge T \leq T_h \wedge (\vartheta = 0 \vee \vartheta = 1))$

abbreviation *therm-flow* :: *real* ⇒ *real* ⇒ *real* ⇒ (*real^4*) *usubst* (*φ*)
where $\varphi\ a\ c\ \tau \equiv [T \mapsto_s -\ exp(-a * \tau) * (c - T) + c,\ t \mapsto_s \tau + t,\ T_0 \mapsto_s T_0,\ \vartheta \mapsto_s \vartheta]$

abbreviation *therm-ctrl* :: *real* ⇒ *real* ⇒ (*real^4*) *nd-fun* (*ctrl*)
where *ctrl* $T_l\ T_h \equiv$
$(t ::= 0);\ (T_0 ::= T);$
$(IF\ (\vartheta = 0 \wedge T_0 \leq T_l + 1)\ THEN\ (\vartheta ::= 1)\ ELSE$
$\ IF\ (\vartheta = 1 \wedge T_0 \geq T_h - 1)\ THEN\ (\vartheta ::= 0)\ ELSE\ skip)$

abbreviation *therm-dyn* :: *real* ⇒ *real* ⇒ *real* ⇒ *real* ⇒ *real* ⇒ (*real^4*) *nd-fun* (*dyn*)
where *dyn* $T_l\ T_h\ a\ T_u\ \tau \equiv$
$IF\ (\vartheta = 0)\ THEN\ x' = f\ a\ 0\ \&\ G\ T_l\ T_h\ a\ 0\ on\ \{0..\tau\}\ UNIV\ @\ 0$
$\ ELSE\ x' = f\ a\ T_u\ \&\ G\ T_l\ T_h\ a\ T_u\ on\ \{0..\tau\}\ UNIV\ @\ 0$

abbreviation $therm\ T_l\ T_h\ a\ L\ \tau \equiv LOOP\ (ctrl\ T_l\ T_h\ ;\ dyn\ T_l\ T_h\ a\ L\ \tau)\ INV\ (I\ T_l\ T_h)$

We formalised the alternative verification components using lenses $vec\text{-}lens_x^V$ and $eucl\text{-}lens_k^n$ for [42]. This section is a recapitulation of the main contributions from that reference. The formalisation includes both the state transformer and a relational semantics. It covers 53 pages of proofs which is equivalent to approximately 2400 lines of code. The formalisation of Isabelle/UTP and lenses comes from [45, 46]. We do not use lenses in the verification examples of the following chapter.

6.5 Affine Systems of ODEs

We take most of this section verbatim from our original publication [69] and the corresponding formalisation [70]. This work focuses on extending the libraries of formalised mathematics of Isabelle/HOL directly related to our components. With this extension, we provide evidence for the fact that improvements to the general purpose proof assistant also benefit the verification components. In particular, by formalising certain well-studied classes of ordinary differential equations, we can use their properties to simplify the verification procedure.

An important class of vector fields with unique solutions are those representing *affine* systems of ODEs. They satisfy the equation

$$X'\,t = A\,t \cdot X\,t + B\,t,$$

for matrix-vector multiplication $\cdot$, $n \times n$ matrices $A\,t$ and vectors $B\,t$, where A and B are continuous functions on T. Equally important are the corresponding *linear* systems where $B\,t = 0$ for all $t \in T$. In the time-independent or *autonomous* case where A and B are constant functions, their unique solutions are well-characterised and globally defined. That

is, flows φ for autonomous affine systems exist and satisfy

$$\varphi\, t\, s = \exp(tA) \cdot s + \exp(tA) \cdot \int_0^t (\exp(-\tau A) \cdot B)\, d\tau,$$

where exp is the matrix exponential $\exp A = \sum_{i \in \mathbb{N}} \frac{1}{i!} A^i$. By formalising a generic version of this particular case in Isabelle/HOL, we do the procedure of Section 4.4 for users rather than asking them to do it. Therefore, we explain this formalisation below which not only serves verification but it actually adds definitions and lemmas from mathematics for a seamless integration of the existing libraries of ODEs and linear algebra.

In Isabelle/HOL, matrices are vectors of vectors—an $m \times n$ matrix A has type $real\hat{}'n\hat{}'m$. The product of matrix A with vector s is denoted $A *v s$; the scaling of vector s by real number c is written $c *_R s$. Thus, a solution X to an affine system with $A :: real \Rightarrow real\hat{}'n\hat{}'n$ and $B :: real \Rightarrow real\hat{}'n$ satisfies the predicate $D\ X = (\lambda t.\ A\ t *v X\ t + B\ t)\ on\ T$.

Affine systems of ODEs are locally Lipschitz continuous with respect to the operator norm $\|M\|_{op} = \bigsqcup \{\|M \cdot s\| \mid \|s\| = 1\}$, where M is a matrix with real coefficients. This specific instance of the norm on matrices is one of the contributions of our formalisation. We introduce it by specialising Isabelle's *onorm* that lives in its HOL-Analysis library.

abbreviation *op-norm* :: ('a::*real-normed-algebra-1*)$\hat{}$'n$\hat{}$'m $\Rightarrow$ *real* (*1*$\|$-$\|_{op}$)
  where $\|A\|_{op} \equiv onorm\ (\lambda x.\ A *v x)$

It is an alternative definition of the operator norm $onorm\, f = Sup\ \{\|f\, x\| / \|x\| \mid x \in V\}$. However, for many proofs, our characterisation of $\|-\|_{op}$ above is more convenient. Hence, we formalise the equivalence as shown below.

lemma *op-norm-def*: $\|A\|_{op} = Sup\ \{\|A *v x\| \mid x.\ \|x\| = 1\}$
  ⟨*proof*⟩

We also show that $\|-\|_{op}$ satisfies the norm axioms and other properties.

lemma *op-norm-ge-0*: $0 \leq \|A\|_{op}$
  using *ex-norm-eq-1 norm-ge-zero norm-matrix-le-op-norm basic-trans-rules*(*23*) **by** *blast*

lemma *op-norm-eq-0*: $(\|A\|_{op} = 0) = (A = 0)$
  unfolding *onorm-eq-0*[*OF blin-matrix-vector-mult*] **using** *matrix-axis-0*[*of 1 A*] **by** *fastforce*

lemma *op-norm-triangle*: $\|A + B\|_{op} \leq (\|A\|_{op}) + (\|B\|_{op})$
  ⟨*proof*⟩

lemma *op-norm-scaleR*: $\|c *_R A\|_{op} = |c| * (\|A\|_{op})$
  unfolding *onorm-scaleR*[*OF blin-matrix-vector-mult*, *symmetric*] *scaleR-vector-assoc* **..**

lemma *norm-matrix-le-mult-op-norm*: $\|A *v x\| \leq (\|A\|_{op}) * (\|x\|)$
  ⟨*proof*⟩

lemma *op-norm-matrix-matrix-mult-le*: $\|A ** B\|_{op} \leq (\|A\|_{op}) * (\|B\|_{op})$
⟨*proof*⟩

lemma *op-norm-le-transpose*: $\|A\|_{op} \leq \|transpose\ A\|_{op}$
⟨*proof*⟩

Using these properties, we can show Lipschitz continuity. Indeed, with Lipschitz constant $\ell = \bigsqcup \{\|A\,t\|_{op} \mid t \in \overline{B_\varepsilon(s)}\}$, we get

$$\|(A\,t)\cdot s_1 - (A\,t)\cdot s_2\| = \|(A\,t)\cdot(s_1 - s_2)\| \leq \|A\,t\|_{op}\,\|s_1 - s_2\| \leq \ell\,\|s_1 - s_2\|\,.$$

In particular ℓ exists by continuity of A and $\|-\|$, and compactness of $\overline{B_\varepsilon(s)}$. The formalisation however, does not immediately require continuity for $\|A\|_{op}$, just that it is bounded above using predicate *bdd-above*.

lemma *lipschitz-cond-affine*:
defines $L \equiv Sup\ \{\|A\ t\|_{op} \mid t.\ t \in T\}$
assumes $t \in T$ **and** *bdd-above* $\{\|A\ t\|_{op} \mid t.\ t \in T\}$
shows $\|A\ t *v x - A\ t *v y\| \leq L * (\|x - y\|)$
⟨*proof*⟩

An alternative Lipschitz constant involves the matrix maximum norm $\|A\|_{max}$ which is the greatest absolute value of A's entries [131]. We have also formalised this norm. Below, we show its relationship to the operator norm which we could use to replace ℓ in the last inequality of the Lipschitz argument above.

abbreviation *max-norm* :: *real^'n^'m* ⇒ *real* $(1\|\text{-}\|_{max})$
where $\|A\|_{max} \equiv Max\ (abs\ `\ (entries\ A))$

lemma *max-norm-def*: $\|A\|_{max} = Max\ \{|A\$i\$j| \mid i\ j.\ i{\in}UNIV \wedge j{\in}UNIV\}$
⟨*proof*⟩

lemma *op-norm-le-max-norm*:
fixes *A* :: *real^('n::finite)^('m::finite)*
shows $\|A\|_{op} \leq real\ CARD('m) * real\ CARD('n) * (\|A\|_{max})$
⟨*proof*⟩

Another benefit of our formalisation of the operator norm is that it allows us to define matrix continuity. Together with local Lipschitz continuity, we can show that affine systems satisfy the assumptions of our *picard-lindeloef* locale.

definition *matrix-continuous-on* :: *real set* ⇒ (*real* ⇒ (*'a::real-normed-algebra-1*)*^'n^'m*) ⇒ *bool*
where *matrix-continuous-on T A* = $\forall\, t{\in}T.\forall\,\varepsilon{>}0.\exists\,\delta{>}0.\forall\,\tau{\in}T.\ |\tau - t|{<}\delta \longrightarrow \|A\ \tau - A\ t\|_{op}{\leq}\varepsilon$

lemma *picard-lindeloef-affine*:
fixes *A* :: *real* ⇒ *'a::{banach,real-normed-algebra-1,heine-borel}^'n^'n*
assumes *Ahyp*: *matrix-continuous-on T A*

and $\bigwedge\tau\ \varepsilon.\ \tau \in T \Longrightarrow \varepsilon > 0 \Longrightarrow$ *bdd-above* $\{\|A\ t\|_{op}\ |t.\ dist\ \tau\ t \leq \varepsilon\}$
and *Bhyp*: *continuous-on T B* **and** *open S*
and $t_0 \in T$ **and** *Thyp*: *open T is-interval T*
shows *picard-lindeloef* $(\lambda\ t\ s.\ A\ t *v s + B\ t)\ T\ S\ t_0$
⟨*proof*⟩

With this result, we obtain a generic instance of *picard-lindeloef.unique-solution* of Section 4.2 for every affine system. This is useful for verification as exemplified in Section 7.1. Assumptions *Ahyp* and *Bhyp*, above, state that functions A and B are continuous. The second one requires that the image of $\overline{B_\tau(\varepsilon)}$ for $\tau \in T$ under $\lambda t.\ \|A\,t\|_{op}$ is bounded above. The remaining assumptions are direct conditions of Picard-Lindelöf's theorem.

Continuity in *Ahyp* is different from that in *Bhyp* because Isabelle's default norm for matrices as vector of vectors A :: *real^'n^'m* is the Euclidean norm, not the operator norm. This issue reverberates also in the general solution for time-independent affine system as, in Isabelle, the exponential operation exp $x = \sum_{n\in\mathbb{N}} \frac{1}{n!}x^n$ is available only within the type-class *real-normed-algebra-1*. Among its axioms, one requires that the identity element 1 must satisfy $\|1\| = 1$. Yet, this is not true for *real^'n^'n*, because $\|(\chi\ i.\ 1)\| \neq 1$. Therefore, we define a sub-type of square matrices, set the operator norm as its default norm, and show that this new type is an instance of *real-normed-algebra-1* and *banach*.

typedef *'m sq-mtx* = *UNIV*::(*real^'m^'m*) *set*
morphisms *to-vec sq-mtx-chi* **by** *simp*

instance *sq-mtx* :: (*finite*) *real-normed-algebra-1*
⟨*proof*⟩

instance *sq-mtx* :: (*finite*) *banach*
⟨*proof*⟩

The command *morphisms* introduces the bijection *to-vec* and its inverse *to-mtx* between *'n sq-mtx* and *real^'n^'n*. Both instantiations require proving that matrices form normed vector spaces. Beyond that, the first instantiation requires showing that they also form a ring. The second instantiation formalises our notion of complete metric space of Section 4.1. That is, it shows that every Cauchy sequence of square matrices converges.

Introducing this new type in Isabelle/HOL means that we must lift previous properties and operations for *real^'n^'n* to *'n sq-mtx*. The code below, shows some examples of this.

lift-definition *sq-mtx-ith* :: *'m sq-mtx* ⇒ *'m* ⇒ (*real^'m*) (**infixl** $$ *90*) **is** ($) .

lift-definition *sq-mtx-vec-mult* :: *'m sq-mtx* ⇒ (*real^'m*) ⇒ (*real^'m*) (**infixl** $*_V$ *90*) **is** $(*v)$.

lift-definition *sq-mtx-inv* :: (*'m*::*finite*) *sq-mtx* ⇒ *'m sq-mtx* ($^{-1}$ [*90*]) **is** *matrix-inv* .

This means that we can write $$ and $*_V$ instead of $ and $*v$ respectively, and we can convert proofs between the new and the old type. We thus obtain the same results as before in the new type, including Picard-Lindelöf's theorem. Notice that this time we do

not need different versions of continuity. Similarly, the other assumptions have absorbed the requirement for boundedness above.

lemma *picard-lindeloef-sq-mtx-affine*:
assumes *continuous-on* $T\ A$ **and** *continuous-on* $T\ B$
and $t_0 \in T$ **and** *is-interval* T **and** *open* T **and** *open* S
shows *picard-lindeloef* $(\lambda t\ s.\ A\ t *_V s + B\ t)\ T\ S\ t_0$
⟨*proof*⟩

The next step is to formalise the general solution for autonomous affine systems and linear systems of ODEs. We formalise the affine version below, while that for linear systems specialises to $X\,t = (\exp\,((t - t_0)\,A)) \cdot s$ for $s \in \mathbb{R}^n$.

lemma *has-vderiv-on-sq-mtx-affine*:
fixes t_0::*real* **and** $A :: ('a::finite)\ sq\text{-}mtx$
defines $lSol\ c\ t \equiv exp\ ((c * (t - t_0)) *_R A)$
shows $D\ (\lambda t.\ lSol\ 1\ t *_V s + lSol\ 1\ t *_V (\int_{t0}^{t}\ (lSol\ (-1)\ \tau *_V B)\ \partial\tau)) =$
$(\lambda t.\ A *_V (lSol\ 1\ t *_V s + lSol\ 1\ t *_V (\int_{t0}^{t}\ (lSol\ (-1)\ \tau *_V B)\ \partial\tau)) + B)\ on\ \{t_0 -\!- t\}$
⟨*proof*⟩

As no conditions on the parameter t are given, these general solutions are proper flows in the sense that they are defined over the entire monoid $\mathbb{R}$ and state space $\mathbb{R}^n$. We formalise these results with the locale *local-flow* of Section 4.2.

lemma *local-flow-sq-mtx-affine*: *local-flow* $(\lambda s.\ A *_V s + B)\ UNIV\ UNIV$
$(\lambda t\ s.\ exp\ (t *_R A) *_V s + exp\ (t *_R A) *_V (\int_0^t (exp\ (-\tau *_R A) *_V B)\partial\tau))$
⟨*proof*⟩

lemma *local-flow-sq-mtx-linear*:
local-flow $((*_V)\ A)\ UNIV\ UNIV\ (\lambda t\ s.\ exp\ (t *_R A) *_V s)$
⟨*proof*⟩

Determining such exponentials may be computationally expensive due to the iteration of matrix multiplication. Exceptions are *diagonalisable matrices* A which are similar to a diagonal matrix D in the sense that there is an invertible P such that $A = P^{-1}DP$. For these matrices,

$$\exp\,A = \exp\,(P^{-1}DP) = P^{-1}(\exp\,D)P,$$

where $\exp D$ in the right hand side is diagonal and easy to characterise: its entries in the main diagonal are the exponential of those in D. Therefore, when working with solutions of autonomous affine (or linear) systems, it is preferable to work with those in diagonal form. As reasoning with general solutions is easier for diagonalisable matrices, we formalise matrix invertibility, similarity and diagonal matrices from linear algebra. We also characterise the exponential of a matrix in terms of these concepts.

lemma *mtx-invertible-def*: *mtx-invertible* $A \longleftrightarrow (\exists A'.\ A' * A = 1 \wedge A * A' = 1)$
⟨*proof*⟩

definition *similar-sq-mtx* :: (*'n*::*finite*) *sq-mtx* ⇒ *'n sq-mtx* ⇒ *bool* (**infixr** ∼ *25*)
where $(A \sim B) \longleftrightarrow (\exists\ P.\ mtx\text{-}invertible\ P \wedge A = P^{-1} * B * P)$

definition *diag-mat f* = $(\chi\ i\ j.\ if\ i = j\ then\ f\ i\ else\ 0)$

lemma *exp-scaleR-diagonal1*:
assumes *mtx-invertible P* **and** $A = P^{-1} * (\text{diag}\ i.\ f\ i) * P$
shows $exp\ (t *_R A) = P^{-1} * (\text{diag}\ i.\ exp\ (t * f\ i)) * P$
⟨*proof*⟩

The first three concepts and related properties are available for matrices of type *real^'n^'n* and *'n sq-mtx*. The exponential is only available for the latter. For example, the notation (diag *i*. *f i*) is the *'n sq-mtx* version of *diag-mat f*.

All the lemmas and abbreviations displayed above are part of our entry in Isabelle's Archive of Formal Proofs [70]. Our work covers over 10 pages of proofs and definitions about matrix limits, norms, and operations. This is equivalent to more than 600 lines of code. Our development of the type *'m sq-mtx* and the diagonalisation of square matrices is over 16 pages long. It spans over 900 lines of code or more than 200 lemmas whose proofs are long due to the various convergence arguments and instantiations. These substantial formalisations allow Isabelle users to prove facts involving derivatives of matrix operations. We exemplify their use in verification in Sections 7.2 and 7.3.

6.6 Summary of the Verification Components

In this section, we do a brief recapitulation of the current state of the verification components formalised so far. We mention the benefits of using one implementation over another and we also describe the theoretical use cases for our developments.

Our verification components based on KATs prove the least amount of properties about hybrid systems as they formalise the minimal logic $\mathsf{d}\mathcal{H}$. They do not include diamond operators and do not compute weakest preconditions. Yet, this is the only version that includes a refinement calculus via rKAT and it is also the only version with a variant that includes lenses and the notational benefits of Isabelle/UTP. The reason for this is only due to the chronological development of the components. That is, the other components discussed in the sequel can also include lenses and refinement but this is left for future work. Overall, we recommend the KAT-based components only for simple verifications or, temporarily, for refinement of hybrid programs.

The MKA version of the verification components with its relational model is our most developed variant. It not only subsumes KATs differential Hoare logic, but extending it is easy due to its high modularity and Isabelle's support for proofs about relations. Adding new results, for instance about MKAs forward diamonds, is straightforward. The alternative state transformer version occasionally is harder to use due to lack of support and frequent use of translations between the type of nondeterministic functions *'a nd-fun* and the function type *'a* ⇒ *'a set*. Yet, this only applies to its development as for verifications and case studies, its use is indistinguishable from the relational version. A possible issue with both MKA variants

is that care must be present when introducing a new concrete semantics for hybrid programs. Class instantiations quickly become non-compositional because Isabelle users can only do this process once. For instance, combining HOL-Analysis and the AFP entry [56] generates such a crash for the relational model of MKAs.

The components based on predicate transformers from quantales or the powerset monad require the most preliminary work for their implementation. However, their verification proofs tend to be faster. This is because, for Isabelle, their underlying types of sets and functions are easier to handle than relations or state transformers. Their dependencies, however, affect common notation like set-elementhood replacing $s \in S$ with *s2p S s* which may make them harder to learn for users that plan to build upon them.

In contrast, the lightweight version based on predicate transformers includes all these automation benefits without any issues about dependencies on previous theories. Not only that, but loading it is a lot faster than for the rest of the components. It is the ideal choice for fast developments and tests.

At the level of verification condition generation, however, non of these implementations is comparatively better than the other. That is, the formalisation of the soundness rules in each of them is similar in length and they implement very similar proof-styles. Furthermore, all of them include the three variants of evolution commands, the traditional one $(x' = f \,\&\, G)_U$, the one based on guarded orbitals ($x' = f \,\&\, G$ on $U\, S$ at t_0), and the one that directly encodes the dynamics $\mathbf{evol}\,\varphi\, G\, U$. Verifying the second of these three involves either following the procedure of Section 4.4 that supplies and certifies solutions and Lipschitz continuity, or providing a differential invariant as in the procedure of Section 5.2. The evolution command $(x' = f \,\&\, G)_U$ is a special case of the orbital based one and it requires the same procedures. We merely have it for resemblance to differential dynamic logic and the derivation of its rules of inference. Thus, we can do some verifications in the style of $\mathsf{d}\mathcal{L}$ with it. On the other hand, dynamics based evolution commands $\mathbf{evol}\,\varphi\, G\, U$ are more optimistic in the sense that verifications with them do not require certifications that φ is correct. Yet, in many applications the function φ is not known. The other two are alternatives to verify those cases.

In the case of our formalisation of affine systems, users have several methods for proving properties about them. The choice of the method depends on whether users know a solution to the system

$$X'\, t = A\, t \cdot X\, t + B\, t, \tag{6.1}$$

and, in the autonomous case $A\, t = A$, it also depends on whether A is diagonalisable.

For instance, certifying that a function X solves system (6.1) is a matter of just stating the corresponding Isabelle formalisation and using our tactic *poly-derivatives* to check that both sides of the equation reduce to the same expression as in Section 4.2.

In Section 7.1, we see that sometimes a characterisation X_1 of the solution to system (6.1) is simpler than other X_2. In those cases, the uniqueness lemmas as provided by *picard-lindeloef* locale instantiated to our lemma *picard-lindeloef-sq-mtx-affine* formalise the fact that $X_1\, t = X_2\, t$. This allows us to conveniently switch between characterisations.

In cases where users do not have a solution to system (6.1), but this is time-independent, they can use our formalisation of the general solution for autonomous affine systems $X'\, t = A \cdot X\, t + B$ in terms of the matrix exponential with lemmas *has-vderiv-on-sq-mtx-affine*

and *has-vderiv-on-sq-mtx-linear*. We also provide theorems for simplifications of the general solution in case that A is diagonalisable $A = P^{-1}DP$. An external tool, like a computer algebra system, can provide the diagonal matrix D and change of basis matrix P. Then it is just a a matter of using our lemma *exp-scaleR-diagonal1* of Section 6.5 to simplify the general solution. Hence, in the particular case of the autonomous version of system (6.1) with A diagonalisable, the procedure of Section 4.4 has already been done on the background with our formalisation. Not only that, but we have already proved that these systems satisfy the *local-flow* locale. Thus, the monoid action equations are available for them.

Yet, there is still much to add for a full treatment of linear systems in Isabelle/HOL. For non-diagonalisable matrices, the more general approach with Jordan Normal forms [131] is missing. However, many non-diagonalisable matrices can still be tackled with our formalisation. See Sections 7.1 and 7.2 for examples of this. Another result missing is a general solution for system (6.1), which is non-autonomous. The approaches to obtaining it have to solve an associated linear system where $B\,t = 0$ [49, 63]. These include the variation of parameters method, the resolvent matrix method or a change in the dimension of system (6.1) by solving instead

$$\begin{pmatrix} X'\,t \\ 1' \end{pmatrix} = \begin{pmatrix} (A\,t) & (B\,t) \\ 0^\top & 0 \end{pmatrix} \cdot \begin{pmatrix} X\,t \\ 1 \end{pmatrix} = \begin{pmatrix} A\,t \cdot X\,t + B\,t \\ 0 \end{pmatrix}$$

which subsumes it in its first entry. Here, $0^\top$ is a transposed zero-vector with the same length as $B\,t$ and $0, 1 \in \mathbb{R}$. Nevertheless, our formalisation is a good basis for implementing these extensions.

In summary, for simple verification tasks users may use KAT and rKAT. This is the recommended component if nice syntax is a requirement as there is a variant with lenses and Isabelle/UTP. For fast development and extensions independent of algebraic preliminaries, the lightweight stripped down predicate transformer components are the better choice. At the level of verification condition generation however, the MKA or any predicate transformer version are interchangeable. The formalisation of affine systems is compositional with any of these approaches. Yet, it improves not only the verification components but also includes lemmas to prove properties about specific instances of system (6.1). The complete Isabelle formalisation, including examples, covers about 170 pages of proofs. Previous versions can be found in the Archive of Formal Proofs [68, 70] while the most recent developments for this thesis are available in the online repository https://github.com/yonoteam/CPSVerification.

Chapter 7

Formal Verifications

Until now, the running example of the `thermostat` hybrid program of Figure 2.5 is the only application where we use our verification components. In order to illustrate more diverse usages, we exhibit a wider variety of verification problems of increasing difficulty. We use each one of them to highlight particular aspects of our components. Despite their early stage of development, our components display various features that we can contrast and compare to existing tools for deductive verification. To do this, throughout the chapter we discuss the following concrete categories when doing these comparisons: the variety of hybrid programs available in the tool, the language or formulas the tool can use to write specifications, its diversity of methods for verifying evolution commands, the length of the proofs, and the amount of proof-automation the tool supports. We also limit our comparisons to $\mathsf{d}\mathcal{L}$'s flagship theorem prover KeYmaera X and to the HHL prover because their verification style is closer to that of our components.

In Section 7.1, we prove an invariance property for a single evolution command with our procedures of Sections 4.4 and 5.2 for supplying solutions to systems of ODEs and providing invariants respectively. We also formalise these results and show alternative verifications with our components for affine systems and the dynamics-based evolution commands. In Section 7.2, we use again the properties of linear systems to verify a sequential composition of an evolution command and an assignment. A final usage of our linear systems formalisation with a diagonalisable matrix is in Section 7.3. Afterwards, we verify the classical example of a bouncing ball in Section 7.4 that helps us test the integration of evolution commands with loops and conditional statements. For this reason, we verify it four times: using invariants, supplying flows, directly annotating the dynamics, and via $\mathsf{d}\mathcal{H}$ and the refinement components with lenses and the Isabelle/UTP framework. After that, we showcase an alternative formalisation of doing refinement proofs in Isabelle/HOL in Section 7.5. Then, we briefly report on our participation in the ARCH2020 friendly competition [97] in Section 7.6. Throughout the section, we verify different benchmark problems of the competition that exhibit the main limitations of our verification components: their proof-automation and better support for invariant reasoning. At the same time, these problems allow us to discuss the versatility of our framework to provide alternative solutions to them. In Section 7.7, we lay the first steps towards a verification of a more complicated system: the control of a quadcopter that uses the standard PID technique [34]. We conclude the chapter with Section 7.8 where we summarise the features and limitations of the components discussed so

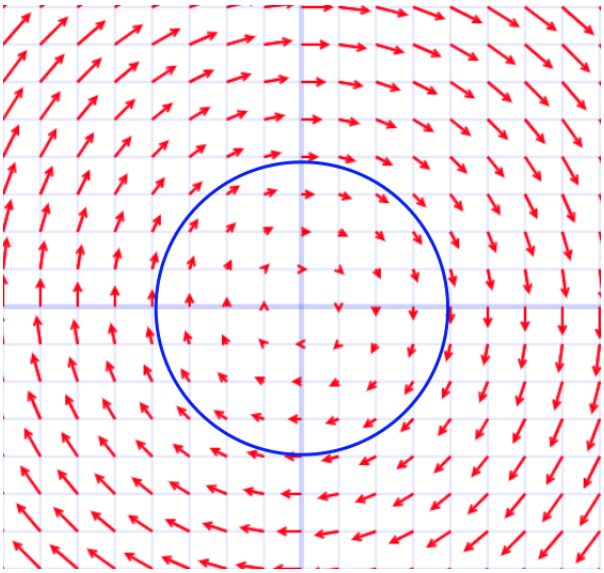


Figure 7.1: Circular motion vector field

far in comparison with other deductive verification tools.

7.1 Circular Motion

Our first example is a simple proof of invariance for an evolution command. Because of its simplicity, it is ideal for showcasing the various verification procedures supported by our components. It also helps us to highlight the difference between them. Its continuous dynamics are described with the differential equations

$$x'\,t = y\,t \qquad \text{and} \qquad y'\,t = -x\,t,$$

that, as a vector field $f : \mathbb{R}^{\{x,y\}} \to \mathbb{R}^{\{x,y\}}$, correspond to the linear system

$$\begin{pmatrix} f\,s\,x \\ f\,s\,y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} s\,x \\ s\,y \end{pmatrix} = \begin{pmatrix} s\,y \\ -s\,x \end{pmatrix}.$$

Figure 7.1 shows a graphical representation of this vector field. It already illustrates that if a particle starts in the circumference $I\,s \leftrightarrow ((s\,x)^2 + (s\,y)^2 = r^2)$, it remains there. Alternatively, to obtain this invariant we can use parametric derivation. That is, abusing notation, we can consider the variation of y relative to x via

$$\frac{dy}{dx} = \frac{y'}{x'} = -\frac{x}{y}.$$

Solving this ODE with respect to x yields the desired $x^2 + y^2 = r^2$ with $r \geq 0$.

Therefore, throughout this section, we are interested in proving

$$\{\lambda s.\ (s\,x)^2 + (s\,y)^2 = r^2\}\ (x' = f\,\&\,G)_U\ \{\lambda s.\ (s\,x)^2 + (s\,y)^2 = r^2\}, \tag{7.1}$$

where $G : \mathbb{R}^{\{x,y\}} \to \mathbb{B}$ and $U : \mathbb{R}^{\{x,y\}} \to \mathcal{P}\,\mathbb{R}$ maps states to intervals. In the next couple of examples, we prove this using either invariance or the system of ODEs' flow. We conclude the section with alternative verifications using properties of affine systems and direct annotations of the dynamics.

Example 7.1.1 (Circular Motion via Invariant). Proceeding as in Example 5.2.1, we follow the procedure from Section 5.2 to show that $I\,s \leftrightarrow ((s\,x)^2 + (s\,y)^2 = r^2)$ is a differential invariant for f and G along U.

As before, steps 1(a) and 1(b) do not apply as I is a single positive clause of a conjunctive formula. Next, we use Lemma 5.2.5.1 to test if an equality is an invariant. For that we need to show

$$\big((X\,t\,x)^2 + (X\,t\,y)^2\big)' = (r^2)',$$

for a function X such that $X'\,t\,i = f\,(X\,t)\,i$ for $i \in \{x, y\}$. The result is calculational.

$$\begin{aligned}\big((X\,t\,x)^2 + (X\,t\,y)^2\big)' &= 2(X\,t\,x)(X'\,t\,x) + 2(X\,t\,y)(X'\,t\,y)\\ &= 2(X\,t\,x)(f\,(X\,t)\,x) + 2(X\,t\,y)(f\,(X\,t)\;y)\\ &= 2(X\,t\,x)(X\,t\,y) - 2(X\,t\,y)(X\,t\,x) = 0\end{aligned}$$

Therefore, by Proposition 5.2.1, we have just shown that

$$\{\lambda s.\ (s\,x)^2 + (s\,y)^2 = r^2\}\,(x' = f\,\&\,G)_U\,\{\lambda s.\ (s\,x)^2 + (s\,y)^2 = r^2\}.$$

In Isabelle/HOL, the corresponding proof is automatic using our tactics for derivatives and differential invariants.

```
abbreviation fpend :: real^2 ⇒ real^2 (f)
  where f s ≡ (χ i. if i = 1 then s$2 else −s$1)

lemma (λs. r² = (s$1)² + (s$2)²) ≤ |x´= f & G] (λs. r² = (s$1)² + (s$2)²)
  by (auto intro!: diff-invariant-rules poly-derivatives)
```

Here, we use the number 1 for x and 2 for variable y. □

Example 7.1.2 (Circular Motion via Flow). Another way to prove the Hoare triple 7.1 is by providing the solution to the differential equations. It corresponds to a rotation on the initial state.

$$\begin{pmatrix}\varphi\,t\,s\,x\\ \varphi\,t\,s\,y\end{pmatrix} = \begin{pmatrix}\cos t & \sin t\\ -\sin t & \cos t\end{pmatrix}\cdot\begin{pmatrix}s\,x\\ s\,y\end{pmatrix} = \begin{pmatrix}(s\,x)\cos\,t + (s\,y)\sin\,t\\ -(s\,x)\sin\,t + (s\,y)\cos\,t\end{pmatrix}.$$

Due to the fact that the system is linear, we know that it satisfies Lipschitz continuity. The following equalities check that $\lambda t.\ \varphi\,t\,s$ solves the system

$$\begin{pmatrix}\varphi'\,t\,s\,x\\ \varphi'\,t\,s\,y\end{pmatrix} = \begin{pmatrix}-\sin t & \cos t\\ -\cos t & -\sin t\end{pmatrix}\cdot\begin{pmatrix}s\,x\\ s\,y\end{pmatrix} = \begin{pmatrix}0 & 1\\ -1 & 0\end{pmatrix}\begin{pmatrix}\cos t & \sin t\\ -\sin t & \cos t\end{pmatrix}\cdot\begin{pmatrix}s\,x\\ s\,y\end{pmatrix} = \begin{pmatrix}f\,(\varphi\,t\,s)\,x\\ f\,(\varphi\,t\,s)\,y\end{pmatrix}.$$

Furthermore, $\varphi\,0\,s = \Delta\cdot s = s$ where Δ is the identity matrix. Therefore, we can safely apply either (h-evol) or (wlp-evol). To do this we observe that

$$\begin{aligned}I\,(\varphi\,t\,s) &\leftrightarrow (\varphi\,t\,s\,x)^2 + (\varphi\,t\,s\,y)^2 = r^2\\ &\leftrightarrow ((s\,x)\cos\,t + (s\,y)\sin\,t)^2 + (-(s\,x)\sin\,t + (s\,y)\cos\,t)^2 = r^2\\ &\leftrightarrow (s^2\,x + s^2\,y)(\cos^2\,t + \sin^2\,t) = r^2 \leftrightarrow I\,s\end{aligned}$$

The *wlp*-rule for evolution commands therefore yields

$$I \leq |(x' = f \,\&\, G)_U]\, I = (\lambda s.\ I\, s \to (\forall t \in U.\ (\forall \tau \in \downarrow U\, t.\ G\,(\varphi\, \tau\, s)) \to I\, s))) = \top.$$

We split the Isabelle formalisation corresponding to the process above in three stages. First, we introduce the flow. Then, we show that the vector field satisfies the assumptions of our *local-flow* locale which involves a proof for Lipschitz continuity. Finally, we use this result to compute the *wlp* and prove the correctness specification.

```
abbreviation circ-flow :: real ⇒ real^2 ⇒ real^2 (φ)
  where φ t s ≡ (χ i. if i = 1 then s$1 * cos t + s$2 * sin t else − s$1 * sin t + s$2 * cos t)

lemma local-flow-circ: local-flow f UNIV UNIV φ
  apply(unfold-locales, simp-all add: local-lipschitz-def lipschitz-on-def vec-eq-iff, clarsimp)
    apply(rule-tac x=1 in exI, clarsimp, rule-tac x=1 in exI)
    apply(simp add: dist-norm norm-vec-def L2-set-def power2-commute UNIV-2)
  by (auto simp: forall-2 intro!: poly-derivatives)

lemma (λs. r^2 = (s$1)^2 + (s$2)^2) ≤ |x´=f & G] (λs. r^2 = (s$1)^2 + (s$2)^2)
  by (force simp: local-flow.fbox-g-ode-subset[OF local-flow-circ])
```

The longest proof corresponds to lemma *local-flow-circ*. The first line of its proof unfolds definitions. The second lines supplies a radius of existence of the solutions and the Lipschitz constant (both equal to 1). The third line discharges the obligation about Lipschitz continuity. The final line uses our tactic *poly-derivatives* to check that the derivatives coincide. Thus, the proof of the correctness specification is a simple application of the corresponding *wlp*-rule based on the fact that *local-flow-circ* holds. □

For completeness of this section, we present two alternative verifications of the Hoare triple 7.1. The first one uses our components for linear systems. First we introduce the matrix of the vector field using our function *mtx* that turns lists into square matrices.

```
abbreviation mtx-circ :: 2 sq-mtx (A)
  where A ≡ mtx
   ([0,  1] #
    [−1, 0] # [])
```

From this we know that the solution to the system of ODEs is $\lambda t\, s.\ \exp{(tA)} \cdot s$. However, the diagonlisation of this matrix involves complex numbers and expanding the exponential requires handling infinite sums. As described in Section 6.6, an alternative method involves using the uniqueness lemmas to replace this solution with a simpler one. Below we show this process for the function φ of the previous Example 7.1.2.

```
lemma mtx-circ-flow-eq: exp (t *R A) *V s = φ t s
  apply(rule local-flow.eq-solution[OF local-flow-sq-mtx-linear, symtrc, of - λs. UNIV], simp-all)
    apply(rule ivp-solsI, simp-all add: sq-mtx-vec-mult-eq vec-eq-iff)
  unfolding UNIV-2 using exhaust-2
  by (force intro!: poly-derivatives simp: matrix-vector-mult-def)+
```

In the proof above, the first line calls the uniqueness lemma of locale *local-flow* which holds because we have shown in our formalisation of linear systems that they satisfy the assumptions of this locale. The remaining lines just prove that $\lambda t.\ \varphi\, t\, s \in \mathsf{Sols}(\lambda s.\ A \cdot s)\,(\lambda s.\ \mathbb{R})\,\mathbb{R}^{\{x,y\}}\,0\,s$ by unfolding definitions and using our derivative tactics. Thus, we can use this result to show the correctness specification.

lemma $(\lambda s.\ r^2 = (s\$1)^2 + (s\$2)^2) \leq |x' = (*_V)\ A\ \&\ G]\ (\lambda s.\ r^2 = (s\$1)^2 + (s\$2)^2)$
  apply(*subst local-flow.fbox-g-ode-subset*[*OF local-flow-sq-mtx-linear*])
  unfolding *mtx-circ-flow-eq* **by** *auto*

The first line applies the *wlp* rule while the second uses the equality given by uniqueness to complete the proof.

However, the simplest formalisation of the Hoare triple 7.1, corresponds to a direct encoding of the function φ in the specification.

lemma $(\lambda s.\ r^2 = (s\$1)^2 + (s\$2)^2) \leq |EVOL\ \varphi\ G\ T]\ (\lambda s.\ r^2 = (s\$1)^2 + (s\$2)^2)$
  by *force*

This is expected because the proof involves no checks for invariance or derivatives.

In comparison with other tools for deductive verification of hybrid systems like KeYmaera X or the HHL prover, our first formalisation using invariants would also be possible in both. In fact, both provers have increased automation that allows them to handle complex invariants more easily than we can. However, the circular motion is so simple, that our basic tactics can discharge its proof obligations easily too.

Contrastingly, our proof supplying the flow is bigger than our proof with invariants because we also need to certify Lipschitz continuity and that the flow is indeed a solution to the system of ODEs. Yet, it also serves to evidence how smoothly we can handle transcendental functions like sines, cosines and exponentials in Isabelle/HOL. For reasons of decidability, these transcendental functions are not supported in KeYmaera X. Nevertheless, KeYmaera X can supply a solution to this circular motion problem by extending the dimension of the state space to reason about sines and cosines indirectly. The HHL prover can also provide the flow of this system as it is also implemented in Isabelle/HOL. However, neither KeYmaera X nor the HHL prover provide syntax for matrix reasoning as we do with our formalisation of linear and affine systems of ODEs. Similarly, they also do not include a hybrid program to reason about the dynamics directly in the specification.

7.2 Docking Station

In this section we consider a hybrid program involving only an assignment and an evolution command. Hence, the example is still simple but it allows us to focus on our formalisation of affine and linear systems. In particular, the dynamics involve a non-diagonalisable matrix.

The verification problem involves a spaceship moving at a constant speed of $v_0 > 0$ and aligned with its docking station at a distance d of the ship which currently is at x_0. To stop exactly at d, the ship determines that it needs a constant negative acceleration of

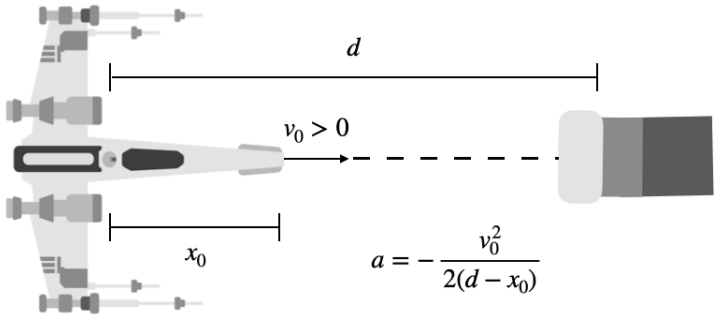


Figure 7.2: A spaceship aligned with its station about to start its docking process.

$a = -\frac{v_0^2}{2(d-x_0)}$. Figure 7.2 illustrates this scenario. The corresponding partial correctness specification is therefore

$$\{\lambda s.\ s\,x = x_0 \wedge s\,v = v_0\}\ \left(a := \lambda s.\ -\frac{v_0^2}{2(d-x_0)}\ ;(x' = f_0\,\&\,G)_{\mathbb{R}_+}\right)\ \{\lambda s.\ s\,v = 0 \leftrightarrow s\,x = d\},$$

where G is any predicate. The dynamics of the system coincide with those of Example 5.2.1, that is $x'''\,t = 0$. Yet, we can write the corresponding vector field f_0 with a matrix

$$\begin{pmatrix} f_0\,s\,x \\ f_0\,s\,v \\ f_0\,s\,a \end{pmatrix} = K \cdot \begin{pmatrix} s\,x \\ s\,y \\ s\,a \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} s\,x \\ s\,y \\ s\,a \end{pmatrix},$$

for variables $V = \{x, v, a\}$ that respectively represent the ship's position, velocity and acceleration. Due to the fact that this is a linear system, it has unique solutions. Moreover, because the system is autonomous, its solutions are equal to $\varphi\,t\,s = \exp(tK) \cdot s$.

Computation of the exponential operator on tK is still manageable because

$$tK = \begin{pmatrix} 0 & t & 0 \\ 0 & 0 & t \\ 0 & 0 & 0 \end{pmatrix}, \qquad (tK)^2 = \begin{pmatrix} 0 & 0 & t^2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad (tK)^3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Therefore, unfolding definitions we get that

$$\exp(tK) = \left(\sum_{n\in\mathbb{N}} \frac{1}{n!}(tK)^n\right) = \left(1 + tK + \frac{(tK)^2}{2!}\right) = \begin{pmatrix} 0 & t & \frac{t^2}{2} \\ 0 & 0 & t \\ 0 & 0 & 0 \end{pmatrix}.$$

Below, the formalisation of these results is straightforward. Notice that the proofs are relatively simple because, in our work with affine systems of ODEs, we have added simplification rules for matrix operations. This produces fast certifications involving them.

lemma *exp-mtx-cnst-acc*: *exp* $(t *_R K) = ((t *_R K)^2/_R\ 2) + (t *_R K) + 1$
  unfolding *exp-def* **apply**(*subst suminf-eq-sum*[*of 2*])
  using *powN-scaleR-mtx-cnst-acc* **by** (*simp-all add*: *numeral-2-eq-2*)

lemma *exp-mtx-cnst-acc-vec-mult-eq*: *exp* $(t *_R K) *_V s =$
vector $[s\$3 * t\hat{}2/2 + s\$2 * t + s\$1, s\$3 * t + s\$2, s\$3]$
apply(*subst exp-mtx-cnst-acc*, *subst pow2-scaleR-mtx-cnst-acc*)
apply(*simp add*: *sq-vec-mult-eq vector-def*)
unfolding *UNIV-3* **by** (*simp add*: *fun-eq-iff*)

In the formalisation above, we use states $s = (s\$1, s\$2, s\$3)$ where $s\$1$ is the ship's position, $s\$2$ is its velocity and $s\$3$ is its acceleration. Therefore, the second lemma states that the exponential of tK applied to a state s generates the traditional kinematics equations

$$\begin{pmatrix} (\exp(tK)\cdot s)\,x \\ (\exp(tK)\cdot s)\,v \\ (\exp(tK)\cdot s)\,a \end{pmatrix} = \begin{pmatrix} (s\,a)\frac{t^2}{2} + (s\,v)t + s\,x \\ (s\,a)t + s\,v \\ s\,a \end{pmatrix},$$

where *vector* is a function that turns Isabelle's lists into vectors. We can use these results to verify the desired correctness specification.

lemma *docking-station-arith*:
assumes $(d::real) > x$ **and** $v > 0$
shows $(v = v^2 * t \;/\; (2 * d - 2 * x)) \longleftrightarrow (v * t - v^2 * t^2 \;/\; (4 * d - 4 * x) + x = d)$
⟨*proof*⟩

lemma *docking-station*:
assumes $d > x_0$ **and** $v_0 > 0$
shows *PRE* $(\lambda s.\ s\$1 = x_0 \wedge s\$2 = v_0)$
HP $((3 ::= (\lambda s.\ -(v_0\hat{}2/(2*(d-x_0))))); x´=(*_V)\ K\ \&\ G)$
POST $(\lambda s.\ s\$2 = 0 \longleftrightarrow s\$1 = d)$
apply(*clarsimp simp*: *le-fun-def local-flow.fbox-g-ode*[*OF local-flow-sq-mtx-linear*[*of K*]])
unfolding *exp-mtx-cnst-acc-vec-mult-eq* **using** *assms* **by** (*simp add*: *docking-station-arith*)

The proof of the specification is simple because it calls our lemma stating that affine systems satisfy the assumptions of the *local-flow* locale. It also calls lemma *docking-station-arith* that restates and shows the last proof obligation. Supplying it to Isabelle's simplifier allows the proof assistant to finish the verification automatically.

As explained before, we have proved a generic theorem that provides the requirements for Lipschitz continuity and certification of the flow with our formalisation of affine systems. This simplifies those parts of the procedure of Section 4.4. Instead, users might have to convert the provided solution in terms of the matrix exponential into simpler expressions as we have done here.

Among other things, the example shows that, in our setting, sequential composition, assignments and ODEs work smoothly together. They get transformed into proof obligations about real numbers automatically with Isabelle's simplifier. This is also easily done with KeYmaera X and the HHL prover. Nevertheless, the main feature displayed in the example is the support of our components for matrix reasoning which is not available in other deductive verification provers. This is not to say that those other tools cannot verify the problem presented above. In fact, both KeYmaera X and the HHL prover could supply an alternative

representation of the solution to the ODE or reason about it with invariants. Yet, the availability of including affine systems opens the possibility for future research that improves the technique as they are used frequently in control engineering [133].

7.3 Overdamped Door

In this section, we present a verification using our formalisation of affine systems of ODEs that uses a diagonalisation. We focus on a common second order ODE in physics and engineering:

$$x'' t = a(x t) + b(x' t).$$

For instance, by setting $a = \frac{1}{CL}$ and $b = \frac{R}{L}$, the ODE models the current in a closed circuit of a resistor (R) in series with an inductor (L), a capacitor (C) and a source of constant voltage [63]. However, we are more interested in its representation as a damped harmonic oscillator by fixing $a = -\frac{k}{m}$ and $b = -\frac{d}{m}$. Here, m represents a mass attached to a spring of constant k and sliding horizontally with a damping factor d. In particular, our system of interest is the overdamped mechanism of a door that prevents if from slamming against its frame or from opening on the other side. Hence, we must require that $b^2 + 4 \cdot a < 0$.

To prove that the assumption $b^2 + 4 \cdot a < 0$ guarantees that the door behaves adequately, we use variable x to indicate the horizontal extension of the spring and, therefore, how much the door is opened. Thus, $s\,x = 0$ represents the spring at its natural state, hence, a closed door. Accordingly, $s\,x > 0$ is an open door and $s\,x < 0$ is the unsafe region. Similarly, $s\,v$ represents the velocity of the horizontal displacement.

To model a person opening the door, we use $\texttt{open\_door}\, s = \{s \mid s\,x > 0 \wedge s\,v = 0\}$. This state transformer covers all the possible cases where the door is open and the person has just stopped pushing or pulling it. After this happens, the evolution command follows. Yet, we represent the second order ODE as a linear system:

$$\begin{pmatrix} f\,s\,x \\ f\,s\,v \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ a & b \end{pmatrix} \cdot \begin{pmatrix} s\,x \\ s\,v \end{pmatrix},$$

or more briefly $f\,s = (A\,a\,b) \cdot s$ for $s \in \mathbb{R}^{\{x,v\}}$. Thus, the hybrid program for our system is

$$\textbf{loop}\ \left(\texttt{open\_door}\,;\, (x' = \lambda s.\ (A\,a\,b) \cdot s\ \&\ G)_{\mathbb{R}_+}\right).$$

We can assume that the loop models a child constantly reopening the door. Moreover, the guard G could be any predicate including the guard that does nothing $\top$.

As the system is linear, we know that the flow is $\varphi\,t\,s = \exp\left(t(A\,a\,b)\right) \cdot s$. To simplify it, we need the eigenvalues ι_1 and ι_2, and the change of basis matrix P that diagonalise $A\,a\,b$. That is, $A\,a\,b = P^{-1}DP$ where D is the diagonal matrix with ι_1 and ι_2 constituting it. Instead of doing the diagonalisation in Isabelle, we can use a computer algebra system to supply it:

$$P = \begin{pmatrix} -\frac{\iota_2}{a} & -\frac{\iota_1}{a} \\ 1 & 1 \end{pmatrix}, \quad \iota_1 = \frac{b - \sqrt{b^2 + 4a}}{2} \quad \text{and} \quad \iota_2 = \frac{b + \sqrt{b^2 + 4a}}{2}.$$

In Isabelle/HOL, we certify the diagonlisation as shown below.

lemma *mtx-hOsc-diagonalizable*:
defines $\iota_1 \equiv (b - sqrt\ (b\hat{}2+4*a))/2$ **and** $\iota_2 \equiv (b + sqrt\ (b\hat{}2+4*a))/2$
assumes $b^2 + a * 4 > 0$ **and** $a \neq 0$
shows $A\ a\ b = P\ (-\iota_2/a)\ (-\iota_1/a) * (\text{diag}\ i.\ if\ i = 1\ then\ \iota_1\ else\ \iota_2) * (P\ (-\iota_2/a)\ (-\iota_1/a))^{-1}$
⟨*proof*⟩

As explained in Section 5.2, we leave the automation of this process that connects a computer algebra system with the certifications in Isabelle for future work.

The exponential $\exp(t(A\,a\,b)) = \exp(tP^{-1}DP) = P^{-1}\exp(tD)P$ in the flow simplifies to

$$P^{-1}\exp(tD)P = \frac{1}{\sqrt{b^2+4a}}\begin{pmatrix} \iota_2\exp(t\iota_1) - \iota_1\exp(t\iota_2) & \exp(t\iota_2) - \exp(t\iota_1) \\ a\exp(t\iota_2) - a\exp(t\iota_1) & \iota_2\exp(t\iota_2) - \iota_1\exp(t\iota_1) \end{pmatrix}.$$

We also certify this with Isabelle.

lemma *mtx-hOsc-solution-eq*:
defines $\iota_1 \equiv (b - sqrt\ (b^2+4*a))/2$ **and** $\iota_2 \equiv (b + sqrt\ (b^2+4*a))/2$
defines $\Phi\ t \equiv mtx$ (
$[\iota_2{*}exp(t{*}\iota_1) - \iota_1{*}exp(t{*}\iota_2),\quad exp(t{*}\iota_2){-}exp(t{*}\iota_1)]\#$
$[a{*}exp(t{*}\iota_2) - a{*}exp(t{*}\iota_1),\ \iota_2{*}exp(t{*}\iota_2){-}\iota_1{*}exp(t{*}\iota_1)]\#[]$)
assumes $b^2 + a * 4 > 0$ **and** $a \neq 0$
shows $P\ (-\iota_2/a)\ (-\iota_1/a) * (\text{diag}\ i.\ exp\ (t * (if\ i{=}1\ then\ \iota_1\ else\ \iota_2))) * (P\ (-\iota_2/a)\ (-\iota_1/a))^{-1}$
$= (1/sqrt\ (b^2 + a * 4)) *_R (\Phi\ t)$
⟨*proof*⟩

We use this result to prove that the system satisfies the *local-flow* locale. Then, we obtain an instance of (wlp-evol) and prove the correctness specification for the overdamped door.

lemma *local-flow-mtx-hOsc*:
defines $\iota_1 \equiv (b - sqrt\ (b\hat{}2+4*a))/2$ **and** $\iota_2 \equiv (b + sqrt\ (b\hat{}2+4*a))/2$
defines $\Phi\ t \equiv mtx$ (
$[\iota_2{*}exp(t{*}\iota_1) - \iota_1{*}exp(t{*}\iota_2),\quad exp(t{*}\iota_2){-}exp(t{*}\iota_1)]\#$
$[a{*}exp(t{*}\iota_2) - a{*}exp(t{*}\iota_1),\ \iota_2{*}exp(t{*}\iota_2){-}\iota_1{*}exp(t{*}\iota_1)]\#[]$)
assumes $b^2 + a * 4 > 0$ **and** $a \neq 0$
shows *local-flow* $((*_V)\ (A\ a\ b))\ UNIV\ UNIV\ (\lambda t.\ (*_V)\ ((1/sqrt\ (b^2 + a * 4)) *_R \Phi\ t))$
⟨*proof*⟩

lemma *overdamped-door*:
assumes $b^2 + a * 4 > 0$ **and** $a < 0$ **and** $b \leq 0$
shows $PRE\ (\lambda s.\ s\$1 = 0)$
$HP\ (LOOP\ open\text{-}door;\ (x\acute{}=(*_V)\ (A\ a\ b)\ \&\ G)\ INV\ (\lambda s.\ 0 \leq s\$1))$
$POST\ (\lambda s.\ 0 \leq s\ \$\ 1)$
apply(*rule fbox-loopI*, *simp-all add*: *le-fun-def*)
apply(*subst local-flow.fbox-g-ode-subset*[*OF local-flow-mtx-hOsc*[*OF assms*(1)]])
using *assms* **apply**(*simp-all add*: *le-fun-def fbox-def*)
unfolding *sq-mtx-scaleR-eq UNIV-2 sq-mtx-vec-mult-eq*
by (*clarsimp simp*: *overdamped-door-arith*)

Again, $s\$1$ denotes position $s\,x$ and $s\$2$, velocity $s\,v$. The verification assumes $a < 0$ and $b \geq 0$ because $a = -\frac{k}{m}$, $b = -\frac{d}{m}$, and the constants k, d and m are often positive. We also use the postcondition as the loop invariant. Then, the proof uses the *wlp*-rules in its first two lines. The remaining three lines discharge proof obligations by calling the respective arithmetical facts for this problem.

Just like in the previous section, the latest example shows that our formalisation of affine systems of ODEs provides more expressiveness to our components. In this case, the formalisation helps us to certify a diagonalisation and use it to reason about the solution of a system of differential equations in terms of matrices and linear operations. These features are not available in other deductive verification tools.

7.4 Bouncing Ball

In this section we present a classical example from the hybrid systems literature [4, 108]: the bouncing ball. Formalising it allows us to combine loops with the rest of the hybrid programs addressed in previous examples. The bouncing ball verification problem is also more complex than our previous examples but it is still relatively simple. It models a ball that has been dropped from rest at an initial height $h \geq 0$ and that bounces with completely elastic collisions with the floor. The states $s \in \mathbb{R}^{\{x,v\}}$ of the system use $s\,x$ to represent the ball's height and $s\,v$ for its velocity. The discrete control of the hybrid program models the bounce with a conditional statement that tests whether the ball has reached the floor. If it has, an assignment flips the direction of the velocity, otherwise it does nothing. The continuous dynamics model the free fall motion with a guard $G\,s = s\,x \geq 0$ that prohibits the ball to go below ground level. A loop guarantees that this process repeats an indefinite amount of times. Hence, the bouncing ball hybrid program is

$$\begin{aligned} \texttt{ctrl} &= \textbf{if } (\lambda s.\ s\,x = 0) \textbf{ then } v := (\lambda s.\ -s\,v) \textbf{ else skip}, \\ \texttt{ball} &= \textbf{loop } \left((x' = f_g \,\&\, G)_{\mathbb{R}_+}\,;\texttt{ctrl}\right), \end{aligned}$$

where f_g is the constant acceleration vector field of Example 5.2.1 and Section 7.2.

The precondition $P\,s \leftrightarrow (s\,x = h \wedge s\,v = 0)$ describes the initial state of the system. We wish to show that the bouncing ball will never go below ground level or above its original initial height h. Hence, the corresponding partial correctness specification is

$$\{P\}\,\texttt{ball}\,\{Q\},$$

where $Q\,s \leftrightarrow 0 \leq s\,x \leq h$.

To prove the specification, the rules (h-loop-inv) or (wlp-loop) require that we provide a loop invariant. In Example 5.2.1 we showed that a differential invariant for the system is $I\,s \leftrightarrow -\frac{1}{2}(s\,v)^2 = g(h - s\,x)$. Hence, for `ball`'s loop invariant J, we use $J\,s = (G\,s \wedge I\,s)$.

Below we provide two examples. In the first one we verify $\{P\}\,\texttt{ball}\,\{Q\}$ using only invariants. In the second one, we use the flow for f_g.

Example 7.4.1 (Bouncing Ball via Invariants). To prove $\{P\}\,\texttt{ball}\,\{Q\}$, we first rewrite it to its equivalent

$$P \leq |\textbf{loop } \left((x' = f_g \,\&\, G)_{\mathbb{R}_+} \textbf{ inv } I\,;\texttt{ctrl}\right) \textbf{ inv } J]\,Q$$

thanks to the relationship between Hoare triples and forward box operators. We also used the fact that invariant annotations α **inv** i are operationally the same as their original programs α. Then we can apply (wlp-loop) and obtain the proof obligations

$$P \leq J, \qquad J \leq |(x' = f_g \,\&\, G)_{\mathbb{R}_+}\ \textbf{inv}\ I\,;\texttt{ctrl}]\,J, \quad \text{and} \quad J \leq Q.$$

The first and last obligations are immediate: if $s\,x = h \wedge s\,v = 0$, then both I and G hold because $-\frac{1}{2}(s\,v)^2 = -\frac{1}{2}0 = 0 = g(h-h) = g(h - s\,x)$ and $0 \leq 0 = s\,x$. Similarly, from $J\,s$ we get $0 \leq s\,x$ while $I\,s$ implies $s\,x \leq h$ because $-g(h - s\,x)$ remains nonnegative as it is equal to $\frac{1}{2}(s\,v)^2$. In Isabelle/HOL, because of lack of automation for proofs with real numbers, we need to prove this latter fact.

lemma *inv-imp-pos-le*[*bb-real-arith*]: $0 > g \Longrightarrow 2 * g * x - 2 * g * h = v * v \Longrightarrow (x::real) \leq h$
⟨*proof*⟩

Thus, the only remaining proof obligation is $J \leq |(x' = f_g \,\&\, G)_{\mathbb{R}_+}\ \textbf{inv}\ I]\,|\texttt{ctrl}]\,J$, where we have already applied (wlp-seq). Hence, we can apply rule (wlp-inv) of Theorem 5.2.4 to once again obtain three proof obligations

$$I \leq I, \qquad I \leq |(x' = f_g \,\&\, G)_{\mathbb{R}_+}]\,G \wedge I, \quad \text{and} \quad G \wedge I \leq |\texttt{ctrl}]\,J.$$

The first obligation trivially holds. By Proposition 5.2.1 and Example 5.2.1, the second obligation also holds. Finally, the third obligation follows by the chain of simplifications below and the definition of $J = (G \wedge I)$,

$$\begin{aligned} |\texttt{ctrl}]\,J\,s &= |\textbf{if}\ (\lambda s.\ s\,x = 0)\ \textbf{then}\ v := (\lambda s.\ -s\,v)\ \textbf{else skip}]\,J\,s \\ &= (s\,x = 0 \wedge |v := \lambda s.\ -s\,v]\,J\,s) \vee (s\,x \neq 0 \wedge |\textbf{skip}]\,J\,s) \\ &= (s\,x = 0 \wedge J\,s[v \mapsto -s\,v]) \vee (s\,x \neq 0 \wedge J\,s) \\ &= (s\,x = 0 \wedge J\,s) \vee (s\,x \neq 0 \wedge J\,s) = J\,s. \end{aligned}$$

Therefore, we have discharged all branches of the proof tree and we have verified that $\{P\}\,\texttt{ball}\,\{Q\}$. In Isabelle/HOL, the argument presented in this example is a brief proof.

lemma *bouncing-ball-inv*: $g < 0 \Longrightarrow h \geq 0 \Longrightarrow$
$(\lambda s.\ s\$1 = h \wedge s\$2 = 0) \leq$
$|LOOP\ ($
$(x' = (f\ g)\ \&\ (\lambda\ s.\ s\$1 \geq 0)\ DINV\ (\lambda s.\ 2 * g * s\$1 - 2 * g * h - s\$2 * s\$2 = 0))\ ;$
$(IF\ (\lambda\ s.\ s\$1 = 0)\ THEN\ (2 ::= (\lambda s.\ -s\$2))\ ELSE\ skip))$
$INV\ (\lambda s.\ 0 \leq s\$1 \wedge 2 * g * s\$1 - 2 * g * h - s\$2 * s\$2 = 0)]$
$(\lambda s.\ 0 \leq s\$1 \wedge s\$1 \leq h)$
apply(*rule fbox-loopI*, *simp-all*, *force*, *force simp*: *bb-real-arith*)
by (*rule fbox-g-odei*) (*auto intro*!: *poly-derivatives diff-invariant-rules*)

Above, we use again $s\$1$ for the position of the ball and $s\$2$ for its velocity. Similarly, *DINV* is our notation in Isabelle/HOL for α **inv** i applied specifically to evolution commands. The first line of the proof blasts away the program structure leaving only evolution commands. The second line concludes by first applying (wlp-inv) and then discharging the arithmetic and invariance obligations automatically with our tactics of Section 4.2. □

Example 7.4.2 (Bouncing Ball via Flow). Alternatively to our proof above, we can compute the *wlp* of the evolution command in the bouncing ball. That is, we retake the proof starting from the proof obligation $J \leq |(x' = f_g \,\&\, G)_{\mathbb{R}_+}]\,|\texttt{ctrl}]\,J$.

We know from Section 7.2, that the flow φ for f_g is given by

$$\begin{pmatrix} \varphi\, t\, s\, x \\ \varphi\, t\, s\, v \end{pmatrix} = \begin{pmatrix} g\frac{t^2}{2} + (s\, v)t + s\, x \\ gt + s\, v \end{pmatrix}.$$

Therefore, applying (wlp-evol) we get

$$|(x' = f_g \,\&\, G)_{\mathbb{R}_+}]\,|\texttt{ctrl}]\,J\, s = \forall t \geq 0.\ (\forall \tau \in [0, t].\ G\,(\varphi\, \tau\, s)) \rightarrow G\,(\varphi\, t\, s) \wedge I\,(\varphi\, t\, s),$$

where

$$I\,(\varphi\, t\, s) = \left(-\frac{1}{2}(gt + s\, v)^2 = g(h - g\frac{t^2}{2} - (s\, v)t - s\, x)\right) = \left(-\frac{1}{2}(s\, v)^2 = g(h - s\, x)\right) = I\, s.$$

Hence, $J \leq |(x' = f_g \,\&\, G)_{\mathbb{R}_+}]\,|\texttt{ctrl}]\,J$ simplifies to

$$\forall s.\ G\, s \wedge I\, s \rightarrow (\forall t \geq 0.\ (\forall \tau \in [0, t].\ G\,(\varphi\, \tau\, s)) \rightarrow I\, s),$$

which trivially holds. The proof in Isabelle is a two line proof as in the previous example. Yet, this time, the preliminary lemmas for arithmetic with real numbers are more since we have to add the fact that $I\,(\varphi\, t\, s) = I\, s$. □

For complete coverage of our components, we show the code for the verification of the bouncing ball with the flow φ directly written in the specification. The formalisation uses lenses and the Isabelle/UTP framework. Hence, instead of computing *wlp*s the proof uses the rules of $\mathsf{d}\mathcal{H}$. We also include the refinement of the bouncing ball. The proof is the same as in the recent Example 7.4.2 but without certification of derivatives or Lipschitz continuity.

```
abbreviation ball-flow :: real ⇒ real ⇒ (real^2) usubst (φ)
  where φ g τ ≡ [x ↦s g · τ ^ 2/2 + v · τ + x, v ↦s g · τ + v]

abbreviation bb-evol g h T ≡
  LOOP (EVOL (φ g) (x ≥ 0) T; (IF (x = 0) THEN (v ::= −v) ELSE skip))
  INV (0 ≤ x ∧ 2 · g · x = 2 · g · h + v · v)

lemma bouncing-ball-dyn:
  assumes g < 0 and h ≥ 0
  shows {x = h ∧ v = 0} bb-evol g h T {0 ≤ x ∧ x ≤ h}
  apply(hyb-hoare U(0 ≤ x ∧ 2 · g · x = 2 · g · h + v · v))
  using assms by (rel-auto′ simp: bb-real-arith)

lemma R-bouncing-ball-dyn:
  assumes g < 0 and h ≥ 0
  shows [x = h ∧ v = 0, 0 ≤ x ∧ x ≤ h] ≥ bb-evol g h T
  apply(refinement; (rule R-bb-assign[OF assms])?)
  using assms by (rel-auto′ simp: bb-real-arith)
```

The proofs of the bouncing ball in Isabelle/HOL are simple because we have added the *wlp*-rules of Section 3.3 to its simplifier which allows Isabelle to do equational reasoning on the program structure. In the case of differential Hoare logic and the refinement calculus, simple tactics *hyb-hoare* and *refinement* supply this increased proof-automation in our framework. The example also displays one of the common consequences of doing deductive verification. Namely, wherever we have not automated the verification process, the proofs' lengths increase. Specifically, as Isabelle lacks support for reasoning about arithmetical facts with real numbers, we have to provide proofs for them separately. Then, we group them in the list of theorems *bb-real-arith* to supply them as part of the *auto* tactic. Contrastingly, other more mature deductive theorem provers like KeYmaera X and the HHL prover can verify the bouncing ball automatically [113]. Yet, together, the three different verifications of the bouncing ball (by invariants, with the flow and with annotated dynamics) cover no more than two pages of proofs in our setting. The longest of them would require four proofs: three proofs of the arithmetical facts in a mathematical style and one of the actual verification.

Overall, the bouncing ball is an important example to test our framework because it shows that the verification components can handle traditional programs like while-loops, conditional branching and assignments, together with the differential equations. The next examples focus on other aspects of the verification components.

7.5 Water Tank

To this point, we have only refined `thermostat` in Example 6.1.1 by doing single refinements in different lemmas and combining them in a final result. Alternatively, we can do an ordered sequence of refinements that build the final hybrid program in one structured proof. This section illustrates such a refinement.

We consider a controller that turns a water pump on and off to keep the volume of water h in a tank within the bounds $h_l \leq h \leq h_h$. Just like for `thermostat`, the controller registers the initial level of water h_0 at the beginning of each of its interventions. It also uses variable π to indicate whether the pump is on or off. The inflow of water is linear at a rate c_i while the outflow's rate is c_o. Therefore, the overal rate of change is proportional to $k \in \{-c_o, c_i - c_o\}$, depending on the value of π, where $c_i > c_o$. Figure 7.3 depicts the system.

Given the linear behaviour of the flow of water, the differential equation $h'\,t = k$ is part of the dynamics of the system. The rest of the variables either do not change $x'\,t = 0$ or define time $t' = 1$. We formalise this with the Isabelle/UTP framework below.

abbreviation $f\ k \equiv [\pi \mapsto_s 0,\ h \mapsto_s k,\ h_0 \mapsto_s 0,\ t \mapsto_s 1]$

We use this vector field in the definition of the dynamics for the hybrid system. They indicate that if the pump is turned on, the evolution command involves both inflow and outflow rates $h'\,t = c_i - c_o$. If it is off, it only uses the outflow rate $h'\,t = -c_o$.

abbreviation $G\ h_x\ k \equiv \mathbf{U}(t \leq (h_x - h_0)/k)$

abbreviation $dI\ h_l\ h_h\ k \equiv \mathbf{U}(h = k \cdot t + h_0 \wedge 0 \leq t \wedge h_l \leq h_0 \wedge h_0 \leq h_h \wedge (\pi = 0 \vee \pi = 1))$

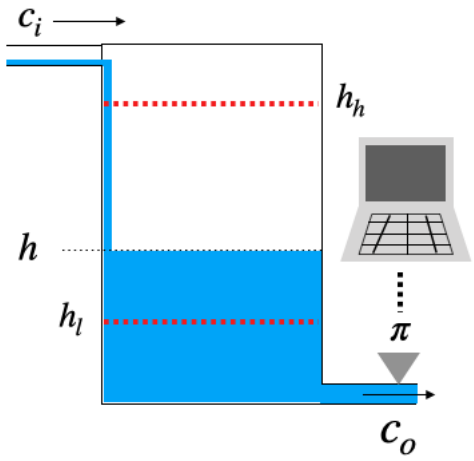


Figure 7.3: A controller for a water tank that should not be emptied nor spilled out

abbreviation *dyn* c_i c_o h_l h_h $\tau \equiv$ *IF* $(\pi = 1)$ *THEN*
$x' = f\ (c_i - c_o)$ & $G\ h_h\ (c_i - c_o)$ *on* $\{0..\tau\}$ *UNIV* @ *0 DINV* $(dI\ h_l\ h_h\ (c_i - c_o))$
ELSE $x' = f\ (-c_o)$ & $G\ h_l\ (-c_o)$ *on* $\{0..\tau\}$ *UNIV* @ *0 DINV* $(dI\ h_l\ h_h\ (-c_o))$

Similarly to `thermostat`, guards $G\,h_h\,(c_i - c_o)$ and $G\,h_l\ - c_o$ limit the duration of the dynamics to guarantee the intervention of the control before the water level gets close to the boundaries. The invariant for the evolution commands is an assertion of four statements: time is nonnegative, the water level remains safe $h_l \leq h_0 \leq h_h$, the pump is either on or off $\pi = 0 \vee \pi = 1$, and the solution to the differential equation is $h = kt + h_0$. The proof of invariance involves our tactics *poly-derivatives* and *diff-invariant-rules*. It is longer than previous proofs because of the inequalities and multiple conjuncts in the invariant. Notice that it is also independent on the choice of the guard, hence the variable *Guard*.

lemma *tank-diff-inv*: $0 \leq \tau \Longrightarrow$ *diff-invariant* $(dI\ h_l\ h_h\ k)\ (f\ k)\ \{0..\tau\}$ *UNIV 0 Guard*
⟨*proof*⟩

The control of the water tank is also similar to that of `thermostat`. That is, it resets t and h_0 and then turns the water pump on or off according to its readings.

abbreviation *ctrl* h_l $h_h \equiv$
$(t ::= 0);(h_0 ::= h);$
$(IF\ (\pi = 0 \wedge h_0 \leq h_l + 1)\ THEN\ (\pi ::= 1)\ ELSE$
$(IF\ (\pi = 1 \wedge h_0 \geq h_h - 1)\ THEN\ (\pi ::= 0)\ ELSE\ skip))$

Then, we integrate control and dynamics in the standard way with a finite iteration. The loop invariant is the same as the invariant for evolution commands without restrictions on the solutions to the differential equations.

abbreviation $I\ h_l\ h_h \equiv \mathsf{U}(h_l \leq h \wedge h \leq h_h \wedge (\pi = 0 \vee \pi = 1))$

abbreviation *tank-dinv* $c_i\ c_o\ h_l\ h_h\ \tau \equiv$
$LOOP\ (ctrl\ h_l\ h_h;\ dyn\ c_i\ c_o\ h_l\ h_h\ \tau)\ INV\ (I\ h_l\ h_h)$

Then, we refine the hybrid program for the water tank in a single structured proof.

lemma *R-tank-inv*:
assumes $0 \leq \tau$ **and** $0 < c_o$ **and** $c_o < c_i$
shows $[I\ h_l\ h_h,\ I\ h_l\ h_h] \geq$ *tank-dinv* $c_i\ c_o\ h_l\ h_h\ \tau$
proof−
have $[I\ h_l\ h_h,\ I\ h_l\ h_h] \geq$
$LOOP\ ((t ::= 0);[I\ h_l\ h_h \wedge t = 0,\ I\ h_l\ h_h])\ INV\ I\ h_l\ h_h$ (**is** - ≥ *?R1*)
by (*refinement*, *rel-auto′*)
moreover have *?R1* ≥ *LOOP*
$((t ::= 0);(h_0 ::= h);[I\ h_l\ h_h \wedge t = 0 \wedge h_0 = h,\ I\ h_l\ h_h])\ INV\ I\ h_l\ h_h$ (**is** - ≥ *?R2*)
by (*refinement*, *rel-auto′*)
moreover have *?R2* ≥
$LOOP\ (ctrl\ h_l\ h_h;[I\ h_l\ h_h \wedge t = 0 \wedge h_0 = h,\ I\ h_l\ h_h])\ INV\ I\ h_l\ h_h$ (**is** - ≥ *?R3*)
by (*simp only*: *mult.assoc*, *refinement*; (*force*)*?*, (*rule R-assign-law*)*?*) *rel-auto′*
moreover have *?R3* ≥ *LOOP* $(ctrl\ h_l\ h_h;\ dyn\ c_i\ c_o\ h_l\ h_h\ \tau)\ INV\ I\ h_l\ h_h$
apply(*simp only*: *mult.assoc*, *refinement*; (*simp*)*?*)
prefer *4* **using** *tank-diff-inv assms* **apply** *force+*
using *tank-inv-arith1 tank-inv-arith2 assms* **by** *rel-auto′*
ultimately show $[I\ h_l\ h_h,\ I\ h_l\ h_h] \geq$ *tank-dinv* $c_i\ c_o\ h_l\ h_h\ \tau$
by *auto*
qed

The proof is an ordered sequence of refinements to finally obtain the specification of *tank-dinv* using the laws of $\mathsf{d}\mathcal{R}$. It uses schematic variables *?R* to abbreviate long expressions. The first refinement introduces the loop structure and the first assignment. A second refinement brings in the second assignment and the precondition for the conditional statement together with the postcondition of the dynamics. Hence, in the third refinement, the control is already in the specification. The final refinement brings the dynamics via lemma *tank-diff-inv* to conclude the proof, as the result follows by transitivity.

Alternatively, the verification of the water tank with $\mathsf{d}\mathcal{H}$ is automatic.

lemma *tank-inv*:
assumes $0 \leq \tau$ **and** $0 < c_o$ **and** $c_o < c_i$
shows $\{I\ h_l\ h_h\}$ *tank-dinv* $c_i\ c_o\ h_l\ h_h\ \tau$ $\{I\ h_l\ h_h\}$
apply(*hyb-hoare* $\mathsf{U}(I\ h_l\ h_h \wedge t = 0 \wedge h_0 = h)$)
prefer *4* **prefer** *7* **using** *tank-diff-inv assms* **apply** *force+*
using *assms tank-inv-arith1 tank-inv-arith2* **by** *rel-auto′*

As before, our tactic *hyb-hoare* blasts away the control structure and provides the loop-invariant. The second line tackles evolution commands with lemma *tank-diff-inv*. Arithmetical lemmas proven separately help us finish the proof.

Overall, the water tank allows us to test the refinement components together with the nicer syntax provided by our state-space abstraction with lenses of Chapter 6. We plan to include both extensions in future versions of the components with predicate transformers and *wlp*s. In comparison with other deductive verification tools, on one hand, the HHL prover does not have a refinement calculus. Nevertheless, it includes other capabilities from the calculus of hybrid communicating sequential processes (HCSP) and the duration calculus that are missing in our implementation. For instance, it includes features to reason about communicating and parallel processes as well as mechanisms to reason about the history of a formula through a "chop" operator [135]. Differential dynamic logic on the other hand does have support for a different kind of refinement [88] to the one presented here. In differential refinement logic ($\mathsf{d}\mathcal{RL}$), one is interested in safely and modularly augmenting simple hybrid programs whereas in our approach, we are interested in building hybrid programs from specifications. However, there is some overlap in the proof rules that both approaches use, specifically, in those rules designed for refinement of regular programs. Yet, our approach does not cover $\mathsf{d}\mathcal{RL}$'s rules for refinement of ODEs and loops as our original motivation for the introduction of $\mathsf{d}\mathcal{R}$ did not intend to simulate $\mathsf{d}\mathcal{RL}$. We leave for future work implementing the refinement rules of that calculus.

7.6 Select ARCH2020 benchmarks

In Section 5.6, we argued that our components do not need all the rules of differential dynamic logic for evolution commands. In particular, we did not prove the differential effect and differential ghosts rules. To support that statement, in this section we present various benchmark problems from the friendly competition of the 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH2020) [97]. In particular, we focus on the problems of the Hybrid Systems Theorem Proving category, where state of the art tools like KeYmaera X [50] and the HHL Prover [135] participate yearly.

In this category, competitors can choose to verify any of the 214 benchmark problems in any of three formats: automated, hints and scripted. To participate in the first mode, the prover must solve the verification problem after only receiving its specification. In the hinted version, annotated loop-invariants and other aids in the description of the problem are also allowed. Finally, interactive verification with user inputs is the scripted format. The benchmark problems are classified as follows: 60 design shapes, 10 problems of case studies, 3 games and 141 nonlinear. The design shape problems test the essentials for verification of hybrid programs with a small focus on $\mathsf{d}\mathcal{L}$'s rules. They go from simple assignments, tests, choices, finite iterations and evolution commands, to problems that specifically need any of the rules described in Section 5.6. The case studies come from previously published verifications, for instance [77] and [116]. The game category checks the prover's ability to verify adversarial dynamics of differential game logic [111]. Finally, the nonlinear problems cover various verifications whose vector field's defining equations are polynomial systems, that is, $x_i' t = P_i\,(t, x_1\,t, \ldots, x_n\,t)$ where P_i is a polynomial function.

We participated in the ARCH2020 competition in the scripted format and decided to focus on all 60 design shape problems and one problem of a case study. We focused on the scripted format because the components still lack proof automation for manipulating real

arithmetic expressions as evidenced in Section 7.4. For the same reason and because our components do not include adversarial dyanimcs, we restrict our participation to mostly the design shape problems. Moreover, the span of a month was too short to cover all 211 problems interactively. In fact, at the end of that period, we fully verified, without any help from an external tool, 52 out of the 60 design shape problems together with the first problem of the first case study. Three of these unsolved design shape problems and another problem of the first case study depend on properties of real numbers that we did not certify with Isabelle but checked with Mathematica® 12.1. By asserting these properties (without proving them) in the proof assistant, the verifications of their corresponding problems succeed. We verified one more of the remaining 5 unsolved problems but not in time to submit it to the competition. The other four require extensions to our components like adding rules of inference relating diamond operators and differential equations or variants of Picard-Lindelöf theorem.

The examples below discuss some of the competition's most challenging problems for our verification components. For reference to the competition we have named each example with the problem's name. After each explanation of the proof obligations we reflect on possible improvements to our framework.

Example 7.6.1 (Potentially overwrite exponential decay). Our first example involves the differential equation $x'\,t = -x\,t$. The solutions to their respective IVPs are exponentials $x\,t = x_0 \exp(t_0 - t)$ with initial condition (t_0, x_0), hence, decreasing functions. However, no matter how much $x\,t$ decreases, it still remains above 0 if $x_0 > 0$. This is the property that we must prove for this benchmark problem. In more detail, the correctness specification is

$$x > 0 \wedge y > 0 \to |x' = -x]\,|(\textbf{loop}\ x := x + 3\ \textbf{inv}\ x > 0) + y := x]\,x > 0 \wedge y > 0,$$

where we have omitted lambdas λ and references to states s. We also use $(x' = f\,x)$ instead of $(x' = f\,\&\,\top)_{\mathbb{R}_+}$. Notice that an argument with invariants is not possible since, by the second item of lemma 5.2.5, $x' \leq 0' \leftrightarrow -x \leq 0 \leftrightarrow \bot$, given that $x > 0$.

In these cases, $\mathsf{d}\mathcal{L}$'s syntax does not cover exponentials by default, but the logic overcomes this obstacle with the differential ghost rule to introduce an auxiliary ODE that aids in the proof of the specification [117]. Yet, in our case, we have the full expressiveness of higher order logic. Hence, in simple cases like this one, we can introduce the solution without any reference to auxiliary ODEs. We discuss more complicated dynamics in the examples below.

The dynamics $x'\,t = -x\,t$ repeat in various benchmark problems. Nevertheless, we can always supply the solution and verify the corresponding partial correctness specifications. As usual, we formalise the vector field, its flow and prove that they satisfy the assumptions of our locales.

```
abbreviation po-exp-dec-f :: real^2 ⇒ real^2 (f)
  where f s ≡ (χ i. if i=1 then −s$1 else 0)

abbreviation po-exp-dec-flow :: real ⇒ real^2 ⇒ real^2 (φ)
  where φ t s ≡ (χ i. if i=1 then s$1 ∗ exp (− t) else s$i)

lemma local-flow-exp-flow: local-flow f UNIV UNIV φ
  ⟨proof⟩
```

Afterwards, we verify the hybrid program.

```
lemma ⌈λs::real^2. s$1 > 0 ∧ s$2 > 0⌉ ≤ wp (x´= f & G)
  (wp ((LOOP (1 ::= (λs. s$1 + 3)) INV (λs. 0 < s$1)) ∪ (2 ::= (λs. s$1)))
  ⌈λs. s$1 > 0 ∧ s$2 > 0⌉)
  apply(subst rel-aka.fbox-mult[symmetric])
  apply(rule rel-aka.fbox-seq-var)
   apply(subst local-flow.wp-g-ode-ivl[OF
      local-flow-exp-flow, where Q=λs. s$1 > 0 ∧ s$2 > 0]; simp)
  apply(subst le-wp-choice-iff, rule conjI)
   apply(subst change-loopI[where I=λs. s$1 > 0 ∧ s$2 > 0])
  by (rule wp-loopI, auto)
```

From the formalisation above, it is evident that $s\$1$ denotes x while $s\$2$ corresponds to y. The proof is longer than proofs in previous sections because we have not provided yet proof automation for manipulating the simplest hybrid programs. That is why the first line applies (wlp-seq) in reverse to obtain a simple forward box. The second line applies (h-seq) to split the structure. The third line discharges the first proof obligation by supplying the flow. The next line addresses the nondeterministic choice, while the remaining two lines discharge each of the emerging branches of the proof tree: one for the loop, the other for the assignment. This is a common pattern with our proofs of the benchmark problems of the competition. They tend to be longer because they use choices, tests and separated box operators instead of our typical if-then-else statements. In future work we plan to provide automation for all hybrid programs and not just while programs. □

Example 7.6.2 (Dynamics: Exponential growth (2)). Our next example shows how we can solve problems in the style of $\mathtt{d}\mathcal{L}$. In particular, we use the method described in Section 5.6. That is, we use differential cuts to add invariants to guards. This strengthening enables us to prove the postcondition merely because the guard implies it. The following specification is among the simplest problems where we used this approach

$$x \geq 0 \land y \geq 0 \rightarrow [x' = y, y' = y^2]\, x \geq 0.$$

We simplify notation by writing ODEs directly in evolution commands without vector fields.

The formalisation is straightforward.

```
lemma ⌈λs::real^2. s$1 ≥ 0 ∧ s$2 ≥ 0⌉ ≤
  wp (x´=(λt s. (χ i. if i=1 then s$2 else (s$2)^2)) & G) ⌈λs. s$1 ≥ 0⌉
  apply(rule-tac C=λs. s$2 ≥ 0 in diff-cut-rule)
  apply(subst g-ode-inv-def[symmetric, where I=λs. s$2 ≥ 0], rule wp-g-odei; simp?)
   apply(rule-tac ν'=λs. 0 and μ'=λs. (s$2) ^2 in diff-invariant-rules(2); (simp add: forall-2)?)
  apply(rule-tac C=λs. s$1 ≥ 0 in diff-cut-rule, simp-all)
  apply(subst g-ode-inv-def[symmetric, where I=λs. s$1 ≥ 0], rule wp-g-odei; simp?)
   apply(rule-tac ν'=λs. 0 and μ'=λs. (s$2) in diff-invariant-rules(2); (simp add: forall-2)?)
  by (rule diff-weak-rule, simp)
```

The first line above uses a differential cut to introduce in the guard the expression $y \geq 0$. The second line then applies (wlp-inv) from Theorem 5.2.4 while the third line completes the

proof of invariance for $y \geq 0$ by using Lemma 5.2.5. Notice that for inequalities, users have to supply the terms μ' and ν' of that lemma. An improved *diff-invariant-rules* tactic that does this automatically is left for future work. The next lines repeat this process for $x \geq 0$ and the last line uses a weakening to indicate that the guard $G \wedge y \geq 0 \wedge x \geq 0$ implies the postcondition $x \geq 0$.

Alternatively, Mathematica® 12.1 provides us with the solution to the ODEs.

```
DSolve[{x'[t] == y[t], y'[t] == y[t]^2, x[0] == x0, y[0] == y0}, {x, y}, t]
```

$$\left\{\left\{y \to \texttt{Function}\left[\{t\}, -\frac{\texttt{y0}}{t\texttt{y0}-1}\right], x \to \texttt{Function}\left[\{t\}, \texttt{x0} - \log\left[t - \frac{1}{\texttt{y0}}\right] + \log\left[-\frac{1}{\texttt{y0}}\right]\right]\right\}\right\}$$

The two functions $y\,t = -y_0/(ty_0 - 1)$ and $x\,t = x_0 - \log(t - 1/y_0) + \log(-1/y_0)$ can be written in Isabelle/HOL and supplied as the flow. We follow this approach for the next example. Nevertheless, the fact that we can employ the $\mathsf{d}\mathcal{L}$-invariant approach is a testament to the flexibility of our components. □

Example 7.6.3 (Darboux inequality). We solved this problem after the competition was finished. It is a good example to highlight delicate details when doing verification of hybrid systems. At first glance, the dynamics for this problem resemble those of the previous example. The specification given is

$$x + z \geq 0 \to [x' = x^2, z' = zx + y \;\&\; y = x^2]\, x + z \geq 0.$$

However, unlike the previous benchmark, an invariant argument does not suffice. To see this consider Lemma 5.2.5.2 and observe that $(x + z)' = x' + z' = x^2 + zx + y = 2x^2 + zx$ and $2x^2 + zx$ is not always greater than 0. Hence, we proceed as in Example 7.6.1.

We obtain the solutions with Mathematica

$$x\,t = \frac{x_0}{1 - x_0 t}, \qquad z\,t = \frac{z_0 - x_0 \log(1 - x_0 t)}{1 - x_0 t}.$$

However, the first detail that we must highlight is that the domain of the solutions needs to be treated carefully. In particular, taking the constant function $\lambda s.\ \mathbb{R}_+$ as in previous examples makes the specification incorrect. A correct function is $\lambda s.\ \{t \geq 0 \mid 1 - (s\,x)t > 0\}$ because this ensures that the solutions are defined. This simple example illustrates why we use $U : S \to \mathcal{P}\,S$ in our definition of guarded orbitals in Section 5.1. Such a focus on the interval of existence is not present in $\mathsf{d}\mathcal{L}$ since its semantics alleviate thinking about them. It makes the verification with our components more subtle and harder. Yet, thinking about these constraints in the context of verification is a positive feature that increases trust in the final result.

With this in mind, we can formalise the fact that the equations above solve the ODE.

```
abbreviation darboux-ineq-f :: real^2 ⇒ real^2 (f)
  where f s ≡ (χ i. if i=1 then (s$1)^2 else (s$2)*(s$1)+(s$1)^2)

abbreviation darboux-ineq-flow2 :: real ⇒ real^2 ⇒ real^2 (φ)
  where φ t s ≡ (χ i. if i=1 then (s$1/(1 − t ∗ s$1)) else
    (s$2 − s$1 ∗ ln(1 − t ∗ s$1))/(1 − t ∗ s$1))
```

lemma *darboux-flow-ivp*: $(\lambda t.\ \varphi\ t\ s) \in Sols\ (\lambda t.\ f)\ (\lambda s.\ \{t.\ 0 \leq t \wedge t * s\ \$\ 1 < 1\})\ UNIV\ 0\ s$
by (*rule ivp-solsI*) (*auto intro*!: *poly-derivatives*
simp: *forall-2 power2-eq-square add-divide-distrib power-divide vec-eq-iff*)

Despite the presence of logarithms in the flow, our *poly-derivatives* tactic still makes the certification proof relatively easy. Nevertheless, another issue with this system occurs in the Lipschitz continuity argument. Ideally, we would like to use the fact that because $x'\,t$ and $z'\,t$ are differentiable, then the corresponding vector field is Lipschitz continuous. However, this fact is not yet present in Isabelle's libraries of Analysis. The alternative direct proof by definition, is complicated due to the term x^2 on the right hand side of the system of ODEs. Thus, we have two options. The first one asserts in the proof assistant that the vector field satisfies Picard-Lindelöf's theorem without actually certifying this fact. The second one skips this process, considers the flow certification above enough, and uses our evolution commands based on direct annotations of the dynamics. We have done both, although here we just show the second one because their proofs only differ by the line that supplies the solution.

lemma *darboux-ineq-arith*:
assumes $0 \leq s_1 + s_2$ **and** $0 \leq (t::real)$ **and** $t * s_1 < 1$
shows $0 \leq s_1\ /\ (1 - t * s_1) + (s_2 - s_1 * ln\ (1 - t * s_1))\ /\ (1 - t * s_1)$
⟨*proof*⟩

lemma $\lceil \lambda s::real\hat{}2.\ s\$1 + s\$2 \geq 0 \rceil \leq$
$wp\ (EVOL\ \varphi\ (\lambda s.\ y = (s\$1)\hat{}2)\ (\lambda s.\ \{t.\ 0 \leq t \wedge t * s\ \$\ 1 < 1\}))$
$\lceil \lambda s.\ s\$1 + s\$2 \geq 0 \rceil$
apply(*subst wp-g-dyn*, *simp-all*)
using *darboux-ineq-arith* **by** *smt*

The proof applies rule (wlp-g-dyn) and uses the real arithmetical fact *darboux-ineq-arith* to discharge the emerging proof obligation.

In conclusion, this example highlights several characteristics of our components. Firstly, it evidences the fact that, in a strongly typed setting like Isabelle/HOL, a correct specification considers the dependency of the solution on the domain of the vector field. Secondly, it effectively uses our dynamics based evolution commands and makes a case in our setting for preferring them over the traditional versions that require further certifications. □

Example 7.6.4 (Darboux equality). This problem and the next one are proofs of invariance. In $\mathsf{d}\mathcal{L}$, one can solve them with the help of the differential ghost rule. However, due to our HOL-expressiveness, we do not need it. We proved both problems interactively with Isabelle's scripting language Isar for mathematical proofs. This was necessary because the rules of inference and procedures presented so far were insufficient. Consequently, the length of our proofs for both problems increased. Overall, this new requirements evidence that our semantic tactics for invariants can be strengthened. Alternatively, we could benefit from formalising the differential ghost rule. Exploring both approaches and their relative strength is an interesting project [117] that we leave for future work.

The partial correctness specification for this example is

$$x + z = 0 \to [x' = Ax^2 + Bx, z' = Azx + Bz]\, 0 = -x - z.$$

Once again, if we try to use Lemma 5.2.5.1, we obtain the following chain of equations

$$(x+z)' = x' + z' = Ax^2 + Bx + Azx + Bz = x(Ax+B) + z(Ax+B) = (x+z)(Ax+B).$$

The final result is only 0 if we take into account the precondition $x + z = 0$. But this is not stated in the lemma. Nevertheless, our definition of invariance does consider the initial state of the system. This is evident in the assumptions $s \in I$ and $X \in \mathsf{Sols}\ f\ U\ S\ t_0\ s$ of Proposition 5.2.2.3.

Therefore, to verify the benchmark problem, we unfolded the characterisation of invariance for this specific problem. That is, we have to prove that if a function $X : \mathbb{R} \to \mathbb{R}^{\{x,z\}}$ satisfies $X\,0\,x + X\,0\,z = 0$, $X'\,t\,x = A(X\,t\,x)^2 + B(X\,t\,x)$ and $X'\,t\,z = A(X\,t\,z)(X\,t\,x) + B(X\,t\,z)$, then it also satisfies $X\,\tau\,x + X\,\tau\,z = 0$ for all $\tau \in \mathbb{R}$. This follows by uniqueness of the solutions to the differential equation $y'\,t = (A(X\,t\,x) + B)y\,t$. In particular, notice that the constant function 0 and $\lambda t.\ X\,t\,x + X\,t\,z$ satisfy such ODE, hence they are the same. In Isabelle/HOL, the formalisation of this argument is in the scripted language Isar and shown below.

```
lemma ⌈λs::real^2. s$1 + s$2 = 0⌉ ≤
  wp (x´= (λt s. (χ i. if i=1 then A*(s$1)^2+B*(s$1) else A*(s$2)*(s$1)+B*(s$2))) &
  G on (λs. UNIV) UNIV @ 0)
  ⌈λs. 0 = - s$1 - s$2⌉
proof−
  have key: diff-invariant (λs. s $ 1 + s $ 2 = 0)
    (λt s. χ i. if i = 1 then A*(s$1)^2+B*(s$1) else A*(s$2)*(s$1)+B*(s$2))
  (λs. UNIV) UNIV 0 G
  proof(clarsimp simp: diff-invariant-eq ivp-sols-def forall-2)
    fix X::real⇒real^2 and t::real
    let ?y = (λt. X t $ 1 + X t $ 2)
    assume init: ?y 0 = 0
      and D1: D (λt. X t $ 1) = (λt. A · (X t $ 1)^2 + B · X t $ 1) on UNIV
      and D2: D (λt. X t $ 2) = (λt. A · X t $ 2 · X t $ 1 + B · X t $ 2) on UNIV
    hence D ?y = (λt. ?y t * (A · (X t $ 1) + B)) on UNIV
      by (auto intro!: poly-derivatives simp: field-simps power2-eq-square)
    hence D ?y = (λt. (A · X t $ 1 + B) · (X t $ 1 + X t $ 2)) on {0−−t}
      using has-vderiv-on-subset[OF - subset-UNIV[of {0−−t}]] by (simp add: mult.commute)
    moreover have continuous-on UNIV (λt. A · (X t $ 1) + B)
      apply (rule vderiv-on-continuous-on)
      using D1 by (auto intro!: poly-derivatives simp: field-simps power2-eq-square)
    moreover have D (λt. 0) = (λt. (A · X t $ 1 + B) · 0) on {0−−t}
      by (auto intro!: poly-derivatives)
    moreover note picard-lindeloef.ivp-unique-solution[OF
      picard-lindeloef-first-order-linear[OF UNIV-I open-UNIV is-interval-univ calculation(2)]
      UNIV-I is-interval-closed-segment-1 subset-UNIV -
      ivp-solsI[OF - - funcset-UNIV, of ?y]
```

```
    ivp-solsI[OF - - funcset-UNIV, of λt. 0], of t λs. 0 0 λs. t 0]
   ultimately show X t $ 1 + X t $ 2 = 0
    using init by auto
 qed
 show ?thesis
   apply(subgoal-tac (λs. 0 = − s$1 − s$2) = (λs. s$1 + s$2 = 0), erule ssubst)
   using key by auto
qed
```

The proof consists in guaranteeing the assumptions of our lemma *ivp-unique-solution* in the *picard-lindeloef* locale. We reference this lemma near the end of the proof and use the schematic variable *?y* to abbreviate $(\lambda t.\ X\ t\ \$\ 1 + X\ t\ \$\ 2)$. The middle part of the proof shows that *?y* and the constant 0 function satisfy the same ODE. In combination with uniqueness, these facts complete our proof.

In contrast, the alternative method of using the solutions to the system of ODEs is not appealing. For instance, $z\,t$ involves nested functions of fractions of exponentials of logarithms of difference of exponentials of fractions of logarithms. Proving arithmetical facts about such complicated expressions would be extremely time consuming. This fact reinforces the need of better proof automation for manipulation of expressions with real numbers in Isabelle.

Therefore, uniqueness theorems are crucial for some proofs of invariance. In this case, we managed to have an adequate formalisation in *picard-lindeloef.ivp-unique-solution*. However, another similar benchmark problem with the dynamics

$$x' = (Ay + Bx)/z^2, z' = (Ax + B)/z\ \&\ y = x^2\ \&\ z^2 > 0$$

is not as easy to verify because the underlying domain is not all of $\mathbb{R}$ and $\mathbb{R}^{\{x,z\}}$. In fact, it is still unsure if the existing uniqueness theorems in Isabelle/HOL allow us to prove this alternative version. We suspect that a different formalisation of Picard-Lindelöf's theorem with closed intervals instead of open intervals as in *picard-lindeloef* might be necessary. Hence, the creation of generic strategies that automate the proof given in this example is left for future work. □

Example 7.6.5 (Open Cases). Contrastingly with the previous example, here we need to prove invariance for a couple of strict inequalities. The partial correctness specification is

$$x^3 > 5 \wedge y > 2 \to [x' = x^3 + x^4, y' = 5y + y^2]\, x^3 > 5 \wedge y > 2.$$

Again we can see that

$$\begin{aligned}(x^3)' &= 3x^2x' = 3x^2(x^3 + x^4),\\ y' &= 5y + y^2,\end{aligned}$$

and neither of them are always greater or equal to 0. Hence, Lemma 5.2.5.2 does not apply. Moreover, it is not clear how the uniqueness argument of the previous example would help us for this inequality. However, the initial condition is still relevant because if $x^3 > 5 > 0$ and $y > 2 > 0$, then $3x^2(x^3 + x^4) > 0$ and $5y + y^2 > 0$. Thus, we need to show that any function X that starts above a constant $X\,t_0 \geq c$, and whose derivative $X'\,t$ is greater than 0 after that point $t \geq t_0$, will remain above c.

Our proof of this fact in Isabelle is complicated and involves properties of suprema, continuity and derivatives. Therefore, we only show the formalisation of the statements below and their use in the proof of the correctness specification.

lemma *has-vderiv-mono-test*:
assumes *T-hyp*: *is-interval T*
and *d-hyp*: $D\ f = f'$ *on* T
and *xy-hyp*: $x{\in}T\ y{\in}T\ x \leq y$
shows $\forall\, x{\in}T.\ (0{::}real) \leq f'\ x \longrightarrow f\ x \leq f\ y$
and $\forall\, x{\in}T.\ f'\ x \leq 0 \longrightarrow f\ x \geq f\ y$
⟨*proof*⟩

lemma *current-vderiv-ge-always-ge*:
fixes *c*::*real*
assumes *init*: $c < x\ t_0$ **and** *ode*: $D\ x = x'$ *on* $\{t_0..\}$
and *dhyp*: $x' = (\lambda t.\ g\ (x\ t))\ \forall\, x{\geq}c.\ g\ x \geq 0$
shows $\forall\, t{\geq}t_0.\ x\ t > c$
⟨*proof*⟩

lemma $0 \leq t \longrightarrow \lceil \lambda s{::}real\hat{}2.\ s\$1\hat{}3{>}5 \wedge s\$2{>}2 \rceil \leq$
wp $(x' = (\lambda t\ s.\ (\chi\ i.\ if\ i{=}1\ then\ s\$1\hat{}3 + s\$1\hat{}4\ else\ 5 * s\$2 + s\$2\hat{}2))\ \&\ G)$
$\lceil \lambda s.\ s\$1\hat{}3{>}5 \wedge s\$2{>}2 \rceil$
apply(*rule diff-invariant-rules*, *simp-all add*: *diff-invariant-eq ivp-sols-def forall-2*; *clarsimp*)
apply(*frule-tac* $x{=}\lambda t.\ X\ t\ \$\ 1\ \hat{}\ 3$ **and**
$g{=}\lambda t.\ 3 * t\hat{}2 + 3 * (root\ 3\ t)\hat{}5$ **in** *current-vderiv-ge-always-ge*)
apply(*rule poly-derivatives*, *simp*, *assumption*, *simp*)
apply (*force simp*: *field-simps odd-real-root-power-cancel*, *force simp*: *add-nonneg-pos*, *force*)
apply(*frule-tac* $x{=}\lambda t.\ X\ t\ \$\ 2$ **in** *current-vderiv-ge-always-ge*)
by (*force*, *force*, *force simp*: *add-nonneg-pos*, *simp*)

The last proof unfolds the characterisation of invariance in its first line. Then it uses our recently formalised lemma on the function x^3 and its derivative. The emerging obligations go away with our derivative tactics and properties about real numbers. Then, the proof does the same procedure for the invariant $y > 2$.

Despite our generic result *current-vderiv-ge-always-ge*, we still need a similar one for non-strict inequalities. However, the proof method does not generalise to this case. Hence, this example reinforces one of the conclusions from the last benchmark problem. Namely, we still need general methods to prove invariance that complement Lemma 5.2.5 or generalise it. □

Example 7.6.6 (LICS: Example 4c relative safety of time-triggered car). Our final example from the competition is representative of the majority of the benchmark problems that we tackled. This is true even for those that were part of case studies. Specifically, for many of them, evolution commands involve the already explained constant acceleration dynamics. The challenge was in the structure of the hybrid program or in the clever use of formulas inside programs and vice versa. Particularly for this example, the verification problem is an implication of forward boxes —not the standard simple Hoare triple or its equivalent

formulations:

$$
\begin{array}{l}
(|x' = v, v' = -b]\, x \leq m) \wedge v \geq 0 \wedge A \geq 0 \wedge b > 0 \rightarrow \\
|\textbf{loop}\ (\\
\quad \big((?(2bm - x \geq v^2 + (A+b)(A\varepsilon^2 + 2\varepsilon * v))\,;\, a := A\big)\ +\!+\ a := -b\big)\,; \\
\quad t := 0\,;\, (x' = v, v' = a, t' = 1\ \&\ v \geq 0\ \&\ t \leq \varepsilon) \\
)\ \textbf{inv}\ v^2 \leq 2b(m - x) \\
]\, x \leq m,
\end{array}
$$

where we use ++ as nondeterministic choice to distinguish it from addition +.

As forward boxes are predicate transformers, asserting them in our semantic setting requires a state $s \in \mathbb{R}^V$. Therefore, the meaning of asserting $|\alpha]\, p$ in a specification as above corresponds to $\forall s.\ |F]\, P\, s$. Specifically in the relational model and without our notational simplifications, this translates to $\forall s.\ (s, s) \in |\mathcal{R}\, F]\lceil P\rceil_{\mathcal{R}}$ according to Section 3.2. Given that our rules for loops and invariants require inequalities $\leq$ instead of containment $\in$, for various benchmark problems we have to provide variants adapted to this notation. For instance, the *wlp* rule for loops with invariants shown below.

lemma *in-wp-loopI*:
I x $\Longrightarrow$ $\lceil I\rceil \subseteq \lceil Q\rceil$ $\Longrightarrow$ $\lceil I\rceil \subseteq$ *wp R* $\lceil I\rceil$ $\Longrightarrow$ *y = x* $\Longrightarrow$ (*x*,*y*) $\in$ *wp* (*LOOP R INV I*) $\lceil Q\rceil$
⟨*proof*⟩

Based on these observations, we formalise the verification problem in Isabelle as follows.

abbreviation *LICS-Ex4c-f* :: *real* ⇒ *real* ⇒ *real^4* ⇒ *real^4* (*f*)
where *f time acc s* ≡ (*χ i. if i=1 then s$2 else* (*if i=2 then acc else if i=3 then 0 else time*))

lemma *LICSexample4c-arith1*:
assumes $v^2 \leq 2 \cdot b \cdot (m - x)$ *0* $\leq t$ *A* $\geq$ *0 b* $>$ *0*
and *key*: $v^2 + (A \cdot (A \cdot \varepsilon^2 + 2 \cdot \varepsilon \cdot v) + b \cdot (A \cdot \varepsilon^2 + 2 \cdot \varepsilon \cdot v)) \leq 2 \cdot b \cdot (m - x)$
and *guard*: $\forall \tau.\ 0 \leq \tau \wedge \tau \leq t \longrightarrow (0{::}real) \leq A \cdot \tau + v \wedge \tau \leq \varepsilon$
shows $(A \cdot t + v)^2 \leq 2 \cdot b \cdot (m - (A \cdot t^2 \,/\, 2 + v \cdot t + x))$ (**is** - ≤ *?rhs*)
⟨*proof*⟩

lemma
assumes *A* ≥ *0 b* > *0 s$2* ≥ *0*
shows (*s*,*s*) ∈ *wp* (*x´=*(*f 0* (−*b*)) & (*λs. True*)) ⌈*λs. s$1* ≤ *m*⌉ ⟹
(*s*,*s*) ∈ *wp*
(*LOOP*
((⌈*λs. 2*b*(m−s$1)* ≥ *s$2^2+*(*A+b*)*(*A*ε^2+2*ε*(s$2))*⌉;(*3* ::= (*λs. A*)))
∪ (*3* ::= (*λs. −b*)));
(*4* ::= (*λs. 0*));
(*x´= *(*λs. f 1* (*s$3*) *s*) & (*λs. s$2* ≥ *0* ∧ *s$4* ≤ *ε*))
INV (*λs. s$2^2* ≤ *2*b*(m−s$1)*))) ⌈*λs. s$1* ≤ *m*⌉
apply(*subst* (*asm*) *local-flow.in-wp-g-ode-subset*[*OF local-flow-LICS-Ex4c-1*], *simp-all*)
apply(*rule in-wp-loopI*)

apply(*erule-tac x=s$2/b* **in** *allE*)
using ‹$b > 0$› ‹$s\$2 \geq 0$› **apply**(*simp add*: *field-simps power2-eq-square*, *simp*)
apply (*smt* ‹$b > 0$› *mult-sign-intros*(6) *sum-power2-ge-zero*)
apply(*simp add*: *rel-aka.fbox-add2*)
apply(*simp-all add*: *local-flow.wp-g-ode-subset*[*OF local-flow-LICS-Ex4c-2*], *safe*)
using *LICSexample4c-arith1*[*OF - - ‹0 ≤ A› ‹0 < b›*] **apply** *force*
by (*auto simp*: *field-simps power2-eq-square*)

Apart from the first line that calls our modified version for loop invariants, the proof is a standard manipulation of forward boxes until we are only left with arithmetical problems. The last two lines call the corresponding property about real numbers. In fact, this was a common issue among most problems. Isabelle lacks proof automation for basic arithmetical operations which forced us to prove these properties by hand and then formalise them as above. This was the most time demanding part of the problems and explains our need to supply these properties with an external tool like Mathematica.

In general, this problem exemplifies the remaining difficulties that we faced when solving the competition's problems. Certifying arithmetical proofs in Isabelle/HOL is time consuming and adapting notation from dL to our semantic setting requires new lemmas. Such a combination affected 8 of the unfinished benchmark verifications: we could not translate in time 2 of them from dL to our components, we finished 4 of them by asserting without proofs the corresponding arithmetical requirements, and we did not certify in time with an external tool the remaining 2. □

Summarising this section, the verification problems exhibited here make it clear that both our components and Isabelle/HOL require more proof automation. Specifically, Example 7.6.1 showed that we can provide the solution to the system of ODEs in some cases where dL requires its domain-specific rules. However, our components can still benefit from generic tactics that blast away the structure of every hybrid program. Isabelle's proof method language Eisbach [93] should make this task relatively easy, and our tactic *hyb-hoare* could be a starting point for this.

Example 7.6.2 shows how our verification components can reason in the style of dL if needed: they can use differential cuts, induction and weakenings as derived in Section 5.6. However, they would benefit from improvements to our tactics for proofs of invariance. This is further evidenced with Examples 7.6.4 and 7.6.5 that show that our list of theorems *diff-invariant-rules* corresponding to Lemma 5.2.5 is incomplete. Finding a complete list of conditions for proofs of invariance would be a valuable conceptual and technical contribution of future work.

Example 7.6.3 shows that our framework (including dH) can use transcendental functions, like exponentials or logarithms, that established tools cannot. Moreover, our tactics for certification of derivatives make this part of the verification process smooth. However, this example also evidences that two parts of the process must improve. One is the certification of Lipschitz continuity and Picard-Lindelöf's theorem in general. The other is the final proof obligations involving properties of real arithmetic which is also a recurring problem in most benchmarks. For Lipschitz continuity, extensions to the libraries of formalised mathematics are necessary. For the other, Isabelle requires generic tactics to prove properties about real numbers.

Finally, Example 7.6.6 simply is representative of most of the problems from the competition. Its only unique characteristic is its non-standard presentation of the verification problem. Although it was easily translated to our setting and proved there, we could not do this on time for other two problems from the competition. For this purpose, using a combination of Isabelle/UTP with lenses and our components might help to translate $\mathtt{d}\mathcal{L}$ formulas to our setting faster.

Despite all these discoveries, our participation in the competition was strong. The fact that we can solve so many different problems in the short span of a month with such recently developed verification components indicates how fast we can modify and extend them.

The interested reader can see a complete list of all 63 benchmark problems in Appendix A. The table only includes the name of the problem and whether we verified it or not. In case we did not, the table also adds a requirement to finish the proof. For a full formalisation of these problems with proofs in .thy files, see our online repository https://github.com/yonoteam/CPSVerification. The verifications are also in a proof document which resides in our repository. The problems cover 30 pages that correspond to more than 2000 lines of code.

7.7 Case Study: PID Control

The final example of the chapter is a first step towards verifying more complex hybrid systems with our components. It involves a widespread regulatory controller in industry: the *proportional integral derivative* (PID) controller. In fact, by 2000, 97% of industrial controllers already used PID feedback [34]. Therefore, verification of these systems has wide impact. In this section we explain basic notions behind PID controllers and formalise one that stabilises a quadcopter's flight [14]. Then, we verify simple properties about it with our components.

The decision making process of controllers often involves a desired value function or *set point* $\mathtt{SP}\,t$ for the parameter that they try to regulate. Similarly, every certain amount of time they register inside a *process variable* $\mathtt{PV}\,t$ the true value of the parameter. Then, controllers obtain the current error $e\,t = \mathtt{SP}\,t - \mathtt{PV}\,t$ for that parameter. Based on this, a *proportional* controller simply multiplies the error by a constant

$$u\,t = K_p(e\,t).$$

Here, the *actuation command* $u\,t$ tells the controller how much feedback it needs to apply. If $e\,t$ is large, then so is $u\,t$ and vice versa. The corresponding components connected to the controller that physically adjust the parameters according to $u\,t$ are *actuators*.

In some systems however, $u\,t = 0$ is problematic as input is still required from the controller to keep the system stable. This scenario can happen when the error is 0. Therefore, a simple proportional controller does not suffice. Thus, adding an *integral* term

$$u\,t = K_p(e\,t) + K_i \int_0^t (e\,\tau)d\tau$$

improves the actuation. This latter term collects previous positive and negative values of the error over time to help in the stabilisation.

Nevertheless, if an external agent tinkers with the system and impedes progress of the actuation, the integral term would accumulate the same error many times. This can lead to overshooting once the tinkerer stops, that is, the actuation could exceed the set point and take too much time to stabilise. Another risk is that proportional-integral (PI) controllers may never reach a steady state, remaining therefore in a constant oscillation. To correct and prevent for these scenarios, a *derivative* term

$$u\,t = K_p(e\,t) + K_i \int_0^t (e\,\tau)d\tau + K_d(e'\,t)$$

detects fast changes in the error and responds accordingly. If the rate of change of the error is large, the value of the derivative term is too. Thus, the sign of the constant K_d can balance the contribution of the other two terms. The addition of these three terms characterises PID controllers.

Hence, a discretisation of the actuation of a PID controller as pseudocode is the following

```
error_sum := 0
loop
  error := SP − PV
  error_sum := error_sum + error
  u := K_p ∗ error + K_i ∗ error_sum ∗ dt + K_d ∗ (error − prev_error)/dt
  prev_error := error
```

where dt indicates the amount of time it takes the controller to intervene. Thus, we assume that after an iteration of the loop, dt seconds pass before its body changes u again. In applications, engineers use various methods to determine the value of the constants K_p, K_i and K_d. They can decide those parameters before implementation by using simulations, or empirically by tuning them after the control is finished. Often, a combination of both approaches is necessary.

For our example, we consider a PID in charge of regulating the roll attitude of a quadcopter's flight. We take the equations for the dynamics from [14]'s Section 7.2. In more detail, a flying vehicle has three angles of motion: its pitch angle p for frontal diving or ascending, its yaw angle y for left and right turns in a horizontal plane, and its roll angle r for dives to the left or to the right. In [14], a simplification of the dynamics for the roll angle uses the second order ODE $r''\,t = T$, where T is the amount of rolling torque that the left and right motors of the quadrotor should produce. The corresponding vector field is our familiar constant acceleration vector field f_T from Example 5.2.1 and Sections 7.2 and 7.4.

We have explained everything we need about the PID for our Isabelle/HOL formalisation. Now, we describe it starting with the set of program variables. We split these in three: continuous variables, discrete variables and proof variables. The first category corresponds to the physical variables of the roll's angle r, its rate of chage v, and its torque T together with time t. The discrete variables include `error` and `error_sum`. Finally, we have added proof variables to help us in the description of the invariants of the system. These include the variables `roll` and `roll_rate` that model the controller's measurement of r and v respectively. Their purpose is just like that of h_0 for the water tank in Section 7.5 that measured the water

level h. We also add variable `veri_test` to indicate the cumulative value of `error_sum`, and variable `counter`, for the amount of times the loop has been executed. Differently from previous sections, here we use a finite set of strings as the set of program variables. This avoids using our memory to relate specific numbers with corresponding variables. Then we define a type based on said set. This is shown below.

abbreviation *kin-PI-strs* $\equiv$ {*''t''*, *''r''*, *''v''*, *''T''*, *''roll''*, *''error''*,
''roll-rate'', *''error-sum''*, *''veri-test''*, *''counter''*}

typedef *kin-PI-vars* = *kin-PI-strs*
morphisms *to-str to-var*
by *blast*

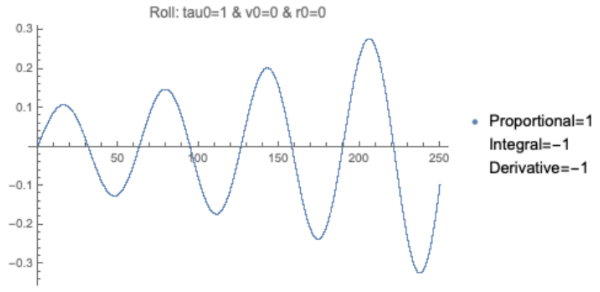
The vector field and flow are well-known to us. However, instead of the function $s\$i$ that queries the value at the ith position of vector s, we have to use its analogous $s\lfloor_V ''i''$ which gives the value of variable *''i''* at state s. The quotation marks *''i''* emphasise the string type.

abbreviation *kin-PI-vec-field* :: *real^kin-PI-vars* $\Rightarrow$ *real^kin-PI-vars* (f)
where $f\ s \equiv (\chi\ i.$ *if* $i = \lfloor_V ''t''$ *then 1 else*
(*if* $i = \lfloor_V ''r''$ *then* $s\lceil_V ''v''$ *else*
(*if* $i = \lfloor_V ''v''$ *then* $s\lceil_V ''T''$ *else 0*)))

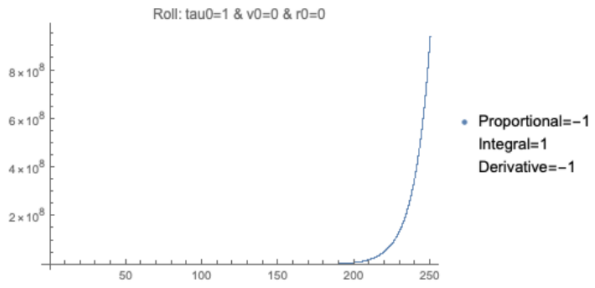
abbreviation *kin-PI-flow* :: *real* $\Rightarrow$ *real^kin-PI-vars* $\Rightarrow$ *real^kin-PI-vars* (φ)
where $\varphi\ t\ s \equiv$
($\chi\ i.$ *if* $i = \lfloor_V ''t''$ *then* $t + s\lceil_V ''t''$ *else*
(*if* $i = \lfloor_V ''r''$ *then* $(s\lceil_V ''T'')*t\hat{}2/2 + (s\lceil_V ''v'')*t + (s\lceil_V ''r'')$ *else*
(*if* $i = \lfloor_V ''v''$ *then* $(s\lceil_V ''T'')*t + (s\lceil_V ''v'')$ *else* $s\$i$)))

Based on this, below we present the formalisation of the hybrid program that combines the PID and its respective roll dynamics. As announced before, we use the difference between the current value of r and its previous value as the error for the PID. We also follow [14] in assigning the result of the PID actuator formula to the torque.

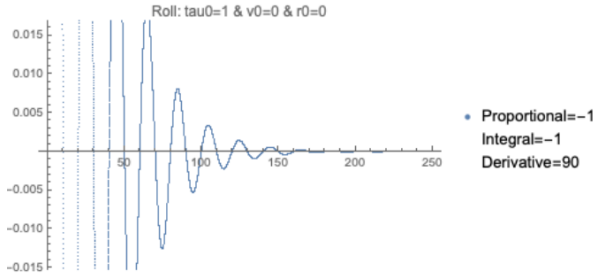
LOOP
(*IF* $(\lambda s.\ s\lceil_V ''t'' = dt)$ *THEN*
— CONTROL
$(\lfloor_V ''error'' ::= (\lambda s.\ s\lceil_V ''r'' - s\lceil_V ''roll''))$;
$(\lfloor_V ''error\text{-}sum'' ::= (\lambda s.\ s\lceil_V ''error\text{-}sum'' + s\lceil_V ''error''))$;
$(\lfloor_V ''T'' ::= (\lambda s.\ Prop * s\lceil_V ''error'' + Integr * dt * s\lceil_V ''error\text{-}sum''))$;
$(\lfloor_V ''roll'' ::= (\lambda s.\ s\lceil_V ''r''))$;
$(\lfloor_V ''roll\text{-}rate'' ::= (\lambda s.\ s\lceil_V ''v''))$;
$(\lfloor_V ''veri\text{-}test'' ::= (\lambda s.\ s\lceil_V ''veri\text{-}test'' + s\lceil_V ''error\text{-}sum''))$;
$(\lfloor_V ''counter'' ::= (\lambda s.\ s\lceil_V ''counter'' + 1))$;
$(\lfloor_V ''t'' ::= (\lambda s.\ 0))$
ELSE
— DYNAMICS
$(x' = f\ \&\ (\lambda s.\ s\lceil_V ''t'' \leq dt)))$



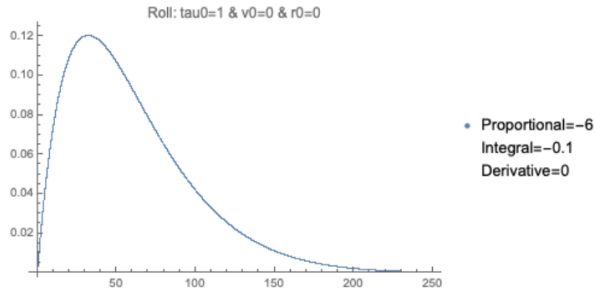
(a) If $K_p > 0$, then r diverges



(b) If $K_i \geq 0$, then r increases



(c) Higher values of K_d increase frecuency



(d) Chosen values for our PID

Figure 7.4: Various PID simulations

In practice, after each time the PID updates variable T, it passes the value of T to another controller (possibly a PID). Accordingly, this different controller uses that value to compute the desired voltage needed at a motor of the quadcopter. Nevertheless, as a simplification of our hybrid program above, we assume that this intermediate process is instantaneous relative to the action of our PID of interest.

As evidenced in previous examples, there is much work to do in terms of proof-automation to make Isabelle/HOL and our components capable of even certifying constraints for a complex interaction involving a PID controller. Instead, to obtain values for the constants K_p, K_i and K_d, we simulated the behaviour of the system in Mathematica. Then, we did a sensitivity analysis to see how variations in K_p, K_i and K_d affect the overall behaviour of $r\,t$. Figure 7.4 shows a small sample of the simulations we performed. Our objective was to find parameters that stabilise the roll angle r to 0. We discovered that, with non-negative values of K_p, r either oscillates or diverges from 0. Similarly, positive values of K_i make r an increasing function. Also, $|K_p|$ must be at least an order of magnitude greater than $|K_i|$ to obtain asymptotic and non-oscillating behaviour. Hence, we chose $K_p = -6$ and $K_i = -0.1$. Finally, for these values, the contribution of K_d was negligible except for positive values near 100 that generated an oscillating and diverging behaviour. Therefore, we set $K_d = 0$ for simplification of our analysis and opted to work with a PI controller instead.

As postconditions, we have verified simple invariants for this hybrid program. A first version starts with every value set to 0 except for the torque which begins at 1. The proof

then uses the following predicate as a postcondition and invariant.

$$(\lambda s.\ (s\lceil_V\!''counter'' \geq 1 \longrightarrow s\lceil_V\!''T'' = Prop * s\lceil_V\!''error'' + Integr * dt * s\lceil_V\!''error\text{-}sum'') \wedge \\ s\lceil_V\!''counter'' \in \mathbb{N} \wedge s\lceil_V\!''v'' = s\lceil_V\!''T'' * s\lceil_V\!''t'' + s\lceil_V\!''roll\text{-}rate'' \wedge \\ s\lceil_V\!''r'' = s\lceil_V\!''T'' * s\lceil_V\!''t''\hat{}2/2 + s\lceil_V\!''roll\text{-}rate'' * s\lceil_V\!''t'' + s\lceil_V\!''roll'')$$

Essentially, we show that after the first iteration, the value of $s\lceil_V\!''T$ is always given by the PID equation. Similarly, the counter is a natural number and the roll and its rate of change always satisfy the flow equations.

Our second verification uses as precondition the values of the variables after one iteration of the hybrid program's loop, and as postcondition, the invariant above. Yet, it adds to the latter a defining equation for variable `roll_rate` based on `veri_test` and `error_sum`.

$$(\lambda s.\ s\lceil_V\!''T'' = Prop * s\lceil_V\!''error'' + Integr * dt * s\lceil_V\!''error\text{-}sum'' \wedge \\ s\lceil_V\!''roll\text{-}rate'' = Prop * dt * (s\lceil_V\!''error\text{-}sum'' - s\lceil_V\!''error'') + \\ Integr * dt\hat{}2 * (s\lceil_V\!''veri\text{-}test'' - s\lceil_V\!''error\text{-}sum'') \wedge \\ s\lceil_V\!''counter'' \in \mathbb{N} \wedge s\lceil_V\!''v'' = s\lceil_V\!''T'' * s\lceil_V\!''t'' + s\lceil_V\!''roll\text{-}rate'' \wedge \\ s\lceil_V\!''r'' = s\lceil_V\!''T'' * s\lceil_V\!''t''\hat{}2/2 + s\lceil_V\!''roll\text{-}rate'' * s\lceil_V\!''t'' + s\lceil_V\!''roll'')$$

The proofs of both correctness specifications apply the same methods as in previous sections. The *wlp* rules blast away the program structure while the remaining proof obligations use arithmetical results.

The invariants provided with these verifications are useful properties to obtain further results of safety. Ideally, the characterisation of `roll_rate` in terms of `veri_test` should enable us to derive an analytical description of `roll` in terms of time that predicts the curve in Figure 7.4d. Then, this should allow us to formally verify that the roll angle does not deviate from 0 more that 0.15 units as the figure shows. Yet, further work is required in order to attain such a derivation. Nevertheless, our proofs here serve to evidence the capabilities of the components to analyse properties of more complicated systems than those presented in previous sections. We can regard these formalisations as a first step towards a methodology that integrates hybrid program verification with controller design.

7.8 Evaluation of the verification components

Up to this point, we have compared our components with other verification tools at the end of each section of this chapter and we have focused on concrete features like diversity of hybrid programs available in the tool, the various kinds of specifications that the tool can encode, the different methods for verification of evolution commands, the lengths of the proofs and the amount of proof-automation that the tool supports. We choose these features because they are the easiest to assess qualitatively. Broader notions like scalability or versatility would be harder to evaluate since they also depend on other factors like the age of the tool or the amount of people developing it. Therefore, in this section, we use the above concrete criteria to further evaluate our components and compare them with other frameworks. Specifically, we compare our components with the other two participants of the ARCH2020 friendly competition in the theorem proving category: the HHL prover and

KeYmaera X. The HHL prover is an important tool for deductive verification of hybrid systems embedded in Isabelle/HOL while KeYmaera X is the successor to the pioneering domain-specific KeYmaera prover.

Diversity of hybrid programs Despite their algebraic foundations based on KATs and MKAs, our verification components for hybrid programs initially had support only for while programs extended with evolution commands [72]. Nevertheless, throughout this thesis we have shown how easy it is to extend the hybrid program grammar in our setting as long as the new constructs have a state transformer (or relational) semantics. This is the reason why we could add to our components the Hoare rules in Section 3.2 about nondeterministic choices and tests in time for the ARCH2020 competition, or the program `open_door` of Section 7.3. This is also the reason why our components provide a variation of evolution commands unavailable in other tools: $\mathbf{evol}\,\varphi\,G\,U$. Yet, unlike differential game logic ($\mathsf{d}\mathcal{GL}$), our components do not support adversarial dynamics. The games of $\mathsf{d}\mathcal{GL}$ involve two players and extend hybrid programs with a dual operator α^d describing that the second player is executing α. The HHL prover does not have all the regular programs that $\mathsf{d}\mathcal{L}$ or our components support. Instead, it can use history formulas from the duration calculus [26] and message passing and parallel composition from communicating sequential processes [64]. Neither $\mathsf{d}\mathcal{GL}$ nor our components provide these features. We leave for future work the possible integration of these hybrid programs into our framework.

Variety of specifications Just like with hybrid programs, specifications are different among these tools. The HHL prover restricts its specifications to Hoare triples with an added duration calculus formula in the postcondition that talks about the history of the hybrid program [135]. Both $\mathsf{d}\mathcal{L}$ and our components can encode Hoare triples, however, neither include this added feature from the duration calculus. As before, our approach closely resembles that of KeYmaera X but not all versions of our components adequately cover the non-Hoare-like specifications of $\mathsf{d}\mathcal{L}$. As seen in Example 7.6.6, the components using relational semantics sometimes require stating that a certain $\mathsf{d}\mathcal{L}$-like predicate is inhabited. For instance, the statement $(s, s) \in |\alpha]\lceil P\rceil$ would represent the $\mathsf{d}\mathcal{L}$-assertion $|\alpha]\,P$. Alternatively, using the light-weight predicate transformer components, the same specification would be $|\alpha]\,P\,s$ which resembles more the forward box of $\mathsf{d}\mathcal{L}$. Moreover, even though our verification components can express various modalities (forward and backward) that $\mathsf{d}\mathcal{L}$ cannot, they are still prototypical. We neglected them for verification and decided to focus more on specifications that resembled the well-known Hoare triples. As a consequence, backward diamond and box operators do not occur in our light-weight verification components and the proof support for them in the MKA-based components is limited to the properties available from the Kleene algebra libraries. We leave for future work the optimisation of rules for verification condition generation for all our modal operators. Apart from the backward modalities, $\mathsf{d}\mathcal{L}$ also lacks support for expressing quantification over predicates as it is a first-order logic, while our components reside in Isabelle's higher order setting. Yet, the recent constructive variant of $\mathsf{d}\mathcal{GL}$ models formulas as predicates in a type theory [21] which could make KeYmaera X handle these cases in a near future.

Verification methods for evolution commands The HHL prover focuses in a single approach for evolution commands: it calls an external invariant generator and it certifies that it is an invariant [135]. In comparison, our components are still not connected to an external invariant generator but they include simple tactics for their certification. KeYmaera X is also connected to the Mathematica based tool Pegasus [127] that supports invariant generation and the certification can be done quickly with dℒ's rules as described in Example 7.6.2. Users can supply the solution to the system of ODEs in both our components and KeYmaera X. However, the restriction to first-order real arithmetic in KeYmaera X limits its ability to express complex solutions like those provided with our formalisation of affine systems of ODEs. Moreover, just like in Example 7.6.5, users can always unfold the definition of invariance and use the tools from mathematical analysis available in Isabelle/HOL.

Proof lengths In the case of the HHL prover, the proofs in its online repository [136] are long. Approximately 200 lines of code correspond to a single verification but this is a choice of style for Isabelle proofs. Common Isabelle tactics like *simp-all* and *intro* could reduce the length of the proofs in the repository. In our setting, verifications with dynamics directly written in specifications are evidently shorter than those where flows need to be supplied. This is because we also need to show that the conditions of *local-flow* hold. In comparison, simple invariants tend to require even shorter proofs than those with flows or annotated dynamics. However, if the invariant is a conjunction of various assertions, the proof quickly becomes longer as in Example 7.6.5. The prover KeYmaera X can solve many of the ARCH2020 competition problems automatically. In the scripted format, our multi-line dℒ-style proofs correspond to one line in KeYmaera X with just two tactics [98].

Proof-automation This criteria is directly related with length of proofs. Therefore, because of the style of proofs chosen with the HHL prover, we limit our discussion for this prover by simply stating that benefits to Isabelle/HOL would also help to automate proofs for it and for our components. However, despite the prototypical state of our various verification components, we have supplied some proof automation for them. Our tactics for derivatives certify various polynomial and transcendental expressions quickly. Similarly, our tactics for invariants can handle equalities, conjunctions and disjunctions automatically while inequalities require user-input. Finally, our Eisbach tactics for Hoare logic and the refinement calculus blast away the program structure for our KAT-based components while a simplification with *wlp* rules does the same for our MKA and predicate transformer components. In contrast, KeYmaera X's `ODE` tactic automatically discharges some proof obligations with a single evolution command. This does more than certifying derivations. For some proofs where the differential ghosts rule is required, KeYmaera X provides the tactic `dbx` that quickly applies the ghosts rule and the corresponding invariant reasoning with cuts and weakenings. Finally, it also provides a `master` tactic that blasts away the program structure and discharges some arithmetical facts. As explained before, discharging real-arithmetical facts in Isabelle/HOL is left for future work.

The tools discussed in this section have various strengths relative to the others. In general, we can observe that while the Isabelle-based tools are more expressive, they are still lacking in proof automation in comparison to KeYmaera X. This is a consequence of the fact that the domain specific prover has accumulated a lot of features that would also benefit the other

two provers. Furthermore, various versions of $\mathsf{d}\mathcal{L}$ compensate for this lack of expressiveness. However, the openness and modularity of the framework presented in this thesis should help in the fast development of related or alternative verification components that include these features.

Except for the benchmark problems of the ARCH2020 competition and the PID of the quadcopter, we have formalised all the examples in this chapter with the various versions of the components depicted in Figure 6.1. For time constraints, the examples from the ARCH2020 competition and the PID only occur in the relational MKA components and in our light-weight predicate transformer components. All these verifications are available in the Archive of Formal Proofs [68, 70] or in our online repository https://github.com/yonoteam/CPSVerification. Like we explained in Section 6.6, the verification examples throughout this chapter for the state transformer and the relational semantics require exactly the same code. This further evidences that our approach for building verification components is modular. For the rest of the components, the proofs are similar up to renaming of lemmas and changes in notation.

Chapter 8

Conclusions

In this thesis, we presented an open, modular semantic framework for deductively verifying hybrid systems in a general-purpose proof assistant. We carried out the concrete implementation of this framework as various verification components in the interactive theorem prover Isabelle/HOL. We also showed the approach at work by solving verification problems and case studies. This is the first time that such an algebraic based component serves to reason about hybrid programs.

The fact that the resulting components are part of a proof assistant, together with the fact that they serve for deductive reasoning, makes them capable of expressing more complex continuous dynamics than established methods. Another benefit from this choice is that the soundness of the rules for verification condition generation is certified during their development. This makes the components correct by construction relative to the trust on the proof assistant. Moreover, because of their openness and their implementation in the well-established Isabelle/HOL, the user base of the proof assistant can contribute and improve them more rapidly than other domain-specific tools. The components also benefit from Isabelle's huge and impressive libraries of formalised mathematics and various proof automation tactics.

8.1 Summary

Figure 1.1 of the introduction and Figure 6.1 of Chapter 6 depict the general steps for implementing this framework. Initially, the formalisation of an algebraic structure that models predicate transformers provides us with laws for computation of weakest liberal preconditions. Alternatively, the algebraic structure can subsume the rules of Hoare logic. Then, the framework requires a shallow embedding of hybrid programs via an instance of the underlying algebra. Finally, we select a representation of the program store that enables us to complete the construction of hybrid programs and the derivation of verification rules.

Figure 6.1 also depicts the modularity of the approach. At every step of the implementation, there are alternative design choices with options beyond what we have shown in this thesis and that are yet to be explored. Firstly, we can choose the underlying algebraic structure. In this work, we have used Kleene algebras with tests, modal Kleene algebras, and three versions of predicate transformers: from the powerset monad, transformers based on

quantales, and a direct encoding. We can instantiate each one of these algebraic entities to an intermediate semantics. We have focused on relational and state transformer semantics in this thesis. Finally, we can choose a concrete model for the program store. Throughout this work we have mostly used functions from a finite set of variables to the real numbers, but we have also dedicated a section to discuss the algebraic model of lenses. In [71], we even used functions from the infinite type of strings to the real numbers.

The framework not only offers modularity, but also extensibility for the components. We can do so in many ways. Firstly, as the embedding is shallow, we can use the model for programs to quickly define more variants of hybrid programs. Examples of this are the nondeterministic assignment of Section 5.6, the `open_door` of Section 7.3 and our definition of evolution commands based on analytical descriptions of the dynamics of the system. The same applies at the level of predicate transformers: apart from forward or backward box and diamond operators, we can define others functions of type $\mathbb{B}^S \to \mathbb{B}^S$ to interact with them. We can also formalise various subclasses of systems of differential equations and derive their corresponding properties as exemplified with affine systems of ODEs in Section 6.5. These extensions are only restricted by what users can develop within higher-order logic.

Finally, the verification examples of Chapter 7 show that the ending result works. That is, users can utilise the emerging verification components to successfully analyse complex hybrid programs, albeit with added tactics to automate the verification condition generation.

8.2 Future work

Other researchers have used our framework. For instance, in [43], the authors successfully take advantage of the extensibility of the approach and use lenses to provide alternative matrix representations of systems of ODEs. Then, they use their implementation to verify some avoidance tactics of an autonomous marine vehicle. We plan to collaborate with them and supply our framework as a basis for the development of a tool for verification of hybrid systems soon. Furthermore, successor work after this thesis could include the derivation and proof of safety specifications for more controllers involved in the stabilisation of the quadcopter's flight.

On the algebraic side of the components, future work might involve exploring total correctness and rely-guarantee methods for concurrent systems. For instance, in [54,55], the verification components for regular programs already include a section for total correctness. The underlying structure consists of divergence Kleene algebras that include a function $\nabla : K \to K$ where $\nabla\, \alpha$ models those states where α does not necessarily terminate. Furthermore, this approach is already available in Isabelle/HOL. Extending its methods for infinite iterations and evolution commands is worth considering. In particular, a total correctness setting might provide alternative manipulations of the evolution commands based on guarded orbitals independent of uniqueness results. On the other hand, in [7], the authors use concurrent Kleene algebras for formal verifications of rely-guarantee specifications in Isabelle/HOL. As explained in the verification of the PID, many controllers operate in parallel within an engineered system. This means that rely-guarantee methods could complement the verification style described in this thesis.

To evidence the generality of our approach, future work could also focus on absorbing

other well-established calculi for verification of hybrid systems. That is, like we did in Section 5.6, incorporation of rules of inference and programming constructs from other logics is possible. For instance, we could explore the possibility of encoding with our components the denotational semantics of the calculus of hybrid communicating sequential processes (HCSP) [135]. This proof system can also derive a hybrid Hoare logic for verification of hybrid systems. If we could derive the rules of inference of that calculus, it would strengthen even more the relevance of our framework.

Similarly, there are many variations of $\mathsf{d}\mathcal{L}$ that we could try to encode. For instance, differential game logic [111] serves to study not only discrete and continuous interactions, but also adversarial dynamics in the form of games. It uses hybrid games, whose semantics are all functions of type $\mathcal{P}\,S \to \mathcal{P}\,S$ where S is the set of game states. With our semantic framework, we should be able to replicate this approach as smoothly as the rest of the constructions depicted in previous chapters. Other interesting alternatives include quantified differential dynamic logic for reasoning about hybrid distributed systems [110] and stochastic differential dynamic logic for stochastic hybrid systems [109].

Our framework also provides the perfect place to explore interesting conceptual ideas about dynamical systems. For example, the evolution commands of Section 5.3—where we directly encode the dynamics in the specification—allow us to use discrete dynamical systems as hybrid programs. This could be helpful to encode with our components well-known discrete systems such as traces of type $\varphi : \mathbb{N} \to S \to S$ or finite automata. In turn, this could become an alternative verification style for simulated hybrid systems. That is, we could verify safety properties about traces generated by simulations of hybrid systems. Additionally, these innovative evolution commands could serve to extend our methods for the analysis of continuous dynamical systems by integrating the Poincaré maps of [76].

In the case of differential equations, a lot of work remains. For starters, the guards of evolution commands are the ideal place to add an algebraic equation. These would lead to a study of differential-algebraic systems of equations with our components [57]. Another important addendum considers that so far our approach has been restricted to ordinary differential equations. Nevertheless many systems in nature are better represented with partial differential equations [78]. An extension of the components in such direction is not only interesting but a relevant mathematical formalisation per se. It could open the way to new formalisations of physical theories like quantum mechanics in proof assistants.

Another very important path for future work involves tool extensions. In [74, 75], the authors formalise Runge-Kutta methods in Isabelle/HOL. These and other numerical methods are the de facto choice for physicists and engineers whenever finding solutions or invariants is difficult [63]. It remains to be seen if integrating their formalisation with our verification components is viable. Nevertheless some extension of certified numerical methods to our framework would make it more relevant for verification.

Another extension involves our formalisation of affine systems of ODEs. It is one of many linear algebra developments in the Archive of Formal Proofs, each of which tackles different properties of matrices and linearity. For example, in [38,132] the focus is on obtaining Jordan Normal Forms. These have been combined with a list representation of vectors to generate executable code in Standard ML or Haskell [37,38]. Considering our components with lenses in Isabelle/UTP and executable euclidean spaces discussed in Section 6.4, the integration of these theories with ours towards the generation of verified code is another research option.

Furthermore, this integration would allow us to reason beyond diagonalisable systems. All of this could also be complemented by formalising the general solution for the time-dependent versions of affine systems of ODE that uses resolvent matrices.

Finally, proof automation is always a pursuable endeavour. In particular, we could address the limitations of our components described in Section 7.6. For instance, although we showed that we can provide semantic proofs for the verification problems where differential ghosts are needed in $\mathsf{d}\mathcal{L}$, a generic method is still missing. In particular, we pointed out that we could improve the tactics for invariance of systems of ODEs by finding conditions for various cases involving it. This path is not only mathematically interesting but would also help in the automation of verification condition generation. Another weakness of the components involved the more time-consuming parts: formalising facts of arithmetic with real numbers. In fact, in Appendix A, we do not assign the simple "verified" status to 10 of the 63 benchmark problems that we did for the ARCH2020 competition. From those 10, we require better proof automation for reasoning with real numbers in 7 of them. Just like our tactic *poly-derivatives* that automatically certifies derivatives, we could develop tactics that certify basic factorisations, algebraic manipulations and operations with transcendental functions. This would greatly benefit the verification procedure. Lastly, at many points of the thesis, we considered a Sledgehammer approach [95, 106] where a computer algebra system supplies the solution to a problem while Isabelle certifies it. This is particularly necessary for providing solutions, generating invariants, diagonalising matrices and finding Lipschitz constants. In our work, because of the short span of a PhD project, we have preferred to focus on more conceptual contributions rather than these technical extensions to the proof assistant. Yet, now that the framework is robust, proof automation remains a crucial task that would strengthen the approach.

In the end, just like nowadays formal methods for software support other well-established approaches in industry [16], deductive verification of hybrid systems should complement the traditional techniques of testing and simulation. Our work in this thesis represents a small step in that direction.

Bibliography

[1] E. Ábrahám-Mumm, M. Steffen, and U. Hannemann. Verification of hybrid systems: Formalization and proof rules in PVS. In *ICECCS 2001*, pages 48–57. IEEE Computer Society, 2001.

[2] M. Althoff. An introduction to CORA 2015. In *ARCH 2015*, pages 120–151. EasyChair, 2015.

[3] R. Alur. Formal verification of hybrid systems. In *EMSOFT 2011*, pages 273–278. ACM, 2011.

[4] R. Alur. *Principles of Cyber-Physical Systems*. The MIT Press, 2015.

[5] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

[6] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In J. Jouannaud and Z. Shao, editors, *CPP 2011*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

[7] A. Armstrong, V. B. F. Gomes, and G. Struth. Algebraic principles for rely-guarantee style concurrency verification tools. In *FM 2014*, volume 8442 of *LNCS*, pages 78–93. Springer, 2014.

[8] A. Armstrong, V. B. F. Gomes, and G. Struth. Kleene algebra with tests and demonic refinement algebras. *Archive of Formal Proofs*, 2014.

[9] A. Armstrong, V. B. F. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2016.

[10] A. Armstrong, G. Struth, and T. Weber. Kleene algebra. *Archive of Formal Proofs*, 2013.

[11] V. I. Arnol'd. *Ordinary Differential Equations*. Springer, 1992.

[12] R. Back and J. von Wright. *Refinement Calculus—A Systematic Introduction*. Springer, 1998.

[13] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV 2011*, pages 171–177, 2011.

[14] R. W. Beard. Quadrotor dynamics and control. Technical report, Brigham Young University, 2008.

[15] S. Bennett. A brief history of automatic control. *IEEE Control Systems Magazine*, 16(3):17–25, 1996.

[16] D. Beyer and T. Lemberger. Software verification: Testing vs. model checking - A comparative evaluation of the state of the art. In O. Strichman and R. Tzoref-Brill, editors, *HVC 2017*, volume 10629 of *LNCS*, pages 99–114. Springer, 2017.

[17] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003.

[18] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[19] L. Blaauwbroek, J. Urban, and H. Geuvers. The tactician - A seamless, interactive tactic learner and prover for Coq. In C. Benzmüller and B. R. Miller, editors, *CICM 2020*, volume 12236 of *LNCS*, pages 271–277. Springer, 2020.

[20] B. Bohrer, N. Fulton, S. Mitsch, A. Platzer, J.-D. Quesel, and M. Völp. KeYmaera X: Cheat sheet, 2020. http://www.ls.cs.cmu.edu/KeYmaeraX/KeYmaeraX-sheet.pdf, Accessed: 15-9-2020.

[21] B. Bohrer and A. Platzer. Constructive hybrid games. In N. Peltier and V. Sofronie-Stokkermans, editors, *IJCAR 2020*, volume 12166 of *LNCS*, pages 454–473. Springer, 2020.

[22] B. Bohrer, V. Rahli, I. Vukotic, M. Völp, and A. Platzer. Formally verified differential dynamic logic. In *CPP 2017*, pages 208–221. ACM, 2017.

[23] S. Boldo, C. Lelay, and G. Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *MCS*, 9(1):41–62, 2015.

[24] L. Bu, Y. Li, L. Wang, X. Chen, and X. Li. BACH 2 : Bounded reachability checker for compositional linear hybrid systems. In *DATE 2010*, pages 1512–1517. IEEE Computer Society, 2010.

[25] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[26] Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Inf. Process. Lett.*, 40(5):269–276, 1991.

[27] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *CAV 2013*, volume 8044 of *LNCS*, pages 258–263. Springer, 2013.

[28] A. Chlipala. *Certified Programming with Dependent Types—A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[29] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logics of Programs, 1981*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[30] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2001.

[31] L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-corn, the constructive Coq repository at nijmegen. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *MKM 2004*, volume 3119 of *LNCS*, pages 88–103. Springer, 2004.

[32] L. Czajka and C. Kaliszyk. Hammer for Coq: Automation for dependent type theory. *JAR 2018*, 61(1-4):423–453, 2018.

[33] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS 2008*, pages 337–340, 2008.

[34] L. Desborough and Y. Miller. Increasing customer value of industrial control performance monitoring? honeywell experience. In *6th International Conference Chemical Process Control, AIChE Symp., Series 326*, 2002.

[35] J. Desharnais and G. Struth. Internal axioms for domain semirings. *Science of Computer Programming*, 76(3):181–203, 2011.

[36] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[37] J. Divasón and J. Aransay. Gauss-Jordan algorithm and its applications. *Archive of Formal Proofs*, 2014.

[38] J. Divasón, O. Kunčar, R. Thiemann, and A. Yamada. Perron-Frobenius theorem for spectral radius analysis. *Archive of Formal Proofs*, 2016.

[39] L. Doyen, G. Frehse, G. J. Pappas, and A. Platzer. Verification of hybrid systems. In *Handbook of Model Checking.*, pages 1047–1110. Springer, 2018.

[40] A. Eggers, N. Ramdani, N. S. Nedialkov, and M. Fränzle. Improving the SAT modulo ODE approach to hybrid systems analysis by combining different enclosure methods. *Software and Systems Modeling*, 14(1):121–148, 2015.

[41] G. E. Fainekos, H. Kress-Gazit, and G. J. Pappas. Hybrid controllers for path planning: A temporal logic approach. In *IEEE Conference on Decision and Control*, pages 4885–4890, 2005.

[42] S. Foster, J. Baxter, A. Cavalcanti, A. Miyazawa, and J. Woodcock. Automating verification of state machines with reactive designs and Isabelle/UTP. *CoRR*, abs/1807.08588, 2018.

[43] S. Foster, M. Gleirscher, and R. Calinescu. Towards deductive verification of control algorithms for autonomous marine vehicles. *CoRR*, abs/2006.09233, 2020. arXiv:2006.09233 [cs.LO].

[44] S. Foster, J. J. Huerta y Munive, and G. Struth. Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In *RAMiCS 2020[postponed]*, pages 169–186, 2020.

[45] S. Foster and F. Zeyda. Optics in Isabelle/HOL. *Archive of Formal Proofs*, 2018.

[46] S. Foster, F. Zeyda, Y. Nemouchi, P. Ribeiro, and B. Wolff. Isabelle/UTP: Mechanised Theory Engineering for Unifying Theories of Programming. *Archive of Formal Proofs*, 2019.

[47] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *ICTAC 2016*, volume 9965 of *LNCS*, pages 295–314, 2016.

[48] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *CAV 2011*, volume 6806 of *LNCS*, pages 379–395. Springer, 2011.

[49] B. Friedland and S. W. Director. *Control Systems Design: An Introduction to State-Space Methods*. McGraw-Hill Higher Education, 1985.

[50] N. Fulton, S. Mitsch, J. Quesel, M. Völp, and A. Platzer. KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In *CADE-25*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015.

[51] S. Gao, S. Kong, and E. M. Clarke. dreal: An SMT solver for nonlinear theories over the reals. In *CADE-24*, pages 208–214, 2013.

[52] J. Gibbons and N. Wu. Folding domain-specific languages: deep and shallow embeddings. In *19th ACM SIGPLAN ICFP 2014*, pages 339–347, 2014.

[53] G. Gierz, K. H. Hofmann, J. D. Lawson, M. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices*. Springer, 1980.

[54] V. B. F. Gomes, W. Guttman, P. Höfner, G. Struth, and T. Weber. Kleene algebra with domain. *Archive of Formal Proofs*, 2016.

[55] V. B. F. Gomes and G. Struth. Modal Kleene algebra applied to program correctness. In *FM 2016*, volume 9995 of *LNCS*, pages 310–325, 2016.

[56] V. B. F. Gomes and G. Struth. Program construction and verification components based on Kleene algebra. *Archive of Formal Proofs*, 2016.

[57] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer, 1996.

[58] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[59] J. Harrison, J. Urban, and F. Wiedijk. History of interactive theorem proving. In *Computational Logic*, pages 135–214. 2014.

[60] T. A. Henzinger. Sooner is safer than later. *Inf. Process. Lett.*, 43(3):135–141, 1992.

[61] T. A. Henzinger. The theory of hybrid automata. In *LICS 1996*, pages 278–292. IEEE Computer Society, 1996.

[62] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *J. Comput. Syst. Sci.*, 57(1):94–124, 1998.

[63] M. W. Hirsch, S. Smale, and R. L. Devaney. *Differential equations, dynamical systems, and linear algebra*. Academic Press, 1974.

[64] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[65] C. A. R. Hoare. An axiomatic basis for computer programming (reprint). In *Software Pioneers*, pages 367–383. Springer Berlin Heidelberg, 2002.

[66] P. Höfner and B. Möller. An algebra of hybrid systems. *J. Logic and Algebraic Programming*, 78(2):74–97, 2009.

[67] J. Hölzl, F. Immler, and B. Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In *ITP 2013*, volume 7998 of *LNCS*, pages 279–294. Springer, 2013.

[68] J. J. Huerta y Munive. Verification components for hybrid systems. *Archive of Formal Proofs*, 2019.

[69] J. J. Huerta y Munive. Affine systems of ODEs in Isabelle/HOL for hybrid-program verification. In *SEFM 2020*, volume 12310 of *LNCS*, pages 77–92. Springer, 2020.

[70] J. J. Huerta y Munive. Matrices for ODEs. *Archive of Formal Proofs*, 2020.

[71] J. J. Huerta y Munive and G. Struth. Verifying hybrid systems with modal Kleene algebra. In *RAMiCS 2018*, volume 11194 of *LNCS*, pages 225–243. Springer, 2018.

[72] J. J. Huerta y Munive and G. Struth. Predicate transformer semantics for hybrid systems: Verification components for Isabelle/HOL. arXiv:1909.05618 [cs.LO], 2019.

[73] F. Immler and J. Hölzl. Numerical analysis of ordinary differential equations in Isabelle/HOL. In *ITP 2012*, volume 7406 of *LNCS*, pages 377–392. Springer, 2012.

[74] F. Immler and J. Hölzl. Ordinary differential equations. *Archive of Formal Proofs*, 2012.

[75] F. Immler and C. Traut. The flow of ODEs. In *ITP 2016*, volume 9807 of *LNCS*, pages 184–199. Springer, 2016.

[76] F. Immler and C. Traut. The flow of ODEs: Formalization of variational equation and Poincaré map. *J. Automated Reasoning*, 62(2):215–236, 2019.

[77] J. Jeannin, K. Ghorbal, Y. Kouskoulas, A. Schmidt, R. Gardner, S. Mitsch, and A. Platzer. A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. *STTT*, 19(6):717–741, 2017.

[78] F. John. *Partial Differential Equations*. Springer, 1986.

[79] B. Jónsson and A. Tarski. Boolean algebras with operators, Part I. *Americal Journal of Mathematics*, 73(4):207–215, 1951.

[80] C. Kaliszyk and J. Urban. Hol(y)hammer: Online ATP service for HOL light. *MCS*, 9(1):5–22, 2015.

[81] Y. Kouskoulas, D. W. Renshaw, A. Platzer, and P. Kazanzides. Certifying the safe design of a virtual fixture control algorithm for a surgical robot. In *HSCC 2013*, pages 263–272, 2013.

[82] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. Comput.*, 110(2):366–390, 1994.

[83] D. Kozen. Kleene algebra with tests. *ACM TOPLAS*, 19(3):427–443, 1997.

[84] D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM TOCL*, 1(1):60–76, 2000.

[85] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[86] J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou. A calculus for hybrid CSP. In *APLAS 2010*, volume 6461 of *LNCS*, pages 1–15. Springer, 2010.

[87] J. Liu, N. Zhan, and H. Zhao. Computing semi-algebraic invariants for polynomial dynamical systems. In S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, editors, *EMSOFT 2011*, pages 97–106. ACM, 2011.

[88] S. M. Loos and A. Platzer. Differential refinement logic. In *LICS 2016*, pages 505–514. ACM, 2016.

[89] S. M. Loos, A. Platzer, and L. Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In *FM 2011*, volume 6664 of *LNCS*, pages 42–56. Springer, 2011.

[90] S. MacLane. *Categories for the Working Mathematician*. Springer, 1971.

[91] E. Makarov and B. Spitters. The picard algorithm for ordinary differential equations in Coq. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 463–468. Springer, 2013.

[92] E. G. Manes. *Predicate Transformer Semantics*. Cambridge University Press, 1992.

[93] D. Matichuk, T. C. Murray, and M. Wenzel. Eisbach: A proof method language for Isabelle. *J. Automated Reasoning*, 56(3):261–282, 2016.

[94] J. C. Maxwell. On Governors. In *Proceedings of the Royal Society of London*, pages 270–283. JSTOR, 1868.

[95] J. Meng, C. Quigley, and L. C. Paulson. Automation for interactive proof: First prototype. *Inf. Comput.*, 204(10):1575–1596, 2006.

[96] S. Mitsch, K. Ghorbal, and A. Platzer. On provably safe obstacle avoidance for autonomous robotic ground vehicles. In *Robotics: Science and Systems*, 2013.

[97] S. Mitsch, J. J. Huerta y Munive, X. Jin, B. Zhan, S. Wang, and N. Zhan. ARCH-COMP20 category report: Hybrid systems theorem proving. In *ARCH20.*, pages 141–161, 2019.

[98] S. Mitsch and A. Platzer. Benchmarks. commit 0e8372e01ac7d7dcdc2bbe730d31a9a6810c1748, 2020. https://github.com/LS-Lab/KeYmaeraX-projects/tree/master/benchmarks.

[99] C. Morgan. *Programming from specifications, 2nd Edition*. Prentice Hall, 1994.

[100] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *ICFP 2008*, pages 229–240, 2008.

[101] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[102] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *CADE 11*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.

[103] P. K. Pandya, H. Wang, and Q. Xu. Toward a theory of sequential hybrid programs. In D. Gries and W. P. de Roever, editors, *PROCOMET '98*, volume 125 of *IFIP Conference Proceedings*, pages 366–384. Chapman & Hall, 1998.

[104] L. C. Paulson. Natural deduction as higher-order resolution. *J. Log. Program.*, 3(3):237–258, 1986.

[105] L. C. Paulson. Isabelle: The next 700 theorem provers. *CoRR*, cs.LO/9301106, 1993.

[106] L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.

[107] A. Platzer. The structure of differential invariants and differential cut elimination. *LMCS*, 8(4), 2008.

[108] A. Platzer. *Logical Analysis of Hybrid Systems*. Springer, 2010.

[109] A. Platzer. Stochastic differential dynamic logic for stochastic hybrid programs. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 446–460. Springer, 2011.

[110] A. Platzer. A complete axiomatization of quantified differential dynamic logic for distributed hybrid systems. *Logical Methods in Computer Science*, 8(4):1–44, 2012.

[111] A. Platzer. Differential game logic. *ACM TOCL*, 17(1):1:1–1:52, 2015.

[112] A. Platzer. A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reason.*, 59(2):219–265, 2017.

[113] A. Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.

[114] A. Platzer. Differential game logic. *Archive of Formal Proofs*, 2019, 2019.

[115] A. Platzer and E. M. Clarke. Computing differential invariants of hybrid systems as fixedpoints. *Formal Methods Syst. Des.*, 35(1):98–120, 2009.

[116] A. Platzer and J. Quesel. European train control system: A case study in formal verification. In *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 246–265. Springer, 2009.

[117] A. Platzer and Y. K. Tan. Differential equation axiomatization: The impressive power of differential ghosts. In *LICS*, pages 819–828. ACM, 2018.

[118] A. Pnueli. The temporal logic of programs. In *18th ASoFoCS, 1977*, pages 46–57. IEEE Computer Society, 1977.

[119] V. Preoteasa. Algebra of monotonic boolean transformers. In *SBMF 2011*, volume 7021 of *LNCS*, pages 140–155. Springer, 2011.

[120] V. Preoteasa. Algebra of monotonic boolean transformers. *Archive of Formal Proofs*, 2011.

[121] A. N. Prior. *Time and Modality*. Clarendon Press, 2003.

[122] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

[123] D. Ricketts, G. Malecha, M. M. Alvarez, V. Gowda, and S. Lerner. Towards verification of hybrid systems in a foundational proof assistant. In *MEMOCODE 2015*, pages 248–257. IEEE, 2015.

[124] D. Rouhling. A formal proof in Coq of a control function for the inverted pendulum. In J. Andronick and A. P. Felty, editors, *CPP 2018*, pages 28–41. ACM, 2018.

[125] N. Schirmer. *Verification of sequential imperative programs in Isabelle-HOL*. PhD thesis, Technical University Munich, Germany, 2006.

[126] J. T. Slagel, L. White, and A. Dutle. Formal verification of semi-algebraic sets and real analytic functions. In C. Hritcu and A. Popescu, editors, *CPP 21*, pages 278–290. ACM, 2021.

[127] A. Sogokon, K. Ghorbal, P. B. Jackson, and A. Platzer. A method for invariant generation for polynomial continuous systems. In *VMCAI 2016*, pages 268–288, 2016.

[128] G. Struth. On the expressive power of Kleene algebra with domain. *Information Processing Letters*, 116(4):284–288, 2016.

[129] G. Struth. Quantales. *Archive of Formal Proofs*, 2018.

[130] G. Struth. Transformer semantics. *Archive of Formal Proofs*, 2018.

[131] G. Teschl. *Ordinary Differential Equations and Dynamical Systems*. AMS, 2012.

[132] R. Thiemann and A. Yamada. Matrices, Jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, 2015.

[133] H. L. Trentelman, A. A. Stoorvogel, and M. Hautus. *Control Theory for Linear Systems*. Communications and Control Engineering. Springer Verlag, 2001.

[134] S. Wang, N. Zhan, and D. P. Guelev. An assume/guarantee based compositional calculus for hybrid CSP. In M. Agrawal, S. B. Cooper, and A. Li, editors, *TAMC 2012*, volume 7287 of *LNCS*, pages 72–83. Springer, 2012.

[135] S. Wang, N. Zhan, and L. Zou. An improved HHL prover: An interactive theorem prover for hybrid systems. In *ICFEM 2015*, pages 382–399, 2015.

[136] S. Wang, N. Zhan, and L. Zou. HHL prover. commit c0b19867aef72275316aeff37ac7dd2f4dc66d24, 2018. https://github.com/wangslyl/hhlprover.

[137] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *CADE-22 2009*, pages 140–145, 2009.

[138] L. Zou, J. Lv, S. Wang, N. Zhan, T. Tang, L. Yuan, and Y. Liu. Verifying chinese train control system under a combined scenario by theorem proving. In E. Cohen and A. Rybalchenko, editors, *VSTTE 2013*, volume 8164 of *LNCS*, pages 262–280. Springer, 2013.

Appendices

Appendix A

ARCH2020 Problems

The table below lists the names and status of the 63 problems that we tackled for the Theorem Proving category of the friendly competition of the 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH2020) [97]. The first 60 correspond to the 60 design shape benchmark problems, while the last three are part of a case study. Overall, we have given the simple "verified" status to 53 of them. Three more are verified with a minor caveat and the remaining 7 require that we extend our components for verification of hybrid programs. All the verifications are available in the online repository https://github.com/yonoteam/CPSVerification both as Isabelle code and in a proof document.

Number	Problem	Status
1	Basic assignment	verified
2	Overwrite assignment on some branches	verified
3	Overwrite assignment in loop	verified
4	Overwrite assignment in ODE	verified
5	Overwrite with nondeterministic assignment	verified
6	Tests and universal quantification	verified
7	Overwrite assignment several times	verified
8	Potentially overwrite dynamics	verified
9	Potentially overwrite exponential decay	verified
10	Dynamics: Cascaded	verified
11	Dynamics: Single integrator time	verified
12	Dynamics: Single integrator	verified
13	Dynamics: Double integrator	verified
14	Dynamics: Triple integrator	verified
15	Dynamics: Exponential decay (1)	verified
16	Dynamics: Exponential decay (2)	verified
17	Dynamics: Exponential decay (3)	verified
18	Dynamics: Exponential growth (1)	verified
19	Dynamics: Exponential growth (2)	verified
20	Dynamics: Exponential growth (4)	verified
21	Dynamics: Exponential growth (5)	verified
22	Dynamics: Rotational dynamics (1)	verified

Number	Problem	Status
23	Dynamics: Rotational dynamics (2)	verified
24	Dynamics: Rotational dynamics (3)	verified
25	Dynamics: Spiral to equilibrium	verified
26	Dynamics: Open cases	verified
27	Dynamics: Closed cases	verified
28	Dynamics: Conserved quantity	verified
29	Dynamics: Darboux equality	verified
30	Dynamics: Fractional Darboux equality	requires variation of Picard-Lindelöf
31	Dynamics: Darboux inequality	verified without Lipschitz continuity
32	Dynamics: Bifurcation	verified
33	Dynamics: Parametric switching between two different damped oscillators	verified with the help of external CAS
34	Dynamics: Nonlinear 1	verified
35	Dynamics: Nonlinear 2	verified
36	Dynamics: Nonlinear 4	verified
37	Dynamics: Nonlinear 5	verified
38	Dynamics: Riccati	verified
39	Dynamics: Nonlinear differential cut	verified
40	STTT Tutorial: Example 1	verified
41	STTT Tutorial: Example 2	verified
42	STTT Tutorial: Example 3a	verified
43	STTT Tutorial: Example 4a	verified
44	STTT Tutorial: Example 4b	verified
45	STTT Tutorial: Example 4c	verified
46	STTT Tutorial: Example 5	verified
47	STTT Tutorial: Example 6	verified
48	STTT Tutorial: Example 7	verified
49	STTT Tutorial: Example 9a	verified
50	STTT Tutorial: Example 9b	requires arithmetic support for exponentials
51	STTT Tutorial: Example 10	requires arithmetic support for real numbers
52	LICS: Example 1 Continuous car accelerates forward	verified
53	LICS: Example 2 Single car drives forward	verified
54	LICS: Example 3a event/triggered car drives forward	verified
55	LICS: Example 4a safe stopping of time-triggered car	verified

Number	Problem	Status
56	LICS: Example 4b progress of time-triggered car	requires diamond operator law for ODEs
57	LICS: Example 4c relative safety of time-triggered car	verified
58	LICS: Example 5 Controllability Equivalence	verified
59	LICS: Example 6 MPC Acceleration Equivalence	verified with the help of external CAS
60	LICS: Example 7 Model-Predictive Control Design car	requires arithmetic support for real numbers
61	ETCS: Essentials	verified
62	ETCS: Proposition 1 (Controllability)	requires arithmetic support for real numbers
63	ETCS: Proposition 4 (Reactivity)	requires careful translation from $\mathsf{d}\mathcal{L}$ to Isabelle

List of Symbols

General Mathematics

Set category of sets and functions

$f : A \to B$ function from A to B

$a \in A$ a is an element of A

$A \subseteq B$ A is a subset of B

$A \times B$ cartesian product of A and B

$A \setminus B$ set of elements in A not in B

B^A functions from A to B

$A \cap B$ intersection of A and B

$A \cup B$ union of A and B

$\bigcup S$ union of all sets in S

$(f \cup g)\, s$ pointwise union of $f\, s$ and $g\, s$

$(f \circ g)$ function composition of f and g

$f[a \mapsto b]$ update value of f at a to b

id identity function

$\emptyset$ empty set

$\top$ constantly true predicate

$\mathbb{B}$ set of boolean values

$\mathbb{Z}$ set of integers

$\mathbb{N}$ set of nonnegative integers

$\mathbb{R}$ set of real numbers

$\mathbb{R}_+$ set of nonnegative reals

$\downarrow Y\, x$ downward closure on Y of x

K_f set of fixpoints of $f : K \to K$

$\mathcal{P}\, S$ set of all subsets of S

$\mathcal{P}\, f\, S$ direct image of S under f

$\mathbf{1}_{\mathcal{C}}$ identity functor on category $\mathcal{C}$

Rel category of sets and relations

$s_1\ R\ s_2$ object s_1 is R-related to s_2

$R_1\,;R_2$ relational composition of R_1 and R_2

$\overline{R}$ complementation of relation R

$R^\smile$ converse of relation R

Id_S identity relation on S

R^* reflexive transitive closure of R

$\lceil P \rceil_{\mathcal{R}}$ lifting from predicates to relations

$\mathbf{Set}_{\mathcal{P}}$ Kleisli category of the powerset monad

$f \circ_K g$ Kleisli composition of f and g

η_S Kleisli unit on S

μ_S union of a collection of subsets of S

f^{*_K} Kleisli finite iteration of f

$f^\dagger$ Kleisli extension of f

f^{op} opposite state transformer of f

$\overline{f}$ complementation of state transformer f

$\mathcal{F}$ isomorphism from relations to state transformers

$\mathcal{R}$ isomorphism from state transformers to relations

$\lceil P \rceil_{\mathcal{F}}$ lifting from predicates to state transformers

$\mathcal{C}(X, Y)$ set of $\mathcal{C}$-morphisms with source X and target Y

$f \circ_{\mathcal{C}} g$ $\mathcal{C}$-composition of morphisms f and g

$\mathbf{Set}^{\mathcal{P}}$ category of Eilenberg-Moore algebras of the powerset monad

∂ lattice dualisation operator

L^{op} lattice with the opposite order to L

$\bigsqcup A$ supremum of the ordered set A

$\bigsqcap A$ infimum of the ordered set A

$\mathcal{T}(L)$ transformers on L

$\mathcal{T}_{\leq}(L)$ order-preserving transformers on L

$\mathcal{T}_{\sqcup}(L)$ sup-preserving transformers on L

$\mathcal{T}_{\sqcap}(L)$ inf-preserving transformers on L

Hybrid Programs

$x := e$ program assignment

$x' = f \,\&\, G$ evolution commands

$\alpha + \beta$ nondeterministic choice

$\alpha \,;\, \beta$ sequential composition

α^* finite iteration

$?\alpha$ test

skip ineffective program

abort aborting program

if t **then** α **else** β conditional branching

while t **do** α while loop

loop α finite iteration

α **inv** i program with annotated invariant

$\mathsf{d}\mathcal{L}$ differential dynamic logic

$(x' = f \,\&\, G)_U$ evolution command on U

$x' = f \,\&\, G$ on $U\,S$ at t_0 orbital based evolution command

evol $\varphi\, G\, U$ dynamics based evolution command

$x :=?$ nondeterministic assignment

DS solve axiom/rule of $\mathsf{d}\mathcal{L}$

DC differential cut axiom/rule of $\mathsf{d}\mathcal{L}$

DW differential weakening axiom/rule of $\mathsf{d}\mathcal{L}$

DI differential induction axiom/rule of $\mathsf{d}\mathcal{L}$

DG differential ghost axiom/rule of $\mathsf{d}\mathcal{L}$

DE differential effect axiom/rule of $\mathsf{d}\mathcal{L}$

Kleene Algebras

KAT Kleene algebra with tests

$\alpha + \beta$ addition of α and β

$\alpha \cdot \beta$ multiplication of α and β

α^* Kleene star of α

$\neg\alpha$ complementation of α

$\alpha \leq \beta$ dioid order

0 additive unit

1 multiplicative unit

$\{p\}\,\alpha\,\{q\}$ Hoare triple

$P \to Q$ P implies Q

$P \leftrightarrow Q$ P if and only if Q

$P \wedge Q$ conjunction of P and Q

$P \vee Q$ disjunction of P and Q

MKA modal Kleene algebra

ad antidomain operation

d domain operation

ar antirange operation

r range operation

$|\alpha]\, p$ forward box of α on p

$|\alpha\rangle\, p$ forward diamond of α on p

$[\alpha|\, p$ backward box of α on p

$[p, q]$ refinement operation

$\langle\alpha|\, p$ backward diamond of α on p

$x : A \Longrightarrow S$ x is a lens with view A and source S

get_x the get function of lens x

put_x the put function of lens x

$x \bowtie y$ lens x is independent of lens y

$x \sharp f$ unrestriction of f on x

$f(x \mapsto e)$ lens-based state update after f

Differential Equations

$F\,(t, x\,t, x'\,t, \ldots, x^{(n)}\,t) = 0$ explicit ODE of order n

$x^{(n)}\,t = g\,(t, x\,t, x'\,t, \ldots, x^{(n-1)}\,t)$ implicit ODE of order n

$x'\,t = g\,(t, x\,t)$ implicit first order ODE

$X'\,t = f\,(t, X\,t)$ vector form of a system of first order ODEs

$f : T \times S \to S$ time-dependent vector field

$f : S \to S$ autonomous vector field

(t_0, s) initial condition

t_0 initial time

$\int_a^b g\,\tau d\tau$ multivariate integral of g from a to b

$\lim_{n\to\infty} s_n$ limit of the sequence $s : \mathbb{N} \to S$

$\overline{B_\varepsilon(s)}$ closed ball of radius ε around s

$|t|$ absolute value of t

$\|s\|$ euclidean norm of vector s

$\exp t$ exponential operator on t

φ_s^f unique solution for initial state s of IVP given by f and (t_0, s)

T_s interval of existence of the unique solution for initial state s of IVP given by f and (t_0, s)

$\varphi : T \to S \to S$ the flow of a system of ODEs

$\gamma^\varphi\, s$ the orbit of s

$\gamma\, X\, G\, U$ the G-guarded orbit of X

Sols $f\, U\, S\, t_0\, s$ set of solutions to the system of ODEs given by f

$\gamma_G^f\, s$ the G-guarded orbital of s

$X'\,t = A\,t \cdot X\,t + B\,t$ vector representation of affine system of ODEs

$X'\,t = A\,t \cdot X\,t$ vector representation of linear system of ODEs

$\|A\|_{op}$ operator norm of matrix A

Isabelle Syntax

AFP Archive of Formal Proofs

$\equiv$ meta-logic equality

$\Longrightarrow$ meta-logic implication

$\bigwedge$ meta-logic universal quantification

prop meta-type of propositions

'a, 'b, ... type-variables

$'a \Rightarrow 'b$ type of functions from $'a$ to $'b$

$'a \times 'b$ product type of $'a$ and $'b$

$'a + 'b$ sum type of $'a$ and $'b$

$'a\ set$ type of sets over $'a$

$'a\ list$ type of lists over $'a$

nat type of natural numbers

int type of integers

real type of real numbers

bool type of boolean values

True boolean value true

False boolean value false

$\lambda x.\ f\ x$ function that maps x to $f\ x$

id identity function

Id identity relation

UNIV set of all terms of a given type

{} emptyset

lemma adds object-logic theorems

definition adds defining equations

abbreviation adds meta-definitions

class declares a new class

begin indicates the start of a context

end indicates the end of a context

assumes adds its argument as an assumption to a context

and extends the input of other keywords in a context

fixes sets a type for a specific symbol in a context

shows declares the proof obligation of a lemma

apply uses a tactic to progress a proof

by uses a tactic to conclude a proof

instantiation declares the beginning of an instantiation context

lift-definition interprets its argument as a class function

instance starts an instantiation proof

interpretation starts an interpretation proof

locale declares a new locale

sublocale starts a locale interpretation proof

context to (re)start a context of assumptions

typedef to (re)start a context of assumptions

named-theorems groups lemmas in a list to create tactics

notation introduces alternative notation for a function

utp-lift-notation allows direct input of predicates without lambda abstractions

U allows direct input of predicates without lambda abstractions

$A *v\ s$ matrix product of A times vector s

$a *_R\ s$ scaling of vector s by real number a

Index