



Aalborg Universitet

AALBORG UNIVERSITY
DENMARK

Scalable Automated Verification for Cyber-Physical Systems in Isabelle/HOL

Munive, Jonathan Julián Huerta y; Foster, Simon; Gleirscher, Mario; Struth, Georg; Laursen, Christian Pardillo; Hickman, Thomas

Publication date:
2024

[Link to publication from Aalborg University](#)

Citation for published version (APA):

Munive, J. J. H. Y., Foster, S., Gleirscher, M., Struth, G., Laursen, C. P., & Hickman, T. (2024). *Scalable Automated Verification for Cyber-Physical Systems in Isabelle/HOL*.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

Take down policy

If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.

Scalable Automated Verification for Cyber-Physical Systems in Isabelle/HOL

Jonathan Julián Huerta y Munive^{[ID](#)}
Czech Institute of Informatics, Robotics and Cybernetics
Prague, Czechia

Simon Foster^{[ID](#)}
University of York
United Kingdom

Mario Gleirscher^{[ID](#)}
University of Bremen
Germany

Georg Struth^{[ID](#)}
University of Sheffield
United Kingdom

Christian Pardillo Laursen^{[ID](#)}
University of York
United Kingdom

Thomas Hickman^{[ID](#)}
Genomics PLC
United Kingdom

22/January/2024

Abstract

We formally introduce IsaVODEs (Isabelle verification with Ordinary Differential Equations), a framework for the verification of cyber-physical systems. We describe the semantic foundations of the framework’s formalisation in the Isabelle/HOL proof assistant. A user-friendly language specification based on a robust state model makes our framework flexible and adaptable to various engineering workflows. New additions to the framework increase both its expressivity and proof automation. Specifically, formalisations related to forward diamond correctness specifications, certification of unique solutions to ordinary differential equations (ODEs) as flows, and invariant reasoning for systems of ODEs contribute to the framework’s scalability and usability. Various examples and an evaluation validate the effectiveness of our framework.

Keywords: cyber-physical systems, hybrid systems, program verification, interactive theorem proving, predicate transformers, lenses

1 Introduction

Cyber-physical systems (CPSs) are computerised systems whose software (the “cyber” part) interacts with their physical environment. The software is modelled as a variable-updating, potentially non-deterministic program, while the environment is modelled by a system of ordinary differential equations (ODEs). When CPSs interact with humans, for example through robotic manipulators, they are invariably safety-critical, which makes their design verification imperative.

However, CPS verification is challenging because of the complex interactions between the software, hardware, and the physical environment. These produce uncountably infinite state spaces, which makes verification generally intractable, and so requires the use of abstractions. A deductive verification approach uses symbolic logics, which support the encoding of solutions and invariants of ODEs. Such an approach has been implemented with the KeYmaera X tool (KYX) [31], which models CPSs via hybrid programs, and verifies their behaviour with differential dynamic logic (d \mathcal{L}). Numerous case studies and competitions support the applicability of this approach [44, 52, 56, 85].

General-purpose interactive theorem provers (ITPs), like Coq, Lean, and Isabelle, also support CPS verification [40, 25]. Their track record for large-scale deductive verification is well-documented [5, 14, 45, 47, 49]. A significant reason for these successes is the combination of

expressive logical languages, and plugins to enhance reasoning capacity and usability. Different from bespoke axiomatic provers like KYX, these tools are inherently extensible since the mathematics is constructed from first principles, rather than postulated, with accompanying soundness guarantees. For example, the recent addition of transcendental functions to KYX necessitated an extension of the tool, whereas these have been available in HOL, Coq, and Isabelle for many years. Moreover, the foundational mathematical libraries are under constant development by the community¹, and so ITP-based verification tools benefit from an ever-growing library of definitions and theorems.

Our ITP of choice, Isabelle/HOL, provides a framework supporting assured software engineering [45, 11], through an open extensible architecture; a flexible syntax pipeline [82, 60]; integration of external analysis tools in Sledgehammer, Nitpick, and cousins; and output of software artefacts in a variety of languages [34]. Isabelle’s rich library for multi-variate analysis (HOL-Analysis) can be applied to CPS verification [37, 42]. Thanks to its higher-order logic, Isabelle supports several modelling facilities essential to control engineering, such as vectors, matrices, and transcendental functions (e.g. \sin , \cos , and e^x) [8, 38], and it has extended reasoning capability through its integration with SMT solvers (CVC4, Z3, Vampire) [61]. Isabelle is therefore ideally suited to reasoning about CPSs, but to harness its facilities we need an accessible and powerful CPS modelling language and verification framework.

In this paper we contribute an Isabelle-based verification framework called IsaVODEs (Isabelle Verification with Ordinary Differential Equations) [40, 25]. IsaVODEs provides a textual language for modelling CPSs, which is constructed as a shallow embedding in Isabelle. The language is equipped with nondeterministic state-transformer-based program semantics and a flexible hybrid store model based on lenses [23, 27]. The program model provides several verification calculi, including Hoare logic and dynamic logic, with modalities for specifying both safety and reachability properties. The store model allows software models to benefit from the full generality of the Isabelle type system, including continuous structures like vectors and matrices, and discrete structures like algebraic data types. Our language can therefore scale to systems with both realistic dynamics, and complex control structures. We harness Isabelle’s syntax translation mechanisms to provide a user-friendly frontend for the language, including declarative context, program notation, and operators for arithmetic, vectors, and matrices. Our technical solution is extensible and endeavours to resemble languages like Modelica, Mathematica, and MATLAB.

Verification of models in IsaVODEs is supported by a library of deduction rules. For reasoning about ODEs, we support both the use of solutions with flow functions, and differential induction, which avoids the need for solutions. We integrate two Computer Algebra Systems (CASs), namely Mathematica and SageMath, into Isabelle for the supply of symbolic solutions to Lipschitz-continuous ODEs in an analogous way to sledgehammer. We also support both a $d\mathcal{L}$ -style differential ghost rule and Darboux rules that enhance IsaVODEs’ invariant reasoning capabilities for complex continuous dynamics.

Orthogonal to this, we also contribute novel laws to support local reasoning in the style of separation logic. We can infer the frame of a program (i.e. the mutable variables), and use a frame rule to prove invariants over variables outside the frame. This is accompanied by a novel form of differentiation called “framed Fréchet derivatives”, which allow us to differentiate with respect to a strict subset of the store variables, particularly ignoring any discrete variables lacking a topological structure. When variables are outside of the frame of a system of ODEs, those variables are unchanged during continuous evolution. Local reasoning further facilitates scalability of our tool, by allowing a verification task to be decomposed into modules supported by individual assumptions and guarantees.

We supply several proof methods to support automated proof and verification condition generation (VCG), including the application of differential induction, certification of ODE solutions, and the various Hoare-logic laws. Care is taken to ensure that the resulting proof obligations are presented in a way that preserves abstraction, and elides irrelevant details of the Isabelle mechanisation, to main-

¹See Isabelle’s Archive of Formal Proofs (<https://www.isa-afp.org/statistics/>) and Lean’s Mathematical Library (https://leanprover-community.github.io/mathlib_stats.html).

tain the user experience. Harnessing Isabelle’s Eisbach tool [53], we also employ various high-level search-based proof methods, which exhaustively apply the proof rules to compute a complete set of VCs. The VCs can finally be tackled in the usual way using tools like auto and sledgehammer.

The framework has been tested successfully on a large set of hybrid verification benchmarks [54], and many larger examples. Our enhancements yield at least the same performance in essential verification tasks as that of domain-specific provers. Initial case studies suggest a simplification of user interaction. Our new components can be found online². Our contributions are highlighted through examples, while additional contributions are noted throughout the text.

This article is a substantial extension of previously published work at the Formal Methods symposium (FM2021) [27]. Our additional contributions include generalisations of the laws for frames (§5.1), differential ghosts, and Darboux rules (§5.4); proof methods for certification of continuity, Lipschitz continuity, and uniqueness of solutions (§6.2); extensions to our previous proof methods (§6); the CAS integrations (§6.5); a more substantial evaluation (§7); and several additional examples (§2, §8). As a result of these enhancements, the usability of the tool is more mature and polished.

We begin our paper by illustrating the framework with a small case study (§2). We then expound our formalisation of dynamical systems (§3.1), the state-transformer hybrid program model (§3.2), the foundations of the store model (§3.3), the specification of continuous evolution (§3.4), and finally VCG through our framework’s algebraic foundations (§3.5). This includes both safety, and also reachability and termination (§3.5.3).


Next, in §4, we introduce our framework’s hybrid modelling language. We supply commands for generating hybrid stores, with variables, constants, and foundational properties (§4.1), and user-friendly notation for expressions (§4.3), matrices, and vectors (§4.4). The notation is automatically processed via rewriting rules supplied to Isabelle’s simplifier, hiding implementation details.

In §5 we present laws for local reasoning about frames (§5.1), the seamless integration of framed ODEs into our hybrid programs (§5.2), and automatic certification of ODE invariants through framed Fréchet derivatives (§5.3). These also serve to supply new $d\mathcal{L}$ -style ghost and Darboux rules that enhance IsaVODEs’ invariant reasoning (§5.4).

We contribute increased IsaVODEs proof-capabilities for the verification process. We formalise theorems necessary for increased automation, like the fact that differentiable functions are Lipschitz-continuous (§6.2) or the first and second derivative test laws (§6.7). We provide methods to automate certification of differentiation (§6.1), uniqueness of solutions (§6.3), invariants for ODEs (§6.6), and VCG (§6.4). Additional automation is provided through the integration of Mathematica and SageMath (§6.5). We then bring all of the results together for the evaluation (§7) using benchmarks and examples (§8). In the remaining sections we review related work (§9), provide concluding statements, and discuss future research avenues (§10).

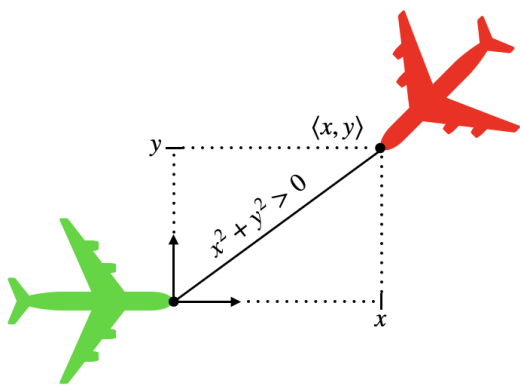
2 Motivating Case Study: Flight Dynamics

In this section, we motivate and demonstrate our framework’s usage with a worked example: an aircraft collision avoidance scenario that was first presented by Mitsch et. al. [55]. It describes an aircraft trying to avoid a collision with a nearby intruding aircraft travelling at the same altitude. We can model this intruding aircraft by considering its coordinates x and y , and angle ϑ , in the reference frame of our own ship. The intruding plane has fixed velocity $v_i > 0$, and our plane has fixed velocity $v_o > 0$ and variable angular velocity ω . This is illustrated in Figure 1a and 1b.

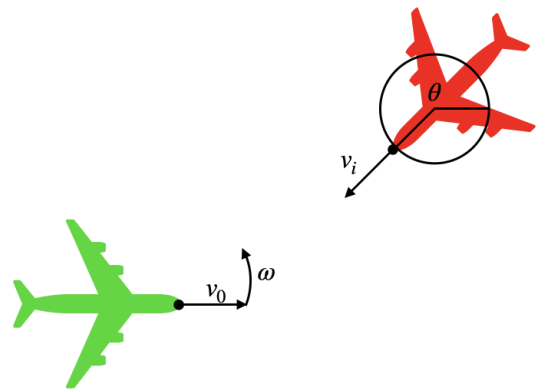
With our tool, we can model this using a `dataspace` command, as shown below: 

```
dataspace planar_flight =
  constants
  v_o :: real (* own_velocity *)
```

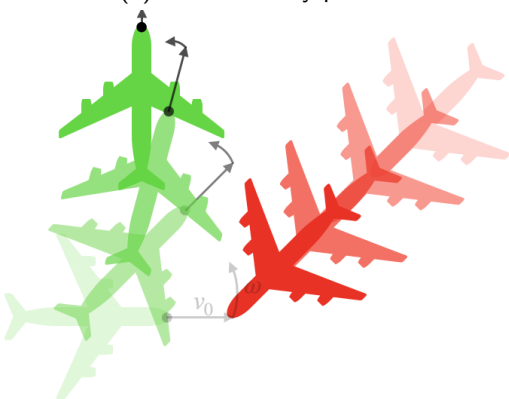
²github.com/isabelle-utp/Hybrid-Verification, also by clicking our icons .



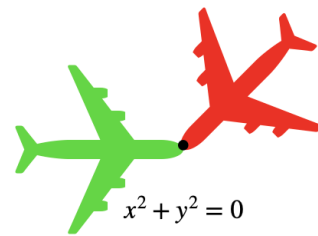
(a) Plane x and y position



(b) Velocities and angle



(c) Evasive maneuvers



(d) Collision

Figure 1: Diagrams illustrating the flight dynamics example

```

    vi :: real (* intruder velocity *)
assumes
    vo_pos: "vo > 0" and
    vi_pos: "vi > 0"
variables (* Coordinates in reference frame of own ship *)
    x :: real (* Intruder x *)
    y :: real (* Intruder y *)
    θ :: real (* Intruder angle *)
    ω :: real (* Angular velocity *)

```

This command allows us to define our constants, any assumptions about the verification problem, and the system's variables. In this case, we postulate two constants v_o and v_i , both of type `real`, for the velocity of the aircraft and intruder, respectively. Here, `real` is the type of precise mathematical real numbers, as opposed to floating points or rationals.

As per the problem statement, we assume that both of these constants are strictly positive. This is specified by the assumptions called `vo_pos` and `vi_pos`, respectively, which can be used as hypotheses in proofs. We supply the variables of this system, which give the relative position of the intruder (x and y), its orientation θ , and its angular velocity ω .

We next specify the ODEs that model the system as a constant called `plant`:

```

definition "plant ≡ {x' = vi * cos θ - vo + ω * y,
                    y' = vi * sin θ - ω * x,
                    θ' = -ω}"

```

The ODEs can be specified in a user-friendly manner, as physicists and engineers would informally state them, in terms of the constants and variables of the system x , y , and φ .

Next, we define a simple controller to avoid collisions, as explained below:

```

abbreviation "I ≡ (vi * sin θ * x - (vi * cos θ - vo) * y
                    > vo + vi)e"

```

```

abbreviation "J ≡ (vi * ω * sin θ * x - vi * ω * cos θ * y
                    + vo * vi * cos θ
                    > vo * vi + vi * ω)e"

```

```

definition "ctrl ≡ (ω ::= 0; ⋀I?) □ (ω ::= 1; ⋀J?)"

```

This is based on two invariants, I and J , for two different scenarios. These are properties that remain true after the unfolding of each scenario. The invariant I holds when going straight is safe, while J holds when an evasion manoeuvre is allowed. Our controller selects whether to set our aircraft's angular velocity to 0 or 1 depending on which invariant holds (I and J respectively). The evasive manoeuvres that occur when the angular velocity is set to 1 are illustrated in figure 1c.


Finally, our model follows the usual structure of an iteration (*) of control choices (`ctrl`) followed by a nondeterministic evolution of the system dynamics (`plant`):

```

definition "flight ≡ (ctrl; plant)*"

```

The behaviour of the system is characterised by all states reachable by executing the controller followed by the plant a finite number of times.

Next, we show how we can formally verify the collision avoidance of this system. We do this by specifying a Hoare triple within an Isabelle lemma: 

```

lemma flight_safe: "{x2 + y2 > 0} flight {x2 + y2 > 0}"
proof -
  have ctrl_post: "{x2 + y2 > 0} ctrl {(\omega = 0 \wedge @I) \vee (\omega = 1 \wedge @J)}"
    unfolding ctrl_def by wlp_full

  have plant_safe_I: "{\omega = 0 \wedge @I} plant {x2 + y2 > 0}"
    unfolding plant_def apply (dInv "\$ \omega = 0 \wedge @I", dWeaken)
    using v_o_pos v_i_pos sum_squares_gt_zero_iff by fastforce

  have plant_safe_J: "{\omega = 1 \wedge @J} plant {x2 + y2 > 0}"
    unfolding plant_def apply (dInv "\omega=1 \wedge @J", dWeaken)
    by (smt (z3) cos_le_one mult_if_delta mult_le_cancel_iff2
        mult_left_le sum_squares_gt_zero_iff v_i_pos v_o_pos)

  show ?thesis
    unfolding flight_def
    apply (intro hoare_kstar_inv hoare_kcomp[OF ctrl_post])
    by (rule hoare_disj_split[OF plant_safe_I plant_safe_J])
qed

```

We formulate collision avoidance using $x^2 + y^2 > 0$ as the invariant in the Hoare triple. A collision occurs when $x^2 + y^2 = 0$, as illustrated in figure 1d. We use Isabelle’s Isar scripting language to break down the verification into several intermediate properties via the Isar command “**have**”, which takes a label followed by a property specification.

We first calculate the postconditions that arise from running the controller (`ctrl`) by using our tactic `wlp_full` (see Section 6). This gives us two possible execution branches – one where $I \wedge \omega = 0$ holds, and one where $J \wedge \omega = 1$ holds. We give this first property the name `ctrl_post`.

The postconditions provide two possible initial states for the plant, which we consider using the properties `plant_safe_I` and `plant_safe_J`. We show that both preconditions guarantee the problem’s postcondition $x^2 + y^2 > 0$ by applying differential induction, a technique that proves an invariant of a system of ODEs without computing a solution (see Section 5.3). In each case, we use our Eisbach-designed method `dInv`, which takes as a parameter the invariant we wish to prove. The invariant is simply the precondition of each Hoare triple. However, we then need to prove that this invariant implies the postcondition, which is done using a further method called `dWeaken`. The remaining proof obligations can be solved using the **Sledgehammer** tool, which calls external automated theorem provers to find a solution and reconstructs it using the names of previously proven theorems in Isabelle’s libraries. In particular, the property `plant_safe_J` is discharged using Z3, which uses trigonometric identities formalised in HOL-Analysis. We display the proof state resulting from applying differential induction below:

$$\begin{aligned}
 & \backslash \$ \omega = 1 \wedge \\
 & v_o * v_i + v_i * \$ \omega \\
 & < v_i * \$ \omega * \sin (\$ \vartheta) * \$ x - v_i * \$ \omega * \cos (\$ \vartheta) * \$ y + \\
 & \quad v_o * v_i * \cos (\$ \vartheta) \longrightarrow \\
 & 0 < (\$ x)^2 + (\$ y)^2 \backslash
 \end{aligned}$$

In the case that **Sledgehammer** is unable to find a proof, this proof obligation is quite readable, and so a manual proof or refutation could be given (See Section 8).

Finally, we put everything together to show that the whole system is safe, using the Isar command **show**, which is used to conclude proofs and restate the overall goal of the lemma (`?thesis`). The proof uses a couple of high-level Hoare logic laws, and the properties that were proven to complete

the proof. This final step can be completely automated using Isabelle's classical reasoner, but we leave the details for the purpose of demonstration.

This completes our overview of our tool and its capabilities. In the remainder of the paper, we will expound the technical foundations of the tool, and our key results.

3 Semantics for hybrid systems verification

We start the section by recapitulating basic concepts from the theory of dynamical systems. We use these notions to describe our approach [40, 27] to hybrid systems verification in general-purpose proof assistants. Specifically, we present hybrid programs and their state transformer semantics. We then introduce our store model and provide intuitions for deriving state transformers for program assignments and ODEs relative to this model. We extend our approach to a predicate transformer semantics to derive laws for verification condition generation (VCG) [40, 9]. That is, we introduce the forward box predicate transformer and use its properties to derive the rules of Hoare logic. Finally, we introduce the forward diamond predicate transformer which serves to prove reachability and progress properties of hybrid systems. Our formalisation of these concepts as a shallow embedding in Isabelle/HOL maximises the availability of the prover's proof automation in VCG.

3.1 Dynamical Systems

In this section, we consider two ways to specify continuous dynamical systems [78]: explicitly via flows and implicitly via systems of ordinary differential equations (ODEs). *Flows* are functions $\varphi : T \rightarrow \mathcal{C} \rightarrow \mathcal{C}$, where T is a non-discrete submonoid of the real numbers \mathbb{R} representing time. Similarly, \mathcal{C} is a set with some topological structure like a vector space or a metric space. We emphasise this intuition using an overhead arrow for elements $\vec{c} \in \mathcal{C}$ and refer to \mathcal{C} as a *continuous* state space. By definition, flows are continuously differentiable (C^1) functions and monoid actions: they satisfy the laws $\varphi(t_1 + t_2) = \varphi t_1 \circ \varphi t_2$ and $\varphi 0 = id_{\mathcal{C}}$. Given a state $\vec{c} \in \mathcal{C}$, the *trajectory* $\varphi_{\vec{c}} : T \rightarrow \mathcal{C}$, such that $\varphi_{\vec{c}} t = \varphi t \vec{c}$, is a curve modelling the continuous dynamical system's evolution in time and passing through \vec{c} . The *orbit map* $\gamma^{\varphi} : \mathcal{C} \rightarrow \mathcal{P} \mathcal{C}$, such that $\gamma^{\varphi} \vec{c} = \mathcal{P} \varphi_{\vec{c}} T$, gives the graph of this curve for \vec{c} , where \mathcal{P} denotes both the powerset operation and the direct image operator.

Systems of ODEs are related to flows through their solutions. Formally, systems of ODEs are specified by *vector fields* $f : T \rightarrow \mathcal{C} \rightarrow \mathcal{C}$, functions assigning vectors to points in space-time. A *solution* $X : T \rightarrow \mathcal{C}$ to the system of ODEs specified by f is then a C^1 -function such that $X' t = f t (X t)$ for all t in some interval $U \subseteq T$. This solution also solves the associated *initial value problem* (IVP) given by (t_0, \vec{c}) (denoted by $X \in ivp\text{-sols } U f t_0 \vec{c}$) if it satisfies $X t_0 = \vec{c}$, with $t_0 \in U$. The existence of solutions to IVPs is guaranteed for continuous vector fields by the Peano theorem, albeit on an interval $U \vec{c}$ depending on the initial condition (t_0, \vec{c}) . Similarly, the Picard-Lindelöf theorem states that all solutions to the IVP $X' t = f t (X t)$ with $X t_0 = \vec{c}$ coincide in some interval $U \vec{c} \subseteq T$ (around t_0) if f is Lipschitz continuous on T . In other words, it states that there is a unique solution to the IVP on $U \vec{c}$. Thanks to this, when f is Lipschitz continuous on T , $t_0 = 0$, and $U \vec{c} = T$ for all $\vec{c} \in \mathcal{C}$, the solutions to the associated IVPs are exactly a flow's trajectories $\varphi_{\vec{c}}$, that is, $\varphi_{\vec{c}}' t = f t (\varphi_{\vec{c}} t)$ and $\varphi_{\vec{c}} 0 = \vec{c}$. Therefore, the flow φ is the function mapping to each $\vec{c} \in \mathcal{C}$ the unique $\varphi_{\vec{c}}$ such that $\varphi_{\vec{c}} \in ivp\text{-sols } T f 0 \vec{c}$.

Example 1. We illustrate the above properties about flows and ODEs with the differential equation $y' t = a \cdot y t + b$, where $a \neq 0$ and $t \in \mathbb{R}$. This is an important ODE modelling for instance idealised bacterial growth [76], radioactive decay [81] or concentration of glucose in the blood (without the intervention of insulin) [2]. First, given that differentiability implies Lipschitz continuity [78], we can verify that this equation has unique solutions (by the Picard-Lindelöf theorem) simply by noticing that $f t = a \cdot y t + b$ is differentiable with derivative $f' t = a \cdot y' t$. The solution to the associated

IVP with initial condition $y(0) = c$ is $\varphi_c t = \frac{b}{a} \cdot e^{a \cdot t} + c \cdot e^{-a \cdot t} - \frac{b}{a}$. Indeed,

$$\begin{aligned} \varphi_c' t &= \left(\frac{b}{a} \cdot e^{a \cdot t} + c \cdot e^{-a \cdot t} - \frac{b}{a} \right)' \\ &= \frac{b}{a} \cdot a \cdot e^{a \cdot t} + c \cdot a \cdot e^{-a \cdot t} \\ &= a \cdot \left(\frac{b}{a} \cdot e^{a \cdot t} + c \cdot e^{-a \cdot t} - \frac{b}{a} \right) + b \\ &= a \cdot \varphi_c t + b. \end{aligned}$$

It is also easy to check that $\varphi_c 0 = c$. Hence, the mapping $\varphi : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}$, such that $\varphi t c = \varphi_c t$, is the unique flow associated to the ODE $y' t = a \cdot y t + b$. Indeed, for a fixed $\tau \in \mathbb{R}$, the function $g : \mathbb{R} \rightarrow \mathbb{R}$ such that $g t = \varphi(t + \tau) c = \varphi_c(t + \tau)$ satisfies $g 0 = \varphi_c \tau$ and also $g' t = \varphi_c'(t + \tau) = a \cdot \varphi_c(t + \tau) + b = a \cdot (g t) + b$. However, by uniqueness, the only function satisfying these two equations is $\varphi_{\varphi_c \tau}$. Hence $g t = \varphi_{\varphi_c \tau} t$, thus, the monoid law $\varphi(t + \tau) c = \varphi t(\varphi_c \tau)$ holds. Thus, the function φ mapping points c to IVP-solutions φ_c of the ODE $y' t = a \cdot y t + b$ satisfies the monoid action laws, and is therefore, a flow. \square

3.2 State transformer semantics for hybrid programs

Having introduced some basic concepts from the theory of dynamical systems, we present our hybrid systems model. We represent these via hybrid programs which are traditionally defined syntactically [35, 65]. Yet, our approach is purely semantic and we merely provide the recursive definition below as a guide to what our semantics should model:


$$\alpha ::= x := e \mid x' = f \ \& \ G \mid \text{!}P? \mid \alpha ; \beta \mid \alpha \sqcap \beta \mid \alpha^*.$$

Typically, x denotes variables; e and f are terms, and G and P are assertions. In dynamic logic [35], the statement $x := e$ represents an *assignment* of variable x to expression e , $\text{!}P?$ models testing whether P holds, while $\alpha ; \beta$, $\alpha \sqcap \beta$ and α^* are the *sequential composition*, *nondeterministic choice*, and *finite iteration* of programs α and β . Well-known while-programs emerge via the equations $\text{if } P \text{ then } \alpha \text{ else } \beta \equiv (\text{!}P? ; \alpha) \sqcap (\text{!}\neg P? ; \beta)$ and $\text{while } P \text{ do } \alpha \equiv (\text{!}P? ; \alpha)^* ; \text{!}\neg P?$. Beyond these, differential dynamic logic (dL) [65] introduces *evolution commands* $x' = f \ \& \ G$ that represent systems of ODEs with *boundary conditions* or *guards* G that delimit the solutions' range to the region described by G .

We use *nondeterministic state transformers* $\alpha : \mathcal{S} \rightarrow \mathcal{P} \mathcal{S}$ as our semantic representation for hybrid programs. Thus, our “hybrid programs” are really arrows in the Kleisli category of the powerset monad. Observe that a subset of these arrows also model “assertions”, namely the *subidentities* of the monadic unit $\eta_{\mathcal{S}}$, such that $\eta_{\mathcal{S}} s = \{s\}$ for all $s \in \mathcal{S}$. That is, the functions mapping each state s either to $\{s\}$ or to \emptyset model assertions where $P s = \{s\}$ represents that P holds for s , and $P s = \emptyset$ that P does not hold for s . Henceforth, we abuse notation and identify predicates $P : \mathcal{S} \rightarrow \mathbb{B}$ (or $P \in \mathbb{B}^{\mathcal{S}}$), sets $\mathcal{P} \mathcal{S}$, and subidentities of $\eta_{\mathcal{S}}$, where \mathbb{B} denotes the Booleans. We also treat predicates as logic formulae by writing, for instance, $P \wedge Q$ and $P \vee Q$ instead of $\lambda s. P s \wedge Q s$ and $\lambda s. P s \vee Q s$. Thus, we denote the constantly true and constantly false predicates by \top and \perp respectively. They coincide with the `skip` and `abort` programs such that `skip` = $\eta_{\mathcal{S}}$ and `abort` $s = \emptyset$ for all $s \in \mathcal{S}$. In this state transformer semantics, sequential compositions correspond to (forward) *Kleisli compositions* $(\alpha ; \beta) s = \bigcup \{ \beta s' \mid s' \in \alpha s \}$, nondeterministic choices are pointwise unions $(\alpha \sqcap \beta) s = \alpha s \cup \beta s$, and finite iterations are the functions $\alpha^* s = \bigcup_{i \in \mathbb{N}} \alpha^i s$, with $\alpha^{i+1} = \alpha ; \alpha^i$ and $\alpha^0 = \text{skip}$.

3.3 Store and expressions model

To introduce our state transformer semantics of assignments and ODEs, we first describe our store model. We use lenses [59, 10, 21] to algebraically characterise the hybrid store. Through an axiomatic approach to accessing and mutating functions, lenses allow us to locally manipulate program

stores [22] and algebraically reason about program variables [23, 28]. Formally, a *lens* x with *source* \mathcal{S} and *view* \mathcal{V} , denoted $x :: \mathcal{V} \Rightarrow \mathcal{S}$, is a pair of functions (get_x, put_x) with $get_x : \mathcal{S} \rightarrow \mathcal{V}$ and $put_x : \mathcal{V} \rightarrow \mathcal{S} \rightarrow \mathcal{S}$ such that 

$$get_x (put_x v s) = v, \quad put_x v \circ put_x v' = put_x v, \quad \text{and} \quad put_x (get_x s) s = s,$$

for all $v, v' \in \mathcal{V}$ and $s \in \mathcal{S}$. Usually, a lens x represents a variable, \mathcal{S} is the system's state space and \mathcal{V} is the value domain for x . Under this interpretation, get_x returns the value of variable x while put_x updates it. Yet, we sometimes interpret \mathcal{V} as a subregion of \mathcal{S} , making get_x a projection or restriction and put_x an overwriting or substituting function. For other state models using diverse variable lenses, such as arrays, maps and local variables, see [23, 22]. Lenses $x, x' :: \mathcal{V} \Rightarrow \mathcal{S}$ are *independent*, $x \bowtie x'$, if $put_x u \circ put_{x'} v = put_{x'} v \circ put_x u$, for all $u, v \in \mathcal{V}$, that is, these operations commute on all states.

We model expressions and assertions used in program syntax as functions $e : \mathcal{S} \rightarrow \mathcal{V}$, which semantically are queries over the store \mathcal{S} returning a value of type \mathcal{V} . We can use the *get* function to perform such queries by “looking up the value of variables”. For instance, if $x, y :: \mathbb{R} \Rightarrow \mathcal{S}$ model independent program variables, and $c \in \mathbb{R}$ is a constant, then the function $\lambda s. (get_x s)^2 + (get_y s)^2 \leq c^2$ represents the “expression” $x^2 + y^2 \leq c^2$. Then, function evaluation corresponds to computing the value of the expression at state $s \in \mathcal{S}$.

With this representation, the state transformer $\lambda s. \{put_x (e s) s\}$ models a program assignment, denoted by $x := e$. More generally, we turn deterministic functions $\sigma : \mathcal{S} \rightarrow \mathcal{S}$ into state transformers via function composition with the Kleisli unit, which we denote by $\langle \sigma \rangle = \eta_{\mathcal{S}} \circ \sigma$. Then, representing expressions e as functions $e : \mathcal{S} \rightarrow \mathcal{V}$, our model for variable assignments becomes $(x := e) = \langle \lambda s. put_x (e s) s \rangle$.

3.4 Model for evolution commands

To derive in our framework a state transformer semantics $\mathcal{S} \rightarrow \mathcal{P}\mathcal{S}$ for evolution commands $x' = f \& G$, observe that a flow's orbit map $\gamma^\varphi : \mathcal{C} \rightarrow \mathcal{P}\mathcal{C}$ with $\gamma^\varphi \vec{c} = \{\varphi t \vec{c} \mid t \in T\}$ is already a state transformer on \mathcal{C} . It sends each \vec{c} in the continuous state space \mathcal{C} to the reachable states of the trajectory $\varphi \vec{c}$. Based on the relationship between flows and the solutions to IVPs from Subsection 3.1, we can generalise this state transformer to a set of all points $X t$ of the solutions X of the system of ODEs represented by f , i.e. $X' t = f t (X t)$, with initial condition $X t_0 = \vec{c}$ over an interval $U \vec{c}$ around t_0 ($t_0 \in U \vec{c}$). Moreover, in line with $d\mathcal{L}$, the solutions should remain within the guard or boundary condition G : $\forall \tau \in U \vec{c}. \tau \leq t \Rightarrow G (X \tau)$. Thus, to specify dynamical systems via ODEs f instead of flows φ , we define the *generalised guarded orbits* map [40]

$$\gamma^f G U t_0 \vec{c} = \{X t \mid t \in U \vec{c} \wedge X \in \text{ivp-sols } U f t_0 \vec{c} \wedge \mathcal{P} X (t \downarrow_{U \vec{c}}) \subseteq G\},$$

that also introduces guards G , initial conditions t_0 , and intervals $U \vec{c}$. The set $t \downarrow_T$ is a downward closure $t \downarrow_T = \{\tau \in T \mid \tau \leq t\}$ which in applications becomes the interval $[0, t] = \{\tau \mid 0 \leq \tau \leq t\}$ because we usually fix $U \vec{c} = \mathbb{R}_{\geq 0} = \{\tau \mid \tau \geq 0\}$ for all $\vec{c} \in \mathcal{C}$. This is why we also abuse notation and write constant interval functions $\lambda \vec{c}. T$ simply as T . Notice that when the flow φ for f exists, $\gamma^f \top T 0 \vec{c} = \gamma^\varphi \vec{c}$.

Lenses support algebraic reasoning about variable frames: the set of variables that a hybrid program can modify. In particular, they allow us to split the state space into continuous and discrete parts. We explain a lens-based lifting of guarded orbits $\gamma^f G U t_0$ from the continuous space $\mathcal{C} \rightarrow \mathcal{P}\mathcal{C}$ to the full state space \mathcal{S} in Subsection 5.2. This produces our state transformer for evolution commands $(x' = f \& G)_{U \vec{c}}^{t_0} : \mathcal{S} \rightarrow \mathcal{P}\mathcal{S}$. Intuitively, it maps each state $s \in \mathcal{S}$ to all X -reachable states within the region G , where X solves the system of ODEs specified by f , and leaves intact the non-continuous part of \mathcal{S} . Having the same type as the above-described state transformers enables us to seamlessly use the same operations on $(x' = f \& G)_{U \vec{c}}^{t_0}$. This also enables us to do modular verification condition generation (VCG) of hybrid systems as described below.

Example 2. In this example, we use a hybrid program `blood_sugar` to model an idealised machine controlling the concentration of glucose in a patient's body. Hybrid programs are often split into discrete control `ctrl` and continuous dynamics `dyn`. Their composition is then wrapped in a finite iteration: $\text{blood_sugar} = (\text{ctrl} ; \text{dyn})^*$. For the control, we use a conditional statement reacting to the patient's blood-glucose. Concretely, the program

$$\text{ctrl} = \text{if } g \leq g_m \text{ then } g := g_M \text{ else skip}$$









states that if the value of the patient's blood-glucose concentration g is below a certain warning threshold $g_m \geq 0$, the maximum healthy dose of insulin, represented as an immediate spike to the patient's glucose $g := g_M$, is injected into the patient's body. Otherwise, the patient is fine and the machine does nothing. The continuous variable g follows the dynamics in Example 1: $y' t = a \cdot y t + b$. We assume $a = -1$ and $b = 0$ so that the concentration of glucose decreases over time. This results in the evolution command $\text{dyn} = (g' = -g \ \& \ \top)_{\mathbb{R}_{\geq 0}}^0$ which we abbreviate as $\text{dyn} = (g' = -g)$. Formally, the assignment $g := g_M$ is the state transformer $\lambda s. \{put_g g_M s\}$, the test $g \leq g_m$ is the predicate $\lambda s. get_g s \leq g_m$, and the evolution command $g' = -g$ is the orbit map γ^φ lifted to the whole space \mathcal{S} , where $\varphi t c = c \cdot e^{-t}$ for all $t \in \mathbb{R}$. \square

3.5 Predicate transformer semantics

Finally, we extend our state transformer semantics to a *predicate transformer* ($\mathbb{B}^{\mathcal{S}} \rightarrow \mathbb{B}^{\mathcal{S}}$) semantics for verification condition generation (VCG). Concretely, we define two predicate transformers and use the definition of the first one for deriving partial correctness laws, including the rules of Hoare logic, and the definition of the second one for deriving reachability laws. We also exemplify the application of these to VCG.

3.5.1 Forward boxes

We define dynamic logic's *forward box* or *weakest liberal precondition* (wlp) $[-] -$ operator $[\alpha] : \mathbb{B}^{\mathcal{S}} \rightarrow \mathbb{B}^{\mathcal{S}}$ as $[\alpha] Q s \Leftrightarrow (\forall s'. s' \in \alpha s \Rightarrow Q s')$, for $\alpha : \mathcal{S} \rightarrow \mathcal{P} \ \mathcal{S}$ and $Q : \mathcal{S} \rightarrow \mathbb{B}$. It is true for those initial system's states that lead to a state satisfying Q after executing α , if α terminates. Well-known *wlp*-laws are derivable and simple consequences from this and our previous definitions [40]. These laws allow us to automate and do VCG much more efficiently than by doing Hoare Logic since all of them, except for the loop rule, are equational simplifications from left to right:

(wlp-skip)	$[\text{skip}] Q$	$= Q$	
(wlp-abort)	$[\text{abort}] Q$	$= \top$	
(wlp-test)	$[iP?] Q$	$= P \Rightarrow Q$	
(wlp-assign)	$[x := e] Q$	$= Q[e/x]$	
(wlp-seq)	$[\alpha ; \beta] Q$	$= [\alpha] [\beta] Q$	
(wlp-choice)	$[\alpha \sqcap \beta] Q$	$= [\alpha] Q \wedge [\beta] Q$	
(wlp-loop)	$[\text{loop } \alpha] Q$	$= \forall n. [\alpha^n] Q$	
(wlp-cond)	$[\text{if } T \text{ then } \alpha \text{ else } \beta] Q$	$= (T \Rightarrow [\alpha] Q) \wedge (\neg T \Rightarrow [\beta] Q)$	

Here, n is a natural number (i.e. $n \in \mathbb{N}$), $\text{loop } \alpha$ is simply α^* , and $Q[e/x]$ is our abbreviation for the function $\lambda s. Q (put_x (e s) s)$ that represents the value of Q after variable x has been updated by the value of evaluating e on $s \in \mathcal{S}$. We write this semantic operation as a substitution to resemble Hoare logic (see Section 4.3). Similarly, the *wlp*-law for evolution commands informally corresponds to

$$(\text{wlp-evol}) \quad [(x' = f \ \& \ G)_{U}^{t_0}] Q s \Leftrightarrow \begin{array}{l} \forall X \in \text{ivp-sols } U f t_0 s. \forall t \in U s. \\ (\forall \tau \in t \downarrow_{U s}. G (X \tau)) \Rightarrow Q (X t). \end{array} \quad \img alt="blue cube icon" data-bbox="780 890 800 905"/>$$

That is, a postcondition Q holds after an evolution command starting at (t_0, s) , if and only if, the postcondition holds $Q(X t)$ for all solutions to the IVP $X' t = f t(X t)$, $X t_0 = s$, for all times in the interval $t \in U s$ whose previous times respect G . Notice that, if there is a flow $\varphi : T \rightarrow \mathcal{C} \rightarrow \mathcal{C}$ for f and $U = T = \mathbb{R}_{\geq 0}$, this simplifies to

$$(\text{wlp-flow}) \quad |x' = f \ \& \ G| Q \ s \Leftrightarrow (\forall t \geq 0. (\forall \tau \in [0, t]. G(\varphi_s \tau)) \Rightarrow Q(\varphi_s t)). \quad \text{🎉}$$

See Section 5 for the formal version of these laws.

3.5.2 Hoare triples

It is well-known that Hoare logic can be derived from the forward box operator of dynamic logic [35]. Thus, we can also write partial correctness specifications as Hoare-triples with our forward box operators via $\{P\} \alpha \{Q\} \Leftrightarrow (P \Rightarrow \text{wlp} \alpha Q)$. From our *wlp*-laws and definitions, the Hoare logic rules below hold:

(h-skip)		$\{P\} \text{ skip } \{P\}$	🎉
(h-abort)		$\{P\} \text{ abort } \{Q\}$	🎉
(h-test)		$\{P\} \text{ !}Q? \{P \wedge Q\}$	🎉
(h-assign)		$\{Q[e/x]\} x := e \{Q\}$	🎉
(h-seq)	$\{P\} \alpha \{R\} \wedge \{R\} \beta \{Q\}$	$\Rightarrow \{P\} \alpha ; \beta \{Q\}$	🎉
(h-choice)	$\{P\} \alpha \{Q\} \wedge \{P\} \beta \{Q\}$	$\Rightarrow \{P\} \alpha \sqcap \beta \{Q\}$	🎉
(h-loop)	$\{I\} \alpha \{I\}$	$\Rightarrow \{I\} \text{ loop } \alpha \{I\}$	🎉
(h-cons)	$(P_1 \Rightarrow P_2) \wedge (Q_2 \Rightarrow Q_1)$ $\wedge \{P_2\} \alpha \{Q_2\}$	$\Rightarrow \{P_1\} \alpha \{Q_1\}$	🎉
(h-cond)	$\{T \wedge P\} \alpha \{Q\}$ $\wedge \{\neg T \wedge P\} \beta \{Q\}$	$\Rightarrow \{P\} \text{ if } T \text{ then } \alpha \text{ else } \beta \{Q\}$	🎉
(h-while)	$\{T \wedge I\} \alpha \{I\}$	$\Rightarrow \{I\} \text{ while } T \text{ do } \alpha \{\neg T \wedge I\}$	🎉
(h-whilei)	$(P \Rightarrow I) \wedge (I \wedge \neg T \Rightarrow Q)$ $\wedge \{I \wedge T\} \alpha \{I\}$	$\Rightarrow \{P\} \text{ while } T \text{ do } \alpha \text{ inv } I \{Q\}$	🎉
(h-loopi)	$(P \Rightarrow I) \wedge (I \Rightarrow Q)$ $\wedge \{I\} \alpha \{I\}$	$\Rightarrow \{P\} \text{ loop } \alpha \text{ inv } I \{Q\}$	🎉
(h-evoli)	$(P \Rightarrow I) \wedge (I \wedge G \Rightarrow Q)$ $\wedge \{I\} (x' = f \ \& \ G)_U^{t_0} \{I\}$	$\Rightarrow \{P\} (x' = f \ \& \ G)_U^{t_0} \text{ inv } I \{Q\}$	🎉
(h-conji)	$\{I\} \alpha \{I\} \wedge \{J\} \alpha \{J\}$	$\Rightarrow \{I \wedge J\} \alpha \{I \wedge J\}$	🎉
(h-disji)	$\{I\} \alpha \{I\} \wedge \{J\} \alpha \{J\}$	$\Rightarrow \{I \vee J\} \alpha \{I \vee J\}$	🎉

where $\alpha \text{ inv } I$ is simply α with the annotated invariant I and it binds less than any other program operator, e.g. $\{P\} \text{ loop } \alpha \text{ inv } I \{Q\} = \{P\} (\text{loop } \alpha) \text{ inv } I \{Q\}$.

For automating VCG, the *wlp*-laws are preferable over the Hoare-style rules since the laws can be added to the proof assistant's simplifier which rewrites them automatically. However, when loops and ODEs are involved, we use the rules (h-whilei), (h-loopi) and (h-evoli). In particular, two workflows emerge for discharging ODEs. If Picard-Lindelöf holds, that is, if there is a unique solution to the system of ODEs and it is known, the law (wlp-flow) is the best choice. Otherwise, we employ the rule (h-evoli) if an invariant is known. See Section 6.2 for a procedure guaranteeing the existence of flows or Section 5.3 for a procedure determining invariance for evolution commands.

Example 3. We prove that $I s \Leftrightarrow \text{get}_g s \geq 0$, or simply $g \geq 0$, is an invariant for the program $\text{blood_sugar} = \text{loop}(\text{ctrl}; \text{dyn})$ from Example 2. That is, we show that $\{I\} \text{blood_sugar} \{I\}$. We start applying (*h-loopi*) and proceed with wlp-laws:

$$\begin{aligned} & \{I\} \text{loop}(\text{ctrl}; \text{dyn}) \text{inv } I \{I\} \\ & \Leftrightarrow (I \Rightarrow I) \wedge (I \Rightarrow |\text{ctrl}; \text{dyn}| I) \wedge (I \Rightarrow I) \\ & = (\forall s. I s \Rightarrow |\text{if } g \leq g_m \text{ then } g := g_M \text{ else skip}| g' = -g| I s) \\ & = (\forall s. I s \Rightarrow (g \leq g_m \Rightarrow |g := g_M| g' = -g| I s) \wedge (g > g_m \Rightarrow |g' = -g| I s)), \end{aligned}$$

where the first equality applies (*wlp-seq*) and unfolds the definition of *ctrl* and *dyn*. The second follows by (*wlp-cond*). Next, given that $\varphi t c = c \cdot e^{-t}$ is the flow for $g' = -g$:

$$|g' = -g| I s \Leftrightarrow (\forall t \geq 0. I[\varphi t c/g]) \Leftrightarrow (\forall t \geq 0. g \cdot e^{-t} \geq 0) \Leftrightarrow g \geq 0,$$









for all $s \in \mathcal{S}$ by (*wlp-flow*), the lens laws, and because $G = \top$ and $k \cdot e^{-t} \geq 0 \Leftrightarrow k \geq 0$. Thus, the conjuncts above simplify to

$$\begin{aligned} g \leq g_m & \Rightarrow (|g' = -g| I)[g_M/g] & g > g_m & \Rightarrow |g' = -g| I s \\ & = g \leq g_m \Rightarrow (g \geq 0)[g_M/g] & & = g > g_m \Rightarrow g \geq 0 \\ & = (g \leq g_m \Rightarrow g_M \geq 0) = \top, & & = \top, \end{aligned}$$

by (*wlp-assign*), and because $g_m, g_M \geq 0$. Thus, $(I \Rightarrow |\text{ctrl}; \text{dyn}| I) = \top$. □

3.5.3 Forward diamonds

Here we add to our VCG approach by including forward diamonds in our verification framework. Our VCG laws from Sections 3.5.1 and 3.5.2 help users prove partial correctness specifications. Yet, our approach is generic and extensible and can cover other types of specifications [33, 40, 77]. For instance, we have integrated the *forward diamond* $|\rightarrow\rangle$ – predicate transformer, defined as $|\alpha\rangle Q s \Leftrightarrow (\exists s'. s' \in \alpha s \wedge Q s')$. It holds if there is a Q -satisfying state of α reachable from s . Due to their semantics, forward diamonds enable us to reason about progress and reachability properties. In applications, this implies that our tool supports proofs describing worst and best case scenarios stating that the modelled system can evolve into an undesired/desired state. See Section 8 for an example showing progress for a dynamical system. We formalise and prove the forward diamonds laws below which are also direct consequences of the duality law $|\alpha\rangle Q = \neg|\alpha\rangle \neg Q$. The example immediately after them merely illustrates their seamless application.

(fdia-skip)	$ \text{skip}\rangle Q$	$= Q$	
(fdia-abort)	$ \text{abort}\rangle Q$	$= \perp$	
(fdia-test)	$ \text{!}P?\rangle Q$	$= P \wedge Q$	
(fdia-assign)	$ x := e\rangle Q$	$= Q[e/x]$	
(fdia-seq)	$ \alpha; \beta\rangle Q$	$= \alpha\rangle \beta\rangle Q$	
(fdia-choice)	$ \alpha \sqcap \beta\rangle Q$	$= \alpha\rangle Q \vee \beta\rangle Q$	
(fdia-loop)	$ \text{loop } \alpha\rangle Q$	$= \exists n. \alpha^n\rangle Q$	
(fdia-cond)	$ \text{if } T \text{ then } \alpha \text{ else } \beta\rangle Q$	$= (T \wedge \alpha\rangle Q) \vee (\neg T \wedge \beta\rangle Q)$.	

Additionally, the (informal) diamond law for evolution commands is

$$\text{(fdia-evol)} \quad |(x' = f \& G)_U^{t_0}\rangle Q s \Leftrightarrow \exists X \in \text{ivp-sols } U f t_0 s. \exists t \in U s. (\forall \tau \in t \downarrow_{U s}. G(X \tau)) \Rightarrow Q(X t). \quad \text{🤖}$$

and the corresponding law for flows $\varphi : T \rightarrow \mathcal{C} \rightarrow \mathcal{C}$ of f and $U = T = \mathbb{R}_{\geq 0}$ is

$$\text{(fdia-flow)} \quad |x' = f \ \& \ G \rangle Q \ s \Leftrightarrow (\exists t \geq 0. (\forall \tau \in [0, t]. G(\varphi_s \tau)) \Rightarrow Q(\varphi_s t)). \quad \text{🌟}$$

Example 4. A similar argument as that in Example 3 allows us to prove the inequality $I \Rightarrow |\text{blood_sugar} \rangle I$ where $I \ s \Leftrightarrow (g \geq 0)$ and $\text{blood_sugar} = \text{loop}(\text{ctrl}; \text{dyn})$. Namely, we observe that by (fdia-evol), the forward diamond of $g' = -g$ and I becomes

$$|g' = -g \rangle I \ s \Leftrightarrow (\exists t \geq 0. (I[g \cdot e^{-t}/g])) \Leftrightarrow (\exists t \geq 0. g \cdot e^{-t} \geq 0) \Leftrightarrow g \geq 0,$$

for all $s \in \mathcal{S}$. Hence, the conjuncts below simplify as shown:

$$\begin{aligned} g \leq g_m \wedge (|g' = -g \rangle I)[g_M/g] & & g_s > g_m \wedge |g' = -g \rangle I \ s \\ = g \leq g_m \wedge (g \geq 0)[g_M/g] & & = g_s > g_m \wedge g_s \geq 0 \\ = g \leq g_m \wedge g_M \geq 0 = g \leq g_m, & & = g_s > g_m. \end{aligned}$$

Therefore, by backward reasoning with the diamond laws, we have

$$\begin{aligned} I &\Rightarrow |\text{loop}(\text{ctrl}; \text{dyn}) \text{ inv } I \rangle I \\ &\Leftarrow I \Rightarrow |\text{ctrl}; \text{dyn} \rangle I \\ &= (\forall s. I \ s \Rightarrow |\text{if } g \leq g_m \text{ then } g := g_M \text{ else skip} \rangle |g' = -g \rangle I \ s) \\ &= (\forall s. I \ s \Rightarrow (g \leq g_m \wedge |g := g_M \rangle |g' = -g \rangle I \ s) \vee (g > g_m \wedge |g' = -g \rangle I \ s)) \\ &= (\forall s. I \ s \Rightarrow g \leq g_m \vee g > g_m) = \top, \end{aligned}$$

where the first implication follows by a rule analogous to (h-loopi) for diamonds. 🌟

Thus, we have shown that $I \Rightarrow |\text{blood_sugar} \rangle I$. □

We have summarised our approach to hybrid systems verification in general purpose proof assistants [27, 40]. This is the basis for describing our contributions for the rest of this article, included among them, the formalisation of forward diamonds into IsaVODEs. Although a similar formalisation has been done before [77], our implementation is more automated due to its use of standard types, e.g. Isabelle predicates ($\mathcal{S} \rightarrow \mathbb{B}$), that have had more support over time. Thus, our formalisation increases the proof capabilities of our Isabelle-based framework and its expressivity, since the forward diamonds enable us to assert the progress of hybrid programs. Other extensions to our framework not described here are the addition of dL's nondeterministic assignments and their corresponding partial correctness and progress laws, as well as the formalisation of variant-based rules on the reachability of finite iterations and while-loops. See previous work [9, 33] for examples with while-loops that our verification framework could tackle. 🌟

4 Hybrid Modelling Language

Here, we describe our implementation of a hybrid modelling language, which takes advantage of lenses and Isabelle's flexible syntax processing features. Beyond the advantages already mentioned, lenses enhance our hybrid store models in several ways. They allow us to model frames—sets of mutable variables—and thus support local reasoning. They also allow us to project parts of the global store onto vector spaces to describe continuous dynamics. These projections can be constructed using three lens combinators: composition, sum and quotient.

The projections particularly allow us to use hybrid state spaces, consisting of both continuous components with a topological structure (e.g. \mathbb{R}^n), and discrete components using Isabelle's flexible type system. This in turn allows our tool to support more general software engineering notations, which typically make use of object-oriented data structures [57]. Moreover, the projections allow us to reason locally about the continuous variables, since discrete variables are outside of the frame during continuous evolution.

4.1 Dataspaces

Most modelling and programming languages support modules with local variables, constant declarations, and assumptions. We have implemented an Isabelle command that automates the creation of hybrid stores, which provide a local theory context for hybrid programs. We call these *dataspaces*, since they are state spaces that can make use of rich data structures in the program variables. 🌈

```
dataspace store = [parent_store +]
  constants c1::C1 ... cn::Cn
  assumes a1:P1 ... an:Pn
  variables x1::T1 ... xn::Tn
```

A dataspace has constants $c_i : C_i$, named constraints $a_i : P_i$ and state variables $x_i : T_i$. In its context, we can create local definitions, theorems and proofs, which are hidden, but accessible using its namespace. Internally, the dataspace command creates an Isabelle **locale** with fixed constants and assumptions, inspired by previous work by Schirmer and Wenzel [74]. Like locales, dataspaces support a form of inheritance, whereby constants, assumptions, and variables can be imported from an existing dataspace (e.g. `parent_store`) and extended with further constants, assumptions, and variables.

Each declared state variable is assigned a lens $x_i :: T_i \Rightarrow \mathcal{S}$, using the abstract store type \mathcal{S} with the lens axioms from Section 3.3 as locale assumptions. We also generate independence assumptions, e.g. $x_i \bowtie x_j$ for $x_i \neq x_j$, that distinguish different variables semantically [23]. Formally, $x \bowtie y$ if $put_x u \circ put_y v = put_y v \circ put_x u$ for all $u, v \in \mathcal{V}$. That is, two lenses are independent if their *put* operations commute on all states.

4.2 Lifted Expressions


As discussed in Section 3.3, expressions in our hybrid modelling language are modelled by functions of type $\mathcal{S} \rightarrow \mathcal{V}$. Assertions are therefore state predicates, or “expressions” where $\mathcal{V} = \mathbb{B}$. Discharging VCs requires showing that assertions hold for all states. For example, the law (**wlp-test**) requires us to prove a VC of the form $P \Rightarrow Q$, that is, $\forall s. Ps \Rightarrow Qs$. Also, if we have state variables x and y , then proving the assertion $x + y \geq x$ corresponds to proving the HOL predicate $get_x s + get_y s \geq get_x s$ for some arbitrary-but-fixed state s , which can readily be discharged using one of Isabelle’s proof methods (*simp*, *auto* etc.). This process is automated by methods *expr-simp* and *expr-auto*.

Nevertheless, there remains a gap between the syntax used in typical programming languages and its semantic representation. Namely, users would prefer writing $x^2 + y^2 \leq c^2$ over $\lambda s. (get_x s)^2 + (get_y s)^2 \leq c^2$, and so, the main technical challenge is to seamlessly transform between the two. This can be achieved using Isabelle’s syntax pipeline, which significantly improves the usability of our tool.

Isabelle’s multi-stage syntax pipeline parses Unicode strings and transforms them into “pre-terms” [46]: elements of the ML `term` type containing syntactic constants. These must be mapped to previously defined semantic constants by syntax translations, before they can be checked and certified in the Isabelle kernel. Printing reverses this pipeline, mapping terms to strings.


We automate the translation between the expression syntax (pre-terms) and semantics using parse and print translations implemented in Isabelle/ML, as part of our Shallow-Expressions component. It lifts pre-terms by replacement of free variables and constants, and insertion of store variables (s) and λ -binders. Its implementation uses the syntactic annotation $(t)_e$ to lift the syntactic term t to a semantic expression in the syntax translation rules. The syntax translation is described by the following equations:

$$(t)_e \equiv [(t)_e]_e, \quad (n)_e \equiv \begin{cases} \lambda s. get_n s & \text{if } n \text{ is a lens,} \\ \lambda s. n s & \text{if } n \text{ is an expression,} \\ \lambda s. n & \text{otherwise,} \end{cases} \quad (f t)_e \equiv \lambda s. f ((t)_e s),$$

where $p \rightleftharpoons q$ means that pre-term p is translated to term q , and q printed as p . Moreover, $[-]_e$ is a constant that marks lifted expressions that are embedded in terms. When the translation encounters a name n (i.e. a free variable or constant), it checks whether n has a definition in the context. If it is a lens (i.e. $n :: \mathcal{V} \Rightarrow \mathcal{S}$), then it inserts a *get*. If it is an expression (i.e. $n : \mathcal{S} \rightarrow \mathcal{V}$), then it is applied to the state. Otherwise, it leaves the name unchanged, assuming it to be a constant. Function applications are also left unchanged by \rightleftharpoons . For instance, $((x + y)^2/z)_e \rightleftharpoons [\lambda s. (get_x s + get_y s)^2/z]_e$ for variables (lenses) x and y and constant z . Once an expression has been processed, the resulting λ -term is enclosed in $[-]_e$. The pretty printer can then recognise a lifted term and print it. This process is fully automated, so that users see only the sugared expression, without the λ -binders, in both the parser and terms' output during the proving process. 

4.3 Substitutions

In our semantic approach, substitutions correspond to functions $\sigma : \mathcal{S} \rightarrow \mathcal{S}$ on the store \mathcal{S} . This interpretation allows us to denote updates as a sequence of variable assignments. That is, instead of directly manipulating the store $s : \mathcal{S}$ with the lens functions, we provide more user-friendly program specifications with the notation $\sigma(x \rightsquigarrow e) = \lambda s. put_x(e s)(\sigma s)$. It allows us to describe assignments as sequences of updates: $[x_1 \rightsquigarrow e_1, x_2 \rightsquigarrow e_2, \dots] = id(x_1 \rightsquigarrow e_1)(x_2 \rightsquigarrow e_2) \dots$, for variable lenses $x_i :: \mathcal{V}_i \Rightarrow \mathcal{S}$ and “expressions” $e_i : \mathcal{S} \rightarrow \mathcal{V}_i$.

Implicitly, any variable y not mentioned in such a substitution is left unchanged: $y \rightsquigarrow y$. We further write $e[v/x] = e \circ [x \rightsquigarrow v]$, for $x :: \mathcal{V} \Rightarrow \mathcal{S}$, $e : \mathcal{S} \rightarrow \mathcal{V}'$, and $v : \mathcal{S} \rightarrow \mathcal{V}$, for the application of substitutions to expressions. This yields standard notations for program specifications, e.g. $(x := e) = \langle [x \rightsquigarrow e] \rangle$ and $w/p \langle [x \rightsquigarrow e] \rangle Q = Q[e/x]$. Using an Isabelle simplification procedure (a “simproc”), the simplifier can reorder assignments alphabetically according to their variable name, and otherwise reduce and evaluate substitutions during VCG [23]. We can extract assignments for x writing $\langle \sigma \rangle_s x = get_x \circ \sigma$ so that, e.g. $\langle [x \rightsquigarrow e_1, y \rightsquigarrow e_2] \rangle_s x$ reduces to e_1 when $x \bowtie y$. 

Example 5. We continue our blood glucose running example and formalise Example 2. First, we declare our problem variables and assumptions via our **dataspace** command. We name this *dataspace* *glucose*, and assume that there is a minimal warning threshold $g_m > 0$ and a maximum dosage $g_M > g_m$. The patient’s glucose is represented via the “continuous” variable g .

```

dataspace glucose =
  constants gm :: real gM :: real
  assumes ge_0: "gm > 0" and ge_gm: "gM > gm"
  variables g :: real

```

Next, inside the *glucose* context we declare, via Isabelle’s **abbreviation** command, the definition of the controller and the dynamics. Our shallow expressions hide the lens infrastructure and, from the user’s perspective, the definitions are Isabelle abbreviations. Notice also, that our recently introduced “substitution” notation allows us to explicitly specify the flow’s behaviour on the continuous variable g . It also occurs implicitly in our declaration of the differential equation $g' = g$ (see Section 5.2).

```

context glucose
begin

abbreviation "ctrl  $\equiv$  IF g  $\leq$  gm THEN g ::= gM ELSE skip"

abbreviation "dyn  $\equiv$  {g' = -g}"

abbreviation "flow  $\tau \equiv$  [g  $\rightsquigarrow$  g * exp (-  $\tau$ )]"

abbreviation "blood_sugar  $\equiv$  LOOP (ctrl; dyn) INV (g  $\geq$  0)"

```


end

Thus, our lens integrations provide a seamless way to formalise hybrid system verification problems in Isabelle. We explore their verification condition generation in Section 6.2. \square

4.4 Vectors and matrices

Vectors and matrices are ubiquitous in engineering applications and users of our framework would appreciate using familiar concepts and notations to them. This is possible due to our modelling language. In particular, vectors are supported by HOL-Analysis using finite Cartesian product types, (A, n) `vec` with the notation A^n . Here, A is the element type, and n is a numeral type denoting the dimension. The type of vectors is isomorphic to $[n] \rightarrow A$ where $[n] = \{1, \dots, n\}$. A matrix is simply a vector of vectors, $A^m \times A^n$, hence a map $[m] \rightarrow [n] \rightarrow A$. Building on this, we supply notation $[[x_{11}, \dots, x_{1n}], \dots, [x_{m1}, \dots, x_{mn}]]$ for matrices and means for accessing coordinates of vectors via hybrid program variables [24]. This notation supports the inference of vector and matrices' dimensions conveyed by the type variables.

Vectors and matrices are often represented as composite objects consisting of several values, e.g. $p = (p_x, p_y) \in \mathbb{R}^2$. When writing specifications, it is often convenient to refer to and manipulate these components individually. We can denote such variables using component lenses and the lens composition operator. We write $x_1 \circledast x_2 :: \mathcal{S}_1 \Rightarrow \mathcal{S}_3$, for $x_1 :: \mathcal{S}_1 \Rightarrow \mathcal{S}_2$, $x_2 :: \mathcal{S}_2 \Rightarrow \mathcal{S}_3$, for the forward composition and $1_{\mathcal{S}} :: \mathcal{S} \Rightarrow \mathcal{S}$ for the units in the lens category, but do not show formal definitions [23]. Intuitively, the composition \circledast selects part of a larger store as illustrated below.

We model vectors in \mathbb{R}^n as part of larger hybrid stores, lenses $v :: \mathbb{R}^n \Rightarrow \mathcal{S}$, and project onto coordinate $v_k :: \mathbb{R} \Rightarrow \mathcal{S}$ using lens composition and a *vector lens* $\Pi(i) :: \mathbb{R} \Rightarrow \mathbb{R}^n$:

$$\begin{aligned} \Pi(i) &= (\text{get}_{\Pi(i)}, \text{put}_{\Pi(i)}), \text{ where} \\ \text{get}_{\Pi(i)} &= (\lambda s. \text{vec-nth } s \ i), \\ \text{put}_{\Pi(i)} &= (\lambda v \ s. \text{vec-upd } s \ i \ v), \end{aligned}$$

and $i \in [n] = \{1, \dots, n\}$. The lookup function $\text{vec-nth} : A^n \rightarrow [n] \rightarrow A$ and update function $\text{vec-upd} : A^n \rightarrow [n] \rightarrow A \rightarrow A^n$ come from HOL-Analysis and satisfy the lens axioms (Section 3.3). Then, as an example, $p_x = \Pi(1) \circledast p$ and $p_y = \Pi(2) \circledast p$ for $p :: \mathbb{R}^2 \Rightarrow \mathcal{S}$, using \circledast to first select the variable p and then the vector-part of the hybrid store. Intuitively, two vector elements are independent, $\Pi(i) \bowtie \Pi(j)$ iff they have different indices, $i \neq j$.

Example 6. To illustrate the use of vector variables, we model the dynamics and a controller for an autonomous boat. We refer readers to previous publications for the verification of an invariant for this system [24, 27]. The boat is manoeuvrable in \mathbb{R}^2 and has a rotatable thruster generating a positive propulsive force f with maximum f_{max} . The boat's state is determined by its position $p = (p_x, p_y)$, velocity $v = (v_x, v_y)$, and acceleration $a = (a_x, a_y)$. We describe this state with the following *dataspace*:

dataspace `AMV =`

```

constants S::ℝ f_max::ℝ assumes fmax:"f_max ≥ 0"
variables p::"ℝ vec[2]" v::"ℝ vec[2]" a::"ℝ vec[2]" φ::ℝ s::ℝ
wps::"(ℝ vec[2]) list" org::"(ℝ vec[2]) set" rs::ℝ rh::ℝ

```

This store model combines discrete and continuous variables and uses the alternative notation $\mathbb{R} \text{ vec}[n]$ for a real-valued vector of dimension n . The **dataspace** specifies a variable for linear speed s , and a constant S for the boat's maximum speed. We also provide discrete variable `wps` for a list of points to pass through in the vehicle's path (way-point path), `org` for a set of points where


obstacles are located (obstacle register), and the requested speed and heading (rs and rh). Our **dataspace** allows us to declare variables $p, v, a : \mathbb{R} \text{ vec}[2]$ and manipulate them using operations for vectors (see Section 5.2). \square


5 Local Reasoning


In this section, we describe our framework's support for local reasoning, which allows us to consider only parts of the state that are changed by a component in the verification. This improves the scalability of our approach, since we can decompose verification tasks into smaller manageable tasks, in an analogous way to separation logic [70]. We show how lenses can be used to characterise a program's frame: the set of variables which may be modified. We then explain how frames extend to evolution commands, such that variables with no derivative (or derivative 0) are outside of the frame. Next, we develop a framed version of differentiation, called *framed Fréchet derivatives*, which allows us to perform local differentiation with respect to a strict subset of the store variables. This, in turn, supports a method, framed differential induction, for proving invariants in the continuous part of the state space. Finally, we introduce a corresponding implementation of $d\mathcal{L}$'s differential ghost rule [65] that augments systems of ODEs with fresh equations to aid invariant reasoning. This rule likewise supports frames.

5.1 Frames

Lenses support algebraic manipulations of variable frames. A *frame* is the set of variables that a program is permitted to change. Variables outside of the frame are immutable. We first show how variable sets can be modelled via lens sums. Then we recall a predicate characterising immutable program variables [26]. Most importantly, we derive a frame rule à la separation logic for local reasoning with framed variables.

Variable lenses $x_1 :: \mathcal{V}_1 \Rightarrow \mathcal{S}$ and $x_2 :: \mathcal{V}_2 \Rightarrow \mathcal{S}$ can be combined into lenses for variable sets with *lens sum* [23], $x_1 \oplus x_2 :: \mathcal{V}_1 \times \mathcal{V}_2 \Rightarrow \mathcal{S}$ if $x_1 \bowtie x_2$ via $get_{x_1 \oplus x_2}(s_1, s_2) = (get_{x_1} s_1, get_{x_2} s_2)$ and $put_{x_1 \oplus x_2}(v_1, v_2) = put_{x_1} v_1 \circ put_{x_2} v_2$. This combines two independent lenses into a single lens with a product view. It can be used to model composite variables, for example, $(x \oplus y) := (e, f)$ is a simultaneous assignment to x and y . We can decompose such a composite update into two atomic updates, with $[(x, y) \rightsquigarrow (e_1, e_2)] = [x \rightsquigarrow e_1, y \rightsquigarrow e_2]$. We can also use lens sums to model finite sets, for example $\{x, y, z\}$ is modelled as $x \oplus (y \oplus z)$. Each variable in such a sum may have a different type, e.g. $\{v_x, \vec{p}\}$ is a valid and well-typed construction. 

Lens sums are only associative and commutative up-to isomorphism of cartesian products. We need heterogeneous orderings and equivalences between lenses to capture this. We define a *lens preorder* [23], $x_1 \preceq x_2 \Leftrightarrow \exists x_3. x_1 = x_3 \circledast x_2$ that captures the part-of relation between $x_1 :: \mathcal{V}_1 \Rightarrow \mathcal{S}$ and $x_2 :: \mathcal{V}_2 \Rightarrow \mathcal{S}$, e.g. $v_x \preceq \vec{v}$ and $\vec{p} \preceq \vec{p} \oplus \vec{v}$. *Lens equivalence* $\cong = \preceq \cap \succeq$ then identifies lenses with the same shape in the store. Then, for variable set lenses up-to \cong , \oplus models \cup , \bowtie models \notin , and \preceq models \subseteq or \in . Since $x_1 \preceq x_1 \oplus x_2$ and $x_1 \oplus x_2 \cong x_2 \oplus x_1$, with our variable set interpretation, we can show, e.g., that $x \in \{x, y, z\}$, $\{x, y\} \subseteq \{x, y, z\}$, and $\{x, y\} = \{y, x\}$. Hence we can use these lens combinators to construct and reason about variable frames. 

We can use variable set lenses to capture the frame of a program. Let $A :: \mathcal{V} \Rightarrow \mathcal{S}$ be a lens modelling a variable set. For $s_1, s_2 \in \mathcal{S}$ let $s_1 \approx_A s_2$ hold if $s_1 = s_2$ up-to the values of variables in A , that is $get_A s_1 = get_A s_2$. Local reasoning within A uses the *lens quotient* [22] $x // A$, which localises a lens $x :: \mathcal{V} \Rightarrow \mathcal{S}$ to a lens $\mathcal{V} \Rightarrow \mathcal{C}$. Assuming $x \preceq A$, it yields $x_1 :: \mathcal{V} \Rightarrow \mathcal{C}$ such that $x = x_1 \circledast A$. For example, $p_x // p = \Pi(1)$ with $\mathcal{C} = \mathbb{R}^n$. 

We can also use lenses to describe when a variable does not occur freely in an expression or predicate with the unrestriction property: $A \# e \Leftrightarrow \forall v. e \circ (put_A v) = e$ [23]. A variable x is unrestricted in e , written $x \# e$, provides that e does not semantically depend on x for its evaluation. For example, $x \# (y + 1)$, when $x \bowtie y$, since $y + 1$ does not mention x . We also define $(-A) \# e \Leftrightarrow$

$\forall s_1 s_2 v. e(\text{put}_A v s_1) = e(\text{put}_A v s_2)$ as the converse, which requires that e does not depend on variables outside of A .

Next, we capture the non-modification of variables by a program. For $\alpha : \mathcal{S} \rightarrow \mathcal{P}\mathcal{S}$ and an expression (or predicate) e we define $\alpha \text{ nmods } e \Leftrightarrow (\forall s_1 \in \mathcal{S}. e(s_1) = e(s_2))$, which describes when e does not depend on the mutable variable of α . The expression e can characterise a set of variables giving the set of immutable variables. For example, we have it that $(x := x + 1) \text{ nmods } (y, z)$, when $x \boxtimes y$ and $x \boxtimes z$, since this assignment changes only x and no other variables.

Intuitively, non-modification $\alpha \text{ nmods } x$, where x is a variable lens, is equivalent to the specification for $\{x = v\} \alpha \{x = v\}$ for fresh logical variable v . This means that x retains its initial value in any final state of α . We prove the following laws for non-modification:

$$\frac{A \# x}{(x := e) \text{ nmods } A} \quad \frac{-}{\text{!}P? \text{ nmods } A} \quad \frac{\alpha \text{ nmods } A \quad \beta \text{ nmods } A}{(\alpha \circledast \beta) \text{ nmods } A}$$

$$\frac{\alpha \text{ nmods } A \quad \beta \text{ nmods } A}{(\alpha \sqcap \beta) \text{ nmods } A} \quad \frac{\alpha \text{ nmods } A}{\alpha^* \text{ nmods } A} \quad \frac{\alpha \text{ nmods } B \quad A \preceq B}{\alpha \text{ nmods } A}$$

The variables in A are immutable for assignment $x := e$ provided x is not in A . A test $\text{!}P?$ modifies no variables, and therefore any set A is immutable. For the programming operators, non-modification is inherited from the parts. The final law shows that we can always shrink the specified set of immutable variables.

With these concepts in place, we derive two frame rules for local reasoning: 

$$\frac{\alpha \text{ nmods } I \quad \{P\} \alpha \{Q\}}{\{P \wedge I\} \alpha \{Q \wedge I\}} \quad \frac{\alpha \text{ nmods } A \quad (-A) \# I \quad \{P\} \alpha \{Q\}}{\{P \wedge I\} \alpha \{Q \wedge I\}}$$

If program α does not modify any variables mentioned in I , then I can be added as an invariant of α . In the first law, non-modification is checked directly of the variables used by I . In the second, which is an instance of the first, we instead infer the immutable variables of A and check that I does not depend on variables outside of A . With these laws, we can import invariants for a program fragment that refer to only those variables that are left unchanged. This allows us to perform modular verification, whereby we need only consider invariants of variables used in a component. In the following section, we show how this can be applied to systems of ODEs.

5.2 Framed evolution commands


We extend previous components [40] for continuous dynamics with function framing techniques that project onto parts of the store. That is, we formally describe the implementation of the evolution command state transformer using the lens infrastructure described so far [27]. Specifically, we use framing to derive continuous vector fields ($\mathcal{C} \rightarrow \mathcal{C}$) and flows from state-wide “substitutions” ($\mathcal{S} \rightarrow \mathcal{S}$). We also add a non-modification rule for evolution commands. This supports local reasoning where evolution commands modify only continuous variables and leave discrete ones—outside a frame—unchanged.

Framing uses the second interpretation of lenses where the frame \mathcal{C} is a subregion of \mathcal{S} that we can access through $x :: \mathcal{C} \Rightarrow \mathcal{S}$. We view the store as divided into its continuous \mathcal{C} and discrete parts and localise continuous variables to the former. The continuous part must have sufficient topological structure to support derivatives and is thus restricted to certain type constructions like normed vector spaces or the real numbers. However, the discrete part may use any type defined in HOL. With this view, we can use get_x and put_x to lift entities defined on \mathcal{C} or project those in \mathcal{S} . For instance, given any $s \in \mathcal{S}$ and a predicate $G : \mathcal{S} \rightarrow \mathbb{B}$ (like the guards in evolution commands), there is a


corresponding restriction $G\downarrow_x^s : \mathcal{C} \rightarrow \mathbb{B}$ such that $G\downarrow_x^s \vec{c} \Leftrightarrow G[\vec{c}/x] \Leftrightarrow G(\text{put}_x \vec{c} s)$. Conversely, for $s \in \mathcal{S}$ and X , a set of vectors in \mathcal{C} , the set $X\uparrow_x^s = \mathcal{P}(\lambda \vec{c}. \text{put}_x \vec{c} s) X$ has values in \mathcal{S} .

More importantly, we can specify ODEs and flows via time-dependent deterministic functions (Section 4.3's substitutions). Given a lens $x :: \mathcal{C} \Rightarrow \mathcal{S}$ from global store \mathcal{S} onto local continuous store \mathcal{C} and $s \in \mathcal{S}$, we can turn any state-wide function $f : T \rightarrow \mathcal{S} \rightarrow \mathcal{S}$ into a vector field $f\downarrow_x^s : T \rightarrow \mathcal{C} \rightarrow \mathcal{C}$ by framing it via $f\downarrow_x^s t \vec{c} = \text{get}_x((f t)(\text{put}_x \vec{c} s))$.

Example 7. Suppose $\mathcal{S} = \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \times \mathcal{S}'$ and $p, v, a :: \mathbb{R}^2 \Rightarrow \mathcal{S}$. The variable set lens $A = (p \oplus v \oplus a) :: \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \Rightarrow \mathcal{S}$ frames the continuous part of the state space \mathcal{S} . The substitution $f : T \rightarrow \mathcal{S} \rightarrow \mathcal{S}$ such that $f t = [p \rightsquigarrow v, v \rightsquigarrow a, a \rightsquigarrow 0]$ then behaves as the identity function on \mathcal{S}' and becomes the vector field $f\downarrow_A^s : T \rightarrow \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2$. Hence, f naturally describes the ODEs $p' t = v t, v' t = a t, a' t = 0$ after framing. \square


Using the previously described liftings and projections, we formally define the semantics of evolution commands. For this, we only need to lift the definition of generalised guarded orbits maps (Section 3) on the continuous \mathcal{C} to the larger space \mathcal{S} . Thus, for substitution $f : T \rightarrow \mathcal{S} \rightarrow \mathcal{S}$, predicate $G : \mathcal{S} \rightarrow \mathbb{B}$, interval function $U : \mathcal{C} \rightarrow \mathcal{P} \mathbb{R}$, and $t_0 \in \mathbb{R}$ the state transformer $\mathcal{S} \rightarrow \mathcal{P} \mathcal{S}$ modelling evolution commands is $(x' = f \& G)_U^{t_0} s = (\gamma^{f\downarrow_x^s} (G\downarrow_x^s) U t_0 x_s)\uparrow_x^s$, or equivalently 

$$(x' = f \& G)_U^{t_0} s = \left\{ \text{put}_x (X t) s \mid \begin{array}{l} t \in U x_s \wedge X \in \text{ivp-sols } U (f\downarrow_x^s) t_0 x_s \\ \wedge \mathcal{P} X (t\downarrow_U x_s) \subseteq G\downarrow_x^s \end{array} \right\},$$

where we abbreviate $\text{get}_x s$ with x_s . That is, evolution commands are state transformers that output those states whose discrete part remains unchanged from s but whose continuous part changes according to the ODEs' solutions within G . With this, the law (**wlp-evol**) formally becomes 

$$[(x' = f \& G)_U^{t_0}] Q s \Leftrightarrow \begin{array}{l} \forall X \in \text{ivp-sols } U (f\downarrow_x^s) t_0 x_s. \forall t \in U x_s. \\ (\forall \tau \in t\downarrow_U x_s. G[X \tau/x]) \Rightarrow Q[X t/x]. \end{array}$$

This says that the postcondition Q holds after an evolution command $(x' = f \& G)_U^{t_0}$ for $s \in \mathcal{S}$ if every solution X to the IVP corresponding to (t_0, x_s) satisfies Q on every time t , provided G holds from the beginning of the interval $t_0 \in U x_s$ until t . Thus, VCG follows our description in Section 3: users must supply flows and evidence for Lipschitz continuity in order to obtain *wlps*. We provide tactics that automate these processes in Section 6.

We use Isabelle's syntax translations to provide a natural syntax for specifying evolution commands. Users can write $\{x'_1 = e_1, \dots, x'_n = e_n \mid G \text{ on } UV @ t_0\}$ directly into the prover where each $x_i :: \mathcal{V}_i \Rightarrow \mathcal{S}$ is a summand of the frame lens $x = \{x_1, \dots, x_n\} :: \mathcal{C} \Rightarrow \mathcal{S}$. Users can thus declare the ODEs in evolution commands coordinate-wise with lifted expressions $e_i : \mathcal{S} \rightarrow \mathcal{V}_i$. They can also omit the parameters G, U, V and t_0 which defaults them to $\top, \mathbb{R}_{\geq 0}, \mathcal{C}$ and 0 , respectively. If desired, they can also use product syntax $(x'_1, \dots, x'_n) = (e_1, \dots, e_n)$ or vector syntax $x' = e$, and specify evolution commands using flows instead of ODEs with the notation $\{\text{EVOL } x = e \tau \mid G\}$. 

With these, non-modification of variables naturally extends to ODEs with the law

$$\frac{x \# A}{\{x' = e \mid G\} \text{ nmods } A}$$

Specifically, any set of variables (A) without assigned derivatives in a system of ODEs is immutable. Then, by application of the frame rule, we can demonstrate that any assertion I that uses only variables outside of x is an invariant of the system of ODEs.


Example 8. We use the autonomous boat from Example 6 to illustrate the use of non-modification. A system of ODEs for the boat's state p, v, a may be specified as follows:

abbreviation "ODE $\equiv \{ p' = v, v' = a, a' = \mathbf{0}, \phi' = \omega, s' = \text{if } s \neq \mathbf{0} \text{ then } (v \cdot a) / s \text{ else } \|a\| \mid s *_R [[\sin(\phi), \cos(\phi)]] = v \wedge \mathbf{0} \leq s \wedge s \leq S \}$ "

We also write derivatives for ϕ and s . The derivative of the former is the angular velocity ω , which has the value $\arccos((v + a) \cdot v / (\|v + a\| \cdot \|v\|))$ when $\|v\| \neq 0$ and 0 otherwise [24]. The linear acceleration (s') is calculated using the inner product of v and a . If the current speed is 0, then s' is $\|a\|$. Immediately after the derivatives, we also specify the guard or boundary condition that serves to constrain the relationship between the velocity vector and the heading ϕ . The guard states that the velocity vector v is equal to s multiplied with the heading unit-vector using scalar multiplication ($*_R$) and our vector syntax. We also require that $0 \leq s \leq S$, i.e. that the linear speed is between 0 and the maximum speed.

All other variables in the store remain outside the evolution frame and do not need to be specified. In particular, notice that the ODE above does not mention the requested speed variable rs . This is a discrete variable that is unchanged during evolution. Therefore, we can show: $\text{ODE} \text{ nmods } rs$. Moreover, using the frame rule we can also demonstrate that $rs > 0$ is an invariant, i.e. $\{rs > 0\} \text{ ODE } \{rs > 0\}$ [27]. \square

5.3 Frames and invariants for ODEs

As discussed in Section 3, an alternative to using flows for verification of evolution commands is finding and certifying invariants for them. Mathematically, evolution commands' invariants coincide with *invariant sets* for dynamical systems or $d\mathcal{L}$'s *differential invariants* [40]. We abbreviate the statement "I is an invariant for the evolution command $(x' = f \& G)_{\mathcal{U}}^{t_0}$ " with the notation $\text{diff-inv } x f G U t_0 I$. In terms of Hoare logic, invariants for evolution commands satisfy 

$$\text{diff-inv } x f G U t_0 I \Leftrightarrow \{I\} (x' = f \& G)_{\mathcal{U}}^{t_0} \{I\}.$$

Informally, $\text{diff-inv } x f G U t_0 I$ asserts that all states in the generalised guarded orbit $(x' = f \& G)_{\mathcal{U}}^{t_0} s$ of $s \in \mathcal{S}$ such that $I s$, will also satisfy I . In dynamical systems parlance, the orbits of the system of ODEs within the region characterised by I remain within I .

A common approach in hybrid program verification for certifying invariants for evolution commands is *differential induction* [63]. It establishes sufficient conditions for guaranteeing that simple predicates, such as (in)equalities, are invariants. From these, more complex predicates like conjunctions or disjunctions of these (in)equalities can be shown to be invariants using the rules (**h-conji**) and (**h-disji**).

Example 9. To prove that the conjunction $x > c \wedge y \geq x$ is an invariant of the pair of ODEs $x' = 1, y' = 2$ with $c \in \mathbb{R}$ (a constant) we need to show that

$$\{x > c \wedge y \geq x\} (x' = 1, y' = 2) \{x > c \wedge y \geq x\}.$$


An application of the rule (**h-conji**) yields the two proof obligations

$$\{x > c\} (x' = 1, y' = 2) \{x > c\}, \text{ and } \{y \geq x\} (x' = 1, y' = 2) \{y \geq x\}.$$

We conclude the proof informally to provide an intuition for how to proceed:

Since the derivative of x is greater than 0, its magnitude is increasing. Hence, for all time $t \geq 0$, the value of $x t$ is greater or equal to its original value $x 0 > c$. This means that the "values" of x remain above c . Similarly, since the derivative of y is greater than that of x and positive, y "grows" faster than x . Hence, the value of y remains greater or equal to that of x . Thus, the predicate $x > c \wedge y \geq d$ is an invariant of $x' = 1, y' = 2$. \square

Formally, if a predicate I is in negation normal form (NNF) in a first-order language for the real numbers $\mathcal{L}_{\mathbb{R}}\langle 0, 1, +, -, \cdot, <, \leq \rangle$, to show that it is an invariant, we can apply the rules (**h-conji**) and (**h-disji**) until the only remaining proof obligations are Hoare triples of literals. The negated literals can also be converted into positive ones via the equivalences $\neg(x < y) \Leftrightarrow y \leq x$, $\neg(x \leq y) \Leftrightarrow y < x$, and $\neg(x = y) \Leftrightarrow (y < x \vee x < y)$. The remaining proof obligations can be discharged by analysing the derivatives of the magnitudes represented in them as done in Example 9. In the sequel, we present the theory to do this analysis formally in our setting.


The discussion in Example 9 compares derivatives of expressions depending on the ODEs' variables. In our semantic approach, the system of ODEs is modelled by a function $f t : \mathcal{S} \rightarrow \mathcal{S}$ that becomes a vector field ($\mathcal{C} \rightarrow \mathcal{C}$) after framing via some $x :: \mathcal{C} \Rightarrow \mathcal{S}$, where \mathcal{C} is the continuous part of \mathcal{S} . Similarly, our "expressions" are really functions $e : \mathcal{S} \rightarrow \mathcal{U}$ (see Section 3.3), and we can assume that \mathcal{U} is a continuous state space to get "continuous expressions". We frame these functions to the continuous part \mathcal{C} of \mathcal{S} to obtain our *framed expressions* $e \downarrow_s^x : \mathcal{C} \rightarrow \mathcal{U}$ such that $e \downarrow_s^x = e \circ (\lambda \vec{c}. \text{put}_x \vec{c} s)$. We wish to "differentiate" these expressions as informally done in Example 9. Hence, for a general and formal treatment of our semantic entities, we use the Fréchet derivative of these framed expressions. More specifically, recall that, if a function $F : \mathcal{C} \rightarrow \mathcal{U}$ between normed spaces \mathcal{C}, \mathcal{U} is Fréchet differentiable at \vec{c} , the Fréchet derivative of F at \vec{c} is the bounded linear operator $D F \vec{c} : \mathcal{C} \rightarrow \mathcal{U}$ that attests this. In the finite-dimensional case, e.g. $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with $m, n \in \mathbb{N}$, the Fréchet derivative $D F \vec{c}$ is the Jacobian. It is well-known that if \vec{e}_i is the i -th unit vector of the canonical ordered base, the function $\lambda \vec{x}. (D F \vec{x}) \vec{e}_i$ provides the i th partial derivatives of F while the directional derivative of F in the direction of \vec{c} is $\lambda \vec{x}. (D F \vec{x}) \vec{c}$. With these ideas in mind, we define our *framed Fréchet derivatives* $\mathcal{D}_x^f e : \mathcal{S} \rightarrow \mathcal{U}$ of expression $e : \mathcal{S} \rightarrow \mathcal{U}$ in the direction of $f : \mathcal{S} \rightarrow \mathcal{S}$ with respect to $x :: \mathcal{C} \Rightarrow \mathcal{S}$ as [27] 

$$(\mathcal{D}_x^f e) s = (D e \downarrow_s^x (\text{get}_x s)) (\text{get}_x (f s)).$$

That is, they are the directional derivatives of framed expressions $e \downarrow_s^x$ in the direction of the projection of f onto the continuous space \mathcal{C} . These framed Fréchet derivatives capture the intuitive analysis performed in Example 9. In fact, the following rules are sound:

$$\begin{aligned} (\text{dinv-eq}) \quad & (G \Rightarrow \mathcal{D}_x^f e_1 = \mathcal{D}_x^f e_2) \Rightarrow \text{diff-inv } x f G \mathbb{R}_{\geq 0} 0 (e_1 = e_2) \quad \text{🌈} \\ (\text{dinv-leq}) \quad & (G \Rightarrow \mathcal{D}_x^f e_1 \leq \mathcal{D}_x^f e_2) \Rightarrow \text{diff-inv } x f G \mathbb{R}_{\geq 0} 0 (e_1 \leq e_2) \quad \text{🌈} \\ (\text{dinv-less}) \quad & (G \Rightarrow \mathcal{D}_x^f e_1 \leq \mathcal{D}_x^f e_2) \Rightarrow \text{diff-inv } x f G \mathbb{R}_{\geq 0} 0 (e_1 < e_2) \quad \text{🌈} \end{aligned}$$

The rule (**dinv-eq**) asserts that showing that an equality is an invariant reduces to showing that both sides of the equality change at the same rate over time. Similarly, the rules (**dinv-leq**) and (**dinv-less**) state that showing that inequalities are invariants requires showing that the rates of change on both sides preserve or augment the initial difference.

From the users' perspective, $\mathcal{D}_x^f e$ operates as the derivative of expression e with respect to the variables x according to the system of ODEs f . Well-known laws hold. 

$$\mathcal{D}_x^f k = 0 \quad \text{if } x \not\# k, \quad (1)$$

$$\mathcal{D}_y^f x = 0 \quad \text{if } x \not\bowtie y, \quad (2)$$

$$\mathcal{D}_X^f x = \langle f \rangle_s x \quad \text{if } x \in X \text{ and } \text{get}_{x//X} \text{ is a bounded linear operator}, \quad (3)$$

$$\mathcal{D}_x^f (e_1 + e_2) = (\mathcal{D}_x^f e_1) + (\mathcal{D}_x^f e_2), \quad (4)$$

$$\mathcal{D}_x^f (e_1 \cdot e_2) = (e_1 \cdot \mathcal{D}_x^f e_2) + (\mathcal{D}_x^f e_1 \cdot e_2), \quad (5)$$

$$\mathcal{D}_x^f e^n = n \cdot (\mathcal{D}_x^f e) \cdot e^{(n-1)}, \quad (6)$$

$$\mathcal{D}_x^f \ln(e) = (\mathcal{D}_x^f e) / e \quad \text{if } e > 0. \quad (7)$$

In summary, users can read (1) and (2) as stating that the derivative of constants or variables outside the frame of differentiation are 0. Law (3) says that the derivative of a variable inside

the frame is dictated by the ODE, and thus, users simply need to substitute according to f . The remaining laws are well-known differentiation properties such as linearity of derivatives or the Leibniz rule.

Example 10. First, consider the various Fréchet derivatives of the expression z^2 . Using the ODE $z' = 1$, the resulting expression is

$$\mathcal{D}_z^{[z \rightsquigarrow 1]} z^2 = 2 \cdot (\mathcal{D}_z^{[z \rightsquigarrow 1]} z) \cdot z = 2 \cdot 1 \cdot z = 2z.$$

However, differentiating with respect to a different variable yields

$$\mathcal{D}_y^{[z \rightsquigarrow 1]} z^2 = 2 \cdot (\mathcal{D}_y^{[z \rightsquigarrow 1]} z) \cdot z = 2 \cdot 0 \cdot z = 0,$$

assuming $y \bowtie z$. Finally, the ODE changes the final result

$$\mathcal{D}_z^{[z \rightsquigarrow 2]} z^2 = 2 \cdot (\mathcal{D}_z^{[z \rightsquigarrow 2]} z) \cdot z = 2 \cdot 2 \cdot z = 4z.$$

Therefore, certifying invariants for evolution commands reduces to computing framed Fréchet derivatives and comparing the results which are often easily computable by rewriting. For instance, to certify that $z > 0$ is an invariant of $z' = z^2$, it suffices to check

$$0 = \mathcal{D}_z^{[z \rightsquigarrow z^2]} 0 \leq \mathcal{D}_z^{[z \rightsquigarrow z^2]} z = z^2$$

by rule (*dinv-less*). We can now formally culminate the proof in Example 9. By rules (*dinv-less*) and (*dinv-leq*) respectively, the assertions $x > c$ and $y \geq x$ are invariants since

$$1 = \mathcal{D}_{x \oplus y}^{[x \rightsquigarrow 1, y \rightsquigarrow 2]} x \geq \mathcal{D}_{x \oplus y}^{[x \rightsquigarrow 1, y \rightsquigarrow 2]} c = 0 \text{ and } 2 = \mathcal{D}_{x \oplus y}^{[x \rightsquigarrow 1, y \rightsquigarrow 2]} y \geq \mathcal{D}_{x \oplus y}^{[x \rightsquigarrow 1, y \rightsquigarrow 2]} x = 1.$$

Thus, this example and Example 9 illustrate how to certify invariants through $d\mathcal{L}$'s differential induction method by applying our semantic framed (Fréchet) differentiation. \square

5.4 Ghosts and Darboux rules

Differential induction does not suffice to prove all invariant certifications [62, 68]. For instance, applying rule (*dinv-less*) to the dynamics $x' = -x$ of Examples 1-4 to show invariance of $x > 0$ does not lead to a concluding proof state. Indeed, $\mathcal{D}_x^{[x \rightsquigarrow -x]} x = -x$ is not necessarily greater or equal to $0 = \mathcal{D}_x^{[x \rightsquigarrow -x]} 0$. For those cases, differential dynamic logic $d\mathcal{L}$ includes the *differential ghost* [68] rule which asserts the correctness of an evolution command given the correctness of a higher-dimensional but equivalent system of ODEs. Here we generalise our previous formalisation of this rule [27] and use it to derive $d\mathcal{L}$'s three Darboux rules [68]. Concretely, we formalise and prove soundness of the rules:

$$(dG) \quad \frac{\{P\} \{x' = f, y' = A \cdot y + b \ \& \ G\} \{Q\}}{\{\exists v. P[v/y]\} \{x' = f \ \& \ G\} \{\exists v. Q[v/y]\}}, \quad \text{🌈}$$

$$(dbx-eq) \quad \frac{\mathcal{D}_{x+Ly}^{f(y \rightsquigarrow -c \cdot y)} e = c \cdot e}{\{e = 0\} \{x' = f \ \& \ G\} \{e = 0\}}, \quad \text{🌈}$$

$$(dbx-ge) \quad \frac{\mathcal{D}_{x+Ly}^{f(y \rightsquigarrow -c \cdot y)} e \geq c \cdot e}{\{e \bowtie 0\} \{x' = f \ \& \ G\} \{e \bowtie 0\}}, \quad \text{🌈}$$

where A is a square matrix, b is a vector, $\bowtie \in \{>, \geq\}$, x and y are independent variables $y \bowtie x$, and y does not appear in G or f : $y \sharp (G, f)$. In contrast with their usual presentation, our semantic formalisation of the Darboux rules also requires the existence of a third independent variable z such

that $z \# (G, f)$ because our proof requires two applications of the (dG) rule. More work is needed to provide an alternative proof without these conditions, and to generalise A and b in (dG) to be functions on x that do not mention y . The use of matrices in the (dG) rule was possible due to our previous work on linear systems [24, 38] but further generalisations are possible in terms of bounded linear operators.

Example 11. *Below, we provide an alternative (dbx-ge)-based Hoare-style proof that $I \Leftrightarrow (g \geq 0)$ is an invariant for $\text{blood_sugar} = \text{loop}(\text{ctrl}; \text{dyn})$ from Examples 2-4.*

$$\begin{aligned}
& \{I\} \text{loop}(\text{ctrl}; \text{dyn}) \text{inv } I \{I\} \\
& \Leftarrow \{I\} \text{ctrl}; \text{dyn} \{I\} \\
& \Leftarrow \{I\} g := g_M \{I\} \wedge \{I\} g' = -g \{I\} \\
& \Leftarrow (g \geq 0) \preceq (g_M \geq 0) \wedge \{g_M \geq 0\} g := g_M \{g \geq 0\} \wedge \mathcal{D}_{g+Ly}^{[g \rightsquigarrow -g, y \rightsquigarrow -1 \cdot y]} g \geq -1 \cdot g \\
& \Leftarrow \top \wedge \top \wedge -g \geq -g \Leftarrow \top.
\end{aligned}$$

The second implication follows by (h-seq), the third one by (h-cons) and (dbx-ge), and the last one is true by (h-assign), definitions, and the framed derivative rules. This concludes the proof. As previously noted, differential induction cannot certify that I is an invariant of $g' = -g$. Although its invariance could be verified with the flow as in Example 3, it is not always straightforward to find the solution to a system of ODEs (see Sections 7 and 8). Hence, in dL-style reasoning, one sometimes needs to embed the ODEs into a higher-dimensional space to prove invariance. Despite extant Isabelle's HOL-based proof strategies to certify this example [39], our formalisation of the Darboux rules expands our pool of methods to tackle similar problems in the style of dL. \square

6 Reasoning Components

Here, we describe our recently improved support for proof automation in our verification framework. Specifically, we discuss the proof methods we have developed using Isabelle's Eisbach tool [53] and the underlying formalisations of mathematical concepts required to make the automation effective. That is, our methods not only employ the proof rules introduced in Sections 3.5 and 5, but we also add lemmas and formalisations in this section that aid in making previously described procedures hidden from the user. Thus, our methods often discharge any side conditions generated in the verification process.

As noted in Example 2, verification problems for hybrid programs often follow the pattern $\text{loop}(\text{ctrl}; \text{dyn}) \text{inv } I$. That is, they are an iteration of a discrete controller intervening in a continuous dynamical system. We can therefore apply invariant reasoning with the laws (h-loopi) and (h-whilei), meaning that often the main task requires verifying $\{I\} \text{ctrl}; \text{dyn} \{I\}$. As foretold at the end of Section 3.5.2, two workflows can be applied to verify this Hoare triple. If the system of ODEs is solvable, then we can insert the flow using law (wlp-flow), and VCG becomes purely equational using the forward box laws (Section 3.5.1). This approach is implemented in our proof method `wlp_full` (described below in Section 6.4). Alternatively, if a solution is not available, we can find a suitable invariant I and use our differential induction proof method `dInduct`, and its variants (Section 6.6), to verify $\{I\} \text{dyn} \{I\}$. In the remainder, we describe the proof methods and formalisations that support each of these workflows. Specifically, for `wlp_simp` we describe our automation of the certification of differentiation, Lipschitz continuity, uniqueness of solutions, and the integration of computer algebra systems (CASs) into `IsaVODEs`. On the `dInduct` side, we provide proof methods for automating differential induction, weakening, and differential ghosts. Finally, we lay the foundations for automating the proofs of obligations that depend on the analysis of real-valued functions. We do this by formalising and proving well-known derivative test theorems ubiquitous in calculus.

6.1 Automatic certification of differentiation

Our tactic `vderiv` for automatically discharging statements of the form $f' = g$ for functions $f, g : \mathbb{R} \rightarrow \mathcal{C}$ has been described before [40] but it is a component of other tactics and we have extended its capabilities here. Essentially, by the chain rule of differential calculus, derivative laws often require further certifications of simpler derivatives. A recursive procedure emerges where the certification of $f' t = g t$ with $f = f_1 \circ \dots \circ f_n$ is determined by a list of proven derivative rules $(f_i(h t))' = f_i'(h t) \cdot (h' t)$ for $i \in \{1, \dots, n\}$ together with $t' = 1$. The tactic `vderiv` recursively applies these rules until it determines that

$$f' t = (f'_n t) \cdot \prod_{i=1}^{n-1} f'_i((f_{i+1} \circ \dots \circ f_n) t).$$


Then, it uses Isabelle's support for higher-order unification to try to show that

$$g t = (f'_n t) \cdot \prod_{i=1}^{n-1} f'_i((f_{i+1} \circ \dots \circ f_n) t).$$

For this work, we have added derivative rules for the real-valued exponentiation $\exp(-)$, the square root $\sqrt{-}$, the tangent $\tan(-)$ and cotangent $\cot(-)$ trigonometric functions, and vectors' inner products $*_R$ and norms $\|-\|$. The tactic `vderiv` is an integral part of those described below, and it is tacitly used in our example of Section 8.1.

6.2 Automatic certification of Lipschitz continuity

As evidenced in Examples 1-4, verification of hybrid systems might depend on knowing that there is a unique solution for a system of differential equations. In practice, certifying the existence and uniqueness of solutions with a general-purpose prover has required finding the Lipschitz-continuity constant [27, 40]. If it exists, then by the Picard-Lindelöf theorem (see Section 3.1), there is an interval around the initial time where solutions to the system of ODEs are unique. Alternatively, a domain-specific prover can restrict the specification language to a fragment where uniqueness is guaranteed [63] albeit limiting the space of verifiable dynamics. Here, we provide the foundation for allowing a general-purpose prover to automatically certify the uniqueness of solutions to IVPs. Namely, we formalise the well-known fact that continuously differentiable (C^1) functions are Lipschitz continuous. The statement of the Isabelle lemma is shown below.

```
lemma c1_local_lipschitz: 
  fixes f::"real  $\Rightarrow$  ('a::{banach,perfect_space})  $\Rightarrow$  'a"
  assumes "open S" and "open T"
    and clhyp: " $\forall \tau \in T. \forall s \in S. D (f \ \tau) \mapsto D (at \ s \ \text{within } S)$ "
    and "continuous_on S D"
  shows "local_lipschitz T S f"
  <proof>
```

Then, a procedure emerges when applying (`wlp-flow`) or (`fdia-flow`):

1. Users try to rewrite a correctness specification that includes an evolution command. That is they try to rewrite $|(x' = f \ \& \ G)_{T}^{t_0} Q \ s$ or $|(x' = f \ \& \ G)_{T}^{t_0} \rangle Q \ s$.
2. In order to guarantee that (`wlp-flow`) or (`fdia-flow`) are applicable, users need to show that there is a flow φ for the ODEs $x' = f$. This is formalised in Isabelle/HOL as the proof obligation `local_flow_on f x T S φ` .
3. Unfolding this predicate's definition yields the obligation `local_lipschitz T S f` to show that the vector field f is Lipschitz continuous on T .

4. Then, users can apply our lemma `c1_local_lipschitz` above which requires them to show that (1) T and S are open sets and that (2) the derivative of $f \tau$ is some function \mathcal{D} that (3) is continuous on S . Users can either supply \mathcal{D} or let Isabelle reconstruct it through its support for higher-order unification later in the proof.
5. Users discharge the remaining proof obligations: openness, continuity, differentiability, and original obligation with $|(x' = f \& G)_U^t| Q_s$ rewritten by (`wlp-flow`).

Example 12. We illustrate the procedure for certifying the uniqueness of solutions in Isabelle below by formalising part of the argument of Example 3. Recall the definitions of the control and dynamics of the problem.

abbreviation "ctrl \equiv IF $g \leq g_m$ THEN $g ::= g_M$ ELSE skip"

abbreviation "dyn \equiv { $g' = -g$ }"

abbreviation "flow $\tau \equiv$ [$g \rightsquigarrow g * \exp(-\tau)$]"

abbreviation "blood_sugar \equiv LOOP (ctrl; dyn) INV ($g \geq 0$)"

The procedure appears in a partial apply-style Isabelle proof below. We label each procedure step with its number and add Isabelle outputs after each `apply` command.

```

lemma "{ $g \geq 0$ } blood_sugar { $g \geq 0$ }"
  apply (wlp_simp) — the forward box  $|g' = -g| 0 \leq g$  appears
  apply (subst fbox_solve[where  $\varphi = \text{flow}$ ]) — Step (1)
  Isabelle's output: local_flow_on [ $g \rightsquigarrow g$ ]  $g \mathbb{R} \mathbb{R}$  flow — Step (2)
  apply ((clarsimp simp: local_flow_on_def)?, unfold_locales; clarsimp?)
  Isabelle's output: local_lipschitz  $\mathbb{R} \mathbb{R} (\lambda t. [g \rightsquigarrow g] \downarrow_{S_{\text{ubst}g}})$  — Step (3)
  apply (rule c1_local_lipschitz) — Step (4)
  Isabelle's outputs: open  $\mathbb{R}$ , open  $\mathbb{R}$ , continuous_on  $\mathbb{R} \mathcal{D}, \dots$ 
  (proof) — Step (5)

```

In practice, certifying openness of \mathbb{R} or intervals $(a, b) = \{x \mid a < x < b\}$ with $a \leq b$ is automatic thanks to Isabelle's simplifier. Finding derivatives or checking continuity is not always as straightforward but simple linear combinations are automatically certifiable. We have bundled the procedure for certifying the uniqueness of solutions to IVPs in an Isabelle tactic `local_flow_on_auto` described in Section 6.3 but below we focus on automating the certification of Lipschitz continuity. \square

In Example 1, we use the fact that continuously differentiable (C^1) functions are Lipschitz continuous to argue that a system of ODEs f has unique solutions by the Picard-Lindelöf theorem. Thus, formalisation of the rule `c1_local_lipschitz` is the first step towards automating the certification of the uniqueness of solutions to IVPs and a crucial step in hybrid system verification (via flows) in general-purpose proof assistants. Next, we provide tactics `c1_lipschitz` and `c1_lipschitzI` to make certification of Lipschitz continuity as seamless in VCG as in pen and paper proofs, like that of Example 1. Our tactics automate an initial application, in a backward reasoning style, of our lemma `c1_local_lipschitz`. The tactics discharge the emerging proof obligations by replicating the behaviour of `vderiv` but using rules for continuity and Fréchet differentiability available in Isabelle's HOL-Analysis library. The difference between both tactics is that `c1_lipschitzI` allows users more control by enabling them to supply the derivative \mathcal{D} . We exemplify `c1_lipschitz` in VCG below:

```

lemma "vwb_lens x  $\implies$  vwb_lens y  $\implies$  vwb_lens z
   $\implies$  x  $\boxtimes$  y  $\implies$  x  $\boxtimes$  z  $\implies$  y  $\boxtimes$  z
   $\implies$  local_lipschitz UNIV UNIV ( $\lambda t::\text{real}. [x \rightsquigarrow \$y, y \rightsquigarrow \$z] \downarrow_{S_{\text{ubst}x +_L y} s}$ )"

```

by c1_lipschitz

```
lemma "vwb_lens x  $\implies$  local_lipschitz UNIV UNIV ( $\lambda t. [x \rightsquigarrow 1 - \$x] \downarrow_{SubstX} s$ )"  
by c1_lipschitz
```

```
lemma "vwb_lens x  $\implies$  x  $\boxtimes$  y  
 $\implies$  local_lipschitz UNIV UNIV ( $\lambda t::real. [x \rightsquigarrow - (\$y * \$x)] \downarrow_{SubstX} s$ )"  
by c1_lipschitz
```

In applications, we use our tactics on framed substitutions, e.g. $[x \rightsquigarrow 1 - \$x] \downarrow_{SubstX}$, that represent systems of ODEs f . VCG also requires assumptions of lens-independence and satisfaction of lens laws due to our use of shallow expressions and frames. These are automatically provided by our `dataspace` Isabelle command and taken into account in our `c1_lipschitz` tactics. The tactic `c1_lipschitz` is indirectly used in the verification problem in Section 8.1. Both tactics automatically discharge proof obligations where the ODEs' (the vector field f) form a linear system of ODEs. This already yields polynomial and transcendental functions as solutions φ to these systems. Accordingly, users can employ our tool support for linear systems of ODEs [38]. We leave the automation of Lipschitz continuity certification for non-linear systems as future work. Alternative proof strategies like differential induction (see Section 8) are also available for cases when these tactics fail.

6.3 Automatic certification of the flow

In Section 6.2, we describe a procedure to verify the partial correctness of an evolution command by supplying the flow to its system of ODEs. Here, we automate the flow certification part of this procedure in a tactic `local_flow_on_auto`. This proof method calls our two previously described tactics `c1_lipschitz` and `vderiv`. The first one discharges the Lipschitz continuity requirement from the Picard-Lindelöf theorem to guarantee uniqueness of solutions to the associated IVP. The tactic `vderiv` certifies that the supplied flow φ is a solution to the system of ODEs. The following lines exemplify our tactic usage in hybrid systems verification tasks. Our tactic is robust as it automatically discharges frequently occurring proof obligations. Notice that for each Isabelle lemma below, there is a corresponding (local) Lipschitz continuity example from Section 6.2.

```
lemma "vwb_lens x  $\implies$  vwb_lens y  $\implies$  vwb_lens z  
 $\implies$  x  $\boxtimes$  y  $\implies$  x  $\boxtimes$  z  $\implies$  y  $\boxtimes$  z  
 $\implies$  local_flow_on [x  $\rightsquigarrow$  $y, y  $\rightsquigarrow$  $z] (x +L y) UNIV UNIV  
( $\lambda t. [x \rightsquigarrow \$z * t^2 / 2 + \$y * t + \$x, y \rightsquigarrow \$z * t + \$y]$ )"  
by local_flow_on_auto
```

```
lemma "vwb_lens x  $\implies$  local_flow_on [x  $\rightsquigarrow$  - $x + 1] x UNIV UNIV  
( $\lambda t. [x \rightsquigarrow 1 - \exp(-t) + \$x * \exp(-t)]$ )"  
by local_flow_on_auto
```

```
lemma "vwb_lens (x::real  $\implies$  's)  $\implies$  x  $\boxtimes$  y  
 $\implies$  local_flow_on [x  $\rightsquigarrow$  - $y * $x] x UNIV UNIV  
( $\lambda t. [x \rightsquigarrow \$x * \exp(-t * \$y)]$ )"  
by local_flow_on_auto
```

Crucially, `local_flow_on_auto` certifies the exponential and trigonometric solutions required for these ODEs as evidenced by its successful application on the examples above and in the verification problem of Section 8.1.

6.4 Automatic VCG with flows

In addition to our derivative, Lipschitz continuity, and flow automatic certifications, we have added various tactics for verification condition generation. The simplest one is `wlp_simp`. It merely calls

Isabelle’s simplifier adding Section 3’s *wlp* equational laws as rewriting rules except those for finite iterations and while loops which it initially tries to remove with invariant reasoning via (**h-loopi**) and (**h-whilei**). The tactic assumes that the hybrid program has the standard shape `loop (ctrl; dyn) inv I` iterating a discrete controller intervening in a continuous dynamical system as seen in Example 2. From this initial proof method, we create two tactics for supplying flows: the tactic `wlp_flow` takes as input a previously proven flow-certification theorem asserting our predicate `local_flow_on`. Alternatively, users can try to make this certification automatic with the tactic `wlp_solve` which requires as input a candidate solution φ . It calls `local_flow_on_auto` after applying `wlp_simp` and trying (**wlp-flow**) with the input φ . The intention is that both `wlp_flow` and `wlp_solve` leave only proof obligations that require reasoning of first-order logic of real numbers. Complementary tactics `wlp_full` and `wlp_expr_solve` try to discharge these remaining proof obligations automatically leaving raw Isabelle terms in the proof obligations without our syntax translations. We have also supplied simple tactics for algebraic reasoning more interactively. Specifically, we have provided a tactic for distributing factors over additions and a tactic for simplifying powers in multi-variable monomials. See Section 8.1 for the application of these in a verification problem.

6.5 Solutions from Computer Algebra Systems

We have integrated two Computer Algebra Systems (CASs), namely SageMath and the Wolfram Engine, into Isabelle/HOL to supply symbolic solutions to ODEs. The user can make use of the integration via the **find_local_flow** command, which supplies a solution to the first ODE it finds within the current subgoal.

Below, we show an application of our integration and its corresponding output in Isabelle. Users can click the greyed-out area to automatically insert the suggestion.

```
lemma local_flow_example:
  "{x > 0} x ::= 1; {x' = 1} {x > 1}"
  find_local_flow
Output:
Calling Wolfram...
λt. [x ↦ t + $x]
try this: apply (wlp_solve "λt. [x ↦ t + $x]")
```

In this case, our plugin requests a solution to the simple ODE $x' = 1$ from the Wolfram Engine. The solution is expressed as a λ -abstraction, which takes the current time as input. In the body, a substitution is constructed which gives the value for each continuous variable at time t . In this case, the solution is simply $x t = t + x 0$, which is represented by the substitution $[x \rightsquigarrow t + x]$.

Our integration follows the following steps in order to produce a solution:

1. Retrieve an Isabelle term describing the system of ODEs.
2. Convert the term into an intermediate representation.
3. Use one of the CAS backends to solve the ODE.
4. Convert the solution to an Isabelle term.
5. Certify the solution using the `wlp_solve` Isabelle tactic.

We consider each of these stages below.

Intermediate representation To have an extensible and modular interface, we implement (1) an intermediate data structure for representing ODEs and their solutions, and (2) procedures for translating back and forth between Isabelle and our intermediate representation. This allows us to

capture the structure of the ODEs without the added overhead of Isabelle syntax. It also gives us a unified interface between Isabelle and any CAS.

Our plugin assumes that a system of ODEs as a substitution $[x \rightsquigarrow e, y \rightsquigarrow f, \dots]$, as described in Section 5.2. Each expression (e, f, \dots) gives the derivative expression for each corresponding variable, potentially in terms of other variables. Naturally, for an ODE, these expressions are formed using only arithmetic operators and mathematical functions. We therefore derive the following constrained grammar for arithmetic expressions:

```
datatype AExp = NConst of string | UOp of string * AExp |
              BOp of string * AExp * AExp | NNat of int | NInt of int |
              NReal of real | CVar of string | SVar of string | IVar
```

`NConst` represents numeric constants, such as e or π . `UOp` is for unary operations and `BOp` is for binary operations. Both of these take a name, and the number of required parameter expressions. The name corresponds to the internal name for the operator in Isabelle/HOL. The operator “+”, for example, has the name `Groups.plus_class.plus`. We then have three constructors for numeric constants: `NNat` for naturals, `NInt` for integers and `NReal` for reals. We have three characterisations of variables: `CVar` are arbitrary but fixed variables, `SVar` are mutable (state) variables, and `IVar` represents the independent variable of our system, usually time.

A system of ODEs is then modelled simply as a finite map from variable names to derivative expressions. Converting back and forth between Isabelle terms (i.e. elements of the `term` datatype in Isabelle/ML) and the `AExp` datatype consists of recursing through the structure of the term, using a lookup table to translate each named arithmetic function.

Integrating a CAS into `IsaVODEs` consists of three separate components: a translation from the intermediate representation to the CAS input format, an interface with the CAS to obtain a solution, and a translation from the solution back into the required format.

Wolfram Engine The Wolfram Engine is the CAS behind WolframAlpha and Mathematica [84]. It is one of the leading CASs on the market, with powerful features for solving differential equations, amongst many other applications. The Wolfram language provides the `DSolve` function which produces solutions to various kinds of differential equations.

The basic building block for this integration is a representation of Wolfram expressions as an ML datatype. In the Wolfram language, everything is an expression [83], so this is sufficient for a complete interface. We represent these expressions as follows:

```
datatype expr = Int of int
              | Real of real
              | Id of string
              | Fun of string * expr list
              | CurryFun of string * expr list list
```

`Real` and `Int` represent real and integer numbers respectively. `Id` represented an identifiers, and `Fun` represents functions with only one list of arguments, which are distinguished from curried functions `CurryFun` with multiple sets of arguments.

Our approach for translation from `AExp` to `expr` is the following:

1. Generate an alphabetically ordered variable mapping to avoid name clashes and ease solution reconstruction.
2. Traverse the expression tree and translate each term to Wolfram.
3. Wrap in the Wolfram `DSolve` function, supplying the state and independent variables along with our ODE.

Once we have a well-formed Wolfram expression, we use the `wolframscript` command-line interface to obtain a solution in the form of a list of Wolfram Rule expressions, which are isomorphic to our substitutions. We can parse the result back into the `expr` type, and translate this into the `AExp` type using a translation table for function names and the variable name mapping we constructed.

SageMath SageMath [79] is an open-source competitor to the Wolfram Engine. It is accessed via calls to a Python API. It integrates several open-source CASs to provide its functionality, choosing in each case the best implementation for a particular symbolic computation. This makes SageMath an ideal target for integration with Isabelle.

The translation process is similar to that of our Wolfram Engine integration, but we also apply some preprocessing to the ODEs in order to make the solving more efficient. Since the CASs integrated with SageMath are better at solving smaller ODEs, we can improve their performance by rewriting the following kinds of ODEs:

1. SODEs formed from a higher order ODE, where an extra variable has been introduced in order to make all derivatives first order, can be recast into their higher-order form. For example, the ODE $x' = 2x + y, y' = x$ can be rewritten as $x'' = 2x + x'$.
2. Independent components of an ODE can be solved independently by the CAS. For example, in the ODE $x' = x, t' = 1$, x and t are independent and so they can be solved independently.

6.6 Automatic differential invariants

We have developed three main proof methods for automating differential invariant proofs: `dWeaken`, `dInduct`, and `dGhost`, which implement differential weakening, induction, and ghosts, respectively [65, 40]. The first, `dWeaken`, applies the differential weakening law to prove $\{P\} \{x' = f \mid G\} \{Q\}$, when $G \Rightarrow Q$. Proof of the implication is attempted via the auto proof method.

Application of differential induction (see §5.3) is automated by the `dInduct` proof method. To prove goals $\{I\} \{x' = f \mid G\} \{I\}$, it (1) applies rules (`dinv-eq`)-(`dinv-less`) to produce a framed derivative expression, and (2) calculates derivatives using the laws (1)-(7), substitution laws, and basic simplification laws. This yields derivative-free equality or inequality predicates. `dInduct` uses only the simplifier for calculating invariants, and so it is both efficient and yields readable VCs.

For cases requiring deduction to solve the VCs, we have implemented `dInduct_auto`, which applies `expr-auto` after `dInduct`, plus further simplification lemmas from HOL-Analysis. Ultimately such heuristics should be based on decision procedures [36, 13, 50, 20], as oracles or as verified components.

Should differential induction fail, our `diff_inv_on` tactics use the vector derivative laws from Isabelle's ODEs library [41, 40]. The latter is more interactive as users need to provide the derivatives of both sides of the (in)equality. Yet, this provides more fine-grained control and makes the tactic more likely to succeed.

While `dInduct_auto` suffices for simpler examples, differential induction must often be combined with weakening and cut rules. These rules have been explained elsewhere [65, 40]. This process is automated by a search-based proof method called `dInduct_mega`. The following steps are executed iteratively until all goals are proved or no rule applies: (1) try any fact labelled with attribute `facts`, (2) try differential weakening to prove the goal, (3) try differential cut to split it into two differential invariants, (4) try `dInduct_auto`. The rules are applied using backtracking so that if one rule fails, another one is tried.

Method `dInduct_mega` is applied by the method `dInv`, which we applied in §2. The `dInv` method allows us to prove a goal of the form $\{P\} \{x' = f \mid G\} \{Q\}$ by supplying an assertion I , and proving that this is an invariant of $\{x' = f \mid G\}$ and also that $P \Rightarrow I$. This method also requires us to apply differential cut and weakening laws.

Finally, the differential ghost law (dG) is automated through the dGhost proof method. We have complemented these tactics with sound rules from differential dynamic logic. Together with the tactics described in this Section, they enable users to seamlessly prove properties about hybrid systems in the style of dL. Yet, users of the general-purpose prover can also use the interactive style provided by Isabelle's Isar scripting language. See Section 8.2 for an example where our tactics largely automate a proof of invariance.

6.7 Derivative tests

To culminate this section, we lay the foundations for further automating the verification process in our framework. Both wlp_full and dInduct methods cannot certify complex real arithmetic proof obligations after discharging any derivative-related intermediate steps. In many cases, this requires determining the local minima and maxima of expressions and using them for further argumentation. A well-known application of differentiation is the analysis of real-valued functions: determining their local minima and maxima or where they are increasing or decreasing. Optimisation problems in physics and engineering frequently require this kind of analysis. Behind it, there are two key results generally called the *first* and *second derivative tests*. We formalise and prove these basic theorems in Isabelle/HOL, setting the foundation for automatic certification of the analysis of real-valued functions (see Section 8.3). We start by defining increasing/decreasing functions and local extrema:

definition "increasing_on T f \longleftrightarrow ($\forall x \in T. \forall y \in T. x \leq y \longrightarrow f x \leq f y$)"


definition "decreasing_on T f \longleftrightarrow ($\forall x \in T. \forall y \in T. x \leq y \longrightarrow f y \leq f x$)"

definition "strict_increasing_on T f \longleftrightarrow ($\forall x \in T. \forall y \in T. x < y \longrightarrow f x < f y$)"

definition "strict_decreasing_on T f \longleftrightarrow ($\forall x \in T. \forall y \in T. x < y \longrightarrow f y < f x$)"

definition "local_maximum_at T f x \longleftrightarrow ($\forall y \in T. f y \leq f x$)"

definition "local_minimum_at T f x \longleftrightarrow ($\forall y \in T. f y \geq f x$)"

We also prove simple consequences of these definitions on closed intervals $[a, b]$ denoted in Isabelle as $\{a..b\}$. For instance, we prove the transitivity of the increasing and decreasing properties over consecutive intervals and their relationship to local extrema. We exemplify some of these results below and refer to our repository for all proved properties. 

```
lemma increasing_on_trans:
  fixes f :: "'a::linorder  $\Rightarrow$  'b::preorder"
  assumes "a  $\leq$  b" and "b  $\leq$  c"
  and "increasing_on {a..b} f" and "increasing_on {b..c} f"
  shows "increasing_on {a..c} f"
  unfolding increasing_on_def
  by auto (smt (verit, best) intervalE nle_le order_trans)
```

```
lemma increasing_on_local_maximum:
  fixes f :: "'a::preorder  $\Rightarrow$  'b::preorder"
  assumes "a  $\leq$  b" and "increasing_on {a..b} f"
  shows "local_maximum_at {a..b} f b"
  by (auto simp: increasing_on_def local_maximum_at_def)
```

```
lemma incr_decr_local_maximum:
  fixes f :: "'a::linorder  $\Rightarrow$  'b::preorder"
  assumes "a  $\leq$  b" and "b  $\leq$  c"
```

```

    and "increasing_on {a..b} f" and "decreasing_on {b..c} f"
  shows "local_maximum_at {a..c} f b"
  unfolding increasing_on_def decreasing_on_def local_maximum_at_def
  by auto (metis intervalE linorder_le_cases)

```

Crucially, the proofs of all these results only require unfolding definitions and calling Isabelle's *auto* proof method and *Sledgehammer* tool [61] to discharge the last proof obligation. We then use our definitions of increasing and decreasing functions to state and prove the first derivative test. It states that if the derivative of a function is greater (resp. less) than 0 over an interval T , then the function is increasing (resp. decreasing) on that interval. The simple 2-line proof above uses an intermediate lemma `has_vderiv_mono_test` hiding our full proof of this basic result. 🌟

```

lemma first_derivative_test:
  assumes T_hyp: "is_interval T"
    and d_hyp: "D f = f' on T"
  shows "∀x∈T. (0::real) ≤ f' x ⇒ increasing_on T f"
    and "∀x∈T. f' x ≤ 0 ⇒ decreasing_on T f"
  unfolding increasing_on_def decreasing_on_def
  using has_vderiv_mono_test[OF assms] by blast

```

For the second derivative test, we formalise a frequently used property of continuous real-valued functions. Namely, that if a function maps a point t to some value above (resp. below) a threshold c , then there is an open set around t filled with points mapped to values above (resp. below) the threshold. This result and similar ones in terms of open balls with fixed radius around t appear in our formalisations. For these, we also complement Isabelle's library of topological concepts with the definition of *neighbourhood* and some of its properties and characterisations. 🌟

definition "neighbourhood N x \longleftrightarrow ($\exists X$. open X \wedge x \in X \wedge X \subseteq N)"

```

lemma continuous_on_Ex_open_less:
  fixes f :: "'a :: topological_space  $\Rightarrow$  real"
  assumes "continuous_on T f"
    and "neighbourhood T t"
  shows "f t > c  $\implies$   $\exists X$ . open X  $\wedge$  t  $\in$  X  $\wedge$  X  $\subseteq$  T  $\wedge$  ( $\forall \tau \in X$ . f  $\tau$  > c)"
    and "f t < c  $\implies$   $\exists X$ . open X  $\wedge$  t  $\in$  X  $\wedge$  X  $\subseteq$  T  $\wedge$  ( $\forall \tau \in X$ . f  $\tau$  < c)"

```

Finally, the second derivative test states that if the derivative of a real-valued function is 0 at a point t , and its second derivative is continuous and positive (resp. negative) at t , then the original function has a local minimum (resp. maximum) at t .

```

lemma second_derivative_test:
  assumes "continuous_on T f'"
    and "neighbourhood T t"
    and f': "D f = f' on T"
    and f'': "D f' = f'' on T"
    and "f' t = (0 :: real)"
  shows "f'' t < 0
     $\implies$   $\exists a b$ . a < t  $\wedge$  t < b  $\wedge$  {a..b}  $\subseteq$  T  $\wedge$  local_maximum_at {a..b} f t"
    and "f'' t > 0
     $\implies$   $\exists a b$ . a < t  $\wedge$  t < b  $\wedge$  {a..b}  $\subseteq$  T  $\wedge$  local_minimum_at {a..b} f t"
  unfolding local_maximum_at_def local_minimum_at_def
  using has_vderiv_max_test[OF assms] has_vderiv_min_test[OF assms]
  by blast+

```

As before, we provide the complete argument in the proof of our lemmas `has_vderiv_max_test` and `has_vderiv_min_test`. We refer interested readers to our repository for complete results. 🌟

Section 8 provides an example where we use these derivative tests to reason about real-arithmetic properties and establish the progress of a dynamical system. Beyond that, this subsection showcases the openness of our framework. Anyone can formalise well-known analysis concepts that provide background theory engineering to increase proof automation or that generalise extant verification methods.

We have presented various tactics that increase proof automation for hybrid systems verification in our framework. The automation supports both, solution and invariant-based reasoning. The definition of the tactics and their testing covers more than 500 lines of Isabelle code. From these 500 lines, the tactics setup (definitions and lemmas) comprises approximately 200 lines. This number does not take into account our described formalisations. The tactics have simplified our verification experience by discharging certifications of the uniqueness of solutions or differential inductions. We have used them extensively in a set of 66 hybrid systems verification problems. See Section 7 for more details.

7 Evaluation

For the evaluation of our verification framework, we have tackled 66 problems of the Hybrid Systems Theorem Proving (HSTP) category from the 9th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH22) [56] Friendly Competition. Of those, 5 are verifications of regular programs, 32 are verifications of continuous dynamics and 29 are verifications of hybrid programs. In a different classification, the first 9 problems test the tool’s ability to handle the interactions between hybrid programs’ constructors through various orders of loops, tests, assignments and dynamics. The next 30 problems test the tool’s ability to tackle different kinds of continuous dynamics: one evolution command after another, an evolution command with many variables at once, and dynamics that in $d\mathcal{L}$ require invariants, ghosts, differential cuts, weakenings, or Darboux rules. 21 problems come from tutorials [64, 69] (9 and 12 respectively) on how to model and prove hybrid systems in $d\mathcal{L}$. They include event and time-triggered controls for straight-line motion and two-dimensional curved motion. 3 hybrid programs come from a case study on the verification of the European train control system protocol [67]. The remaining 3 involve linear and nonlinear dynamics.



We fully proved in Isabelle 58 of the 66 problems. Of the remaining 8, we proved 4 with our verification framework while leaving some arithmetic certifications to external computer algebra systems. We could not find the proofs for 2 of the remaining 4 problems while the last 2 require the generalisation of our Darboux rule or a different proof method. This performance is similar to state-of-the-art tools participating in the same competition [56] on those benchmark problems in the proof-interaction/scripted format. In general, our first approach to solving all 66 problems was a combination of our w/p ’s tactics with a supplied solution. Yet, we only were able to solve 38 of them with this approach. The remaining 20 problems required us to employ higher-order logic methods or $d\mathcal{L}$ techniques like differential induction, ghosts, cuts, weakenings or Darboux rules. Specifically, our three proofs using Isabelle’s higher-order logic and analysis methods involved nonlinear dynamics which immediately required us to increase our interaction with the tool. A generalisation of our invariance certification methods would alleviate this because as solutions to the systems of ODEs become more complex, their certification is more difficult. This explains why invariant reasoning becomes prominent in the remaining 17 solved problems.

For 14 problems, we provided several proofs to exemplify our tool’s diversity of methods. Between providing solutions and using differential induction, neither method is comparatively easier to use than the other in the 14 problems tested. Most of the time they end up with the same number of lines of code (LOC) per proof and whenever one has more LOC for a problem, a different problem favours equally the other method. Quantitatively speaking, we used the solution method 36 times and the induction method 26 times. We used differential ghosts 4 times and our Darboux rule twice. In terms of LOC, the average length of the statement of a problem is 3.72 lines with a median and mode

of 1 totalling 246 LOCs for all problem statements. The average number of LOC of the problems' shortest proofs is 8.11 with a median of 3 and a mode of 1 totalling 511 LOC for problems' proofs. The shortest proof for a problem is 1 while the longest is 98 LOC. These figures do not take into account any additional lemmata about real numbers necessary for having fully certified proofs in Isabelle/HOL. In general, 23 of the 62 (at least partially) solved problems require the assertion of real arithmetical facts for their full verification. Thus, automation of first-order logic arithmetic for real numbers in general-purpose ITPs is highly desired for the scalability of end-to-end verification within them. Otherwise, trust in computer algebra tools will be necessary.

Our additions of tactics to the verification framework have highly reduced the number of LOC to verify a problem as evidenced by the fact that 48 benchmark problems are solved with a single call to one of our tactics. Our addition of the fact that C^1 -functions are Lipschitz continuous has largely contributed to the success of this automation: 18 of our proofs call our tactic `wlp_solve` and 9 use `local_flow_on_auto`. Finally, thanks to our shallow expressions and our nondeterministic assignments, our formalisation of the benchmark problems is now fully faithful to the ARCH22 competition-required $d\mathcal{L}$ syntax.

8 Examples

In this section, we showcase the benefits of our contributions by applying them in some of the ARCH22 competition benchmarks and examples of our own.

8.1 Rotational dynamics 3.

Our first problem illustrates the integration of all our features for ODEs. It describes the preservation of $I \Leftrightarrow d_1^2 + d_2^2 = w^2 \cdot p^2 \wedge d_1 = -w \cdot x_2 \wedge d_2 = w \cdot x_1$ for the ODEs $x_1' = d_1, x_2' = d_2, d_1' = -w \cdot d_2, d_2' = w \cdot d_1$. Observe that the relationship between d_1 and d_2 in the system of ODEs is similar to that of scaled sine and cosine functions. Moreover, the invariant states a Pythagorean relation among them. Thus, we can expect the flow to involve trigonometric functions and the problem to be solved with differential invariants in $d\mathcal{L}$ due to its previously limited capabilities to explicitly state these functions [32]. Indeed, differential induction in Isabelle/HOL can prove this example in one line:

```
lemma "(d12 + d22 = w2 * p2 ∧ d1 = - w * x2 ∧ d2 = w * x1)e ≤
  |{x1' = d1, x2' = d2, d1' = - w * d2, d2' = w * d1}|
  (d12 + d22 = w2 * p2 ∧ d1 = - w * x2 ∧ d2 = w * x1)"
by (intro fbox_invs; diff_inv_on_eq)
```

In the proof above, the application of the lemma `fbox_invs` as an introduction rule splits the Hoare-triple in three invariant statements, one for each conjunct. The semicolon indicates to Isabelle that the subsequent tactic should be applied to all emerging proof obligations. Therefore, our tactic `diff_inv_on_eq` implementing differential induction for equalities (`dinv-eq`) is applied to each of the emerging invariant statements, which concludes the proof.

Nevertheless, we can also tackle this problem directly via the flow. Despite the fact that we suspect that the solutions involve trigonometric functions, obtaining the general solution is time-consuming. Therefore, we call our integration between the Wolfram language and Isabelle/HOL `find_local_flow` to provide the solution for us. We can use the supplied solution in the proof.

```
lemma "w ≠ 0 ⇒ (d12 + d22 = w2 * p2 ∧ d1 = - w * x2 ∧ d2 = w * x1)e ≤
  |{x1' = d1, x2' = d2, d1' = - w * d2, d2' = w * d1}|
  (d12 + d22 = w2 * p2 ∧ d1 = - w * x2 ∧ d2 = w * x1)"
find_local_flow
apply (wlp_solve "λt. [d1 ~> $d1 * cos (t * w) + - 1 * $d2 * sin (t * w),
  d2 ~> $d2 * cos (t * w) + $d1 * sin (t * w),
```

```

x1 ~> $x1 + 1/w * $d2 * (- 1 + cos (t * w)) + 1/w * $d1 * sin (t * w),
x2 ~> $x2 + 1/w * $d1 * (1 + - cos (t * w)) + 1/w * $d2 * sin (t * w)]")
apply (expr_auto add: le_fun_def field_simps)
subgoal for s t
  apply mon_pow_simp
  apply (mon_simp_vars "get_x1 s" "get_x2 s" )
  using rotational_dynamics3_arith by force
done

```

The first line of the proof applies our tactic `wlp_solve` with the suggested solution from `find_local_flow`. The solution's syntax specifies one expression per variable in the system of ODEs. As explained in Section 6, the tactic `wlp_solve` applies the rule (`wlp-flow`), certifies that the corresponding vector field is Lipschitz-continuous by calling our tactic `c1_lipschitz`, and certifies that it is indeed the solution to the system of ODEs via our tactic `vderiv`. The `subgoal` command allows us to specify the name of the variables in the proof obligation so that we can use those names in our subsequent tactics. Our new tactic `mon_pow_simp` simplifies powers in the monomial expressions of the proof obligation. In the next line, our tactic `mon_simp_vars` calls `mon_pow_simp` twice but reorders factors in the order of its inputs `get_x1 s` and `get_x2 s`. The last line supplies the lemma `rotational_dynamics3_arith` (below) whose proof was provided by Sledgehammer [61].

```

lemma rotational_dynamics3_arith:
  "w2 * (get_x1 s)2 + w2 * (get_x2 s)2 = p2 * w2
  ⇒ w2 * ((cos (t * w))2 * (get_x1 s)2)
  + (w2 * ((sin (t * w))2 * (get_x1 s)2)
  + (w2 * ((cos (t * w))2 * (get_x2 s)2)
  + w2 * ((sin (t * w))2 * (get_x2 s)2)) = p2 * w2"
  <proof>

```

This example shows the versatility of using general-purpose proof assistants for hybrid systems verification. Users can provide automation methods for invariant or flow certification. This allows the integration of unverified tools into the verification process. Fast certification enables ITPs to quickly validate CAS' inputs. In terms of our contributions, this example showcases our shallow embedding's intuitive syntax, our integration of the Wolfram language for suggesting simple solutions, and our tactics automating VCG, C1-lipschitz continuity, derivatives certification, and real-arithmetic reasoning.

8.2 Dynamics: Conserved quantity

We prove that the inequality $I \Leftrightarrow x_1^4 \cdot x_2^2 + x_1^2 \cdot x_2^4 - 3 \cdot x_1^2 \cdot x_2^2 + 1 \leq c$ is an invariant of the system

$$f = \begin{cases} x_1' = 2 \cdot x_1^4 \cdot x_2 + 4 \cdot x_1^2 \cdot x_2^3 - 6 \cdot x_1^2 \cdot x_2, \\ x_2' = -4 \cdot x_1^3 \cdot x_2^2 - 2 \cdot x_1 \cdot x_2^4 + 6 \cdot x_1 \cdot x_2^2, \end{cases}$$

where we abuse notation and "equate" the vector field with its representation as a system of ODEs. In contrast with the previous benchmark, the solution to this system of ODEs is not easy to describe analytically. The following is a subexpression to the solution for x_2 according to Wolfram|Alpha

$$\left(\int_1^t \frac{1}{\sqrt{\tau^2(\tau^6 - 6\tau^4 + 9\tau^2 + c_1)} \sqrt{\frac{-\tau^4 + 3\tau^2 - \sqrt{\tau^2(\tau^6 - 6\tau^4 + 9\tau^2 + c_1)}}{\tau^2}}} d\tau \right)^{-1}$$

The solution for x_2 involves another four factors with integrals of fractions with denominators having square roots of square roots. Instead of computing these solutions and certifying them in Isabelle,

we perform differential induction. We show that I is an invariant by proving that the framed Fréchet derivatives on both sides of the inequality are 0. In Isabelle/HOL, certification of this reasoning is automatic due to our `dInduct_mega` tactic.

lemma "($x_1^4 \cdot x_2^2 + x_1^2 \cdot x_2^4 - 3 \cdot x_1^2 \cdot x_2^2 + 1 \leq c$)_e ≤
 |{ $x_1' = 2 \cdot x_1^4 \cdot x_2 + 4 \cdot x_1^2 \cdot x_2^3 - 6 \cdot x_1^2 \cdot x_2$,
 $x_2' = -4 \cdot x_1^3 \cdot x_2^2 - 2 \cdot x_1 \cdot x_2^4 + 6 \cdot x_1 \cdot x_2^2$ }|
 ($x_1^4 \cdot x_2^2 + x_1^2 \cdot x_2^4 - 3 \cdot x_1^2 \cdot x_2^2 + 1 \leq c$)"
 by dInduct_mega

Although the problem is simpler through differential induction, readers should be aware that the simplicity of proving this invariance in Isabelle benefits greatly from our increased automation. We describe below the automated steps to conclude the proof.

1. The inequality is transformed into a proof of invariance $\text{diff-inv}(x_1 \oplus x_2) f \top \mathbb{R}_+ 0 I$
2. Backward reasoning with (`dinv-leq`) requires showing $\mathcal{D}_{x_1 \oplus x_2}^f e \leq \mathcal{D}_{x_1 \oplus x_2}^f c$
3. The right hand side reduces to 0 while the simplifier performs the following rewrites on the left-hand side $e = x_1^4 \cdot x_2^2 + x_1^2 \cdot x_2^4 - 3 \cdot x_1^2 \cdot x_2^2 + 1$ (abbreviating $x_i' = \mathcal{D}_{x_1 \oplus x_2}^f x_i$)

$$\begin{aligned} \mathcal{D}_{x_1 \oplus x_2}^f e &= 4 \cdot x_1^3 \cdot x_2^2 \cdot x_1' + 2 \cdot x_1^4 \cdot x_2 \cdot x_2' + 2 \cdot x_1 \cdot x_2^4 \cdot x_1' \\ &\quad + 4 \cdot x_1^2 \cdot x_2^3 \cdot x_2' - 6 \cdot x_1 \cdot x_2^2 \cdot x_1' - 6 \cdot x_1^2 \cdot x_2 \cdot x_2' \\ &= 8 \cdot x_1^7 \cdot x_2^3 + 16 \cdot x_1^5 \cdot x_2^5 - 24 \cdot x_1^5 \cdot x_2^3 - 8 \cdot x_1^7 \cdot x_2^3 - 4 \cdot x_1^5 \cdot x_2^5 \\ &\quad + 12 \cdot x_1^5 \cdot x_2^3 + 4 \cdot x_1^5 \cdot x_2^5 + 8 \cdot x_1^3 \cdot x_2^7 - 12 \cdot x_1^3 \cdot x_2^5 - 16 \cdot x_1^5 \cdot x_2^5 \\ &\quad - 8 \cdot x_1^3 \cdot x_2^7 + 24 \cdot x_1^3 \cdot x_2^5 - 12 \cdot x_1^5 \cdot x_2^3 - 24 \cdot x_1^3 \cdot x_2^5 \\ &\quad + 36 \cdot x_1^3 \cdot x_2^3 + 24 \cdot x_1^5 \cdot x_2^3 + 12 \cdot x_1^3 \cdot x_2^5 - 36 \cdot x_1^3 \cdot x_2^3 \\ &= 0 \end{aligned}$$

4. Since $\mathcal{D}_{x_1 \oplus x_2}^f e = 0 \leq 0 = \mathcal{D}_{x_1 \oplus x_2}^f c$, the proof ends satisfactorily.

Thus, our tactic `dInduct_mega` hides various logical, algebraic, differential and numerical computations and certifications. This example showcases the scalability of hybrid systems verifications in interactive theorem provers. Namely, with adequate tactic implementations, ITPs become more automated tools and easier to use over time.

8.3 Reachability of a rocket launch

Consider a rocket's vertical liftoff and assume it loses fuel at a constant rate of $k > 0$ kilograms per second starting with $m_0 > k$ kilograms while its acceleration is equal to the amount of fuel left in it. These assumptions do not accurately model rockets' liftoff; however, they suffice to produce a behaviour approximating the observed phenomena, see Figure 2, and facilitate the presentation of our contributions. The corresponding system of ODEs and its solution are

$$f = \begin{cases} m' = -k, \\ v' = m, \\ y' = v, \\ t' = 1, \end{cases} \quad \varphi \tau = \begin{cases} m \tau = -k \cdot \tau + m_0, \\ v \tau = -k \cdot \frac{\tau^2}{2} + m_0 \cdot \tau = \tau \cdot (-k \cdot \frac{\tau}{2} + m_0), \\ y \tau = -k \cdot \frac{\tau^3}{6} + m_0 \cdot \frac{\tau^2}{2} = \frac{\tau^2}{2} \cdot (-k \cdot \frac{\tau}{3} + m_0), \\ t \tau = \tau, \end{cases}$$

where m is the fuel's mass, v is the rocket's velocity, y is its altitude, and t models time.

We study the rocket's behaviour before it reaches a maximum altitude with its first-stage propulsion because the second stage should begin before the rocket starts falling. We therefore prove two

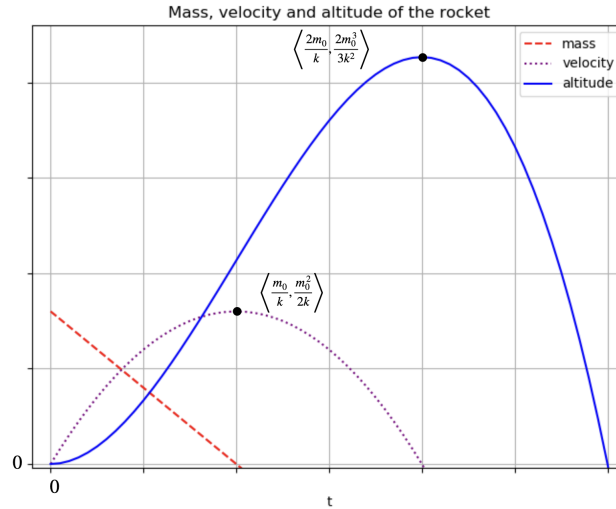


Figure 2: Depiction of the rocket behaviour assuming $m_0 > k$

things about the initial propulsion stage. The first is that no matter which height h we consider strictly below the maximum altitude $H = 2m_0^3/(3k^2)$, there will always be a state of the rocket greater than h approaching H . The second is that all scenarios with a single propulsion from the ground lead to altitudes lower than H . Using the abbreviations “odes” for f and “flow” for φ , the verification of the first specification is now possible thanks to our implementation of forward diamonds.

lemma local_flow_on_rocket:

```
"local_flow_on [y↦$v, v↦$m, t↦1, m↦-k] (y+Lv+Lt+Lm) UNIV UNIV flow"
by local_flow_on_auto
```

```
lemma "(0 ≤ h ∧ h < H ∧ m = m_0 ∧ m_0 > k ∧ t = 0 ∧ v = 0 ∧ y = 0)e ≤ |odes> (h ≤ y)"
using k_ge_1
by (subst fdia_g_ode_frame_flow[OF local_flow_on_rocket]; expr_simp)
(auto simp: field_simps power3_eq_cube intro!: exI[of _ "2*m_0/k"])
```

The proof starts with the law (**fdia-flow**) using the lemma `local_flow_on_rocket` asserting that φ is the flow for f . The proof of this is now automatic due to our tactic `local_flow_on_auto`. The second line of the proof for our specification culminates with some arithmetical reasoning, where we provide the time $2m_0/k$ as the witness for the existential quantifier in the law (**fdia-flow**). That is, the time for the second 0-intercept of the velocity when the maximum altitude is reached.

The second specification looks equally simple and its three line proof is deceiving (see below). Our contributed tactics automatically handle VCG, derivative certifications, and uniqueness. However, as reported in Section 7, emerging arithmetic proof obligations often have to be checked separately. We do this with the lemma `rocket_arith` proved using our derivative tests from Section 6.7.

```
lemma "(m = m_0 ∧ m_0 > k ∧ t = 0 ∧ v = 0 ∧ y = 0)e
≤ |ode> (y ≤ 2*m_0^3/(3*k^2))"
apply (wlp_solve "flow")
using k_ge_1 rocket_arith
by (expr_simp add: le_fun_def)
```

lemma rocket_arith:

```
assumes "(k::real) > 1" and "m_0 > k" and "x ∈ {0..}"
shows "- k*x^3/6 + m_0*x^2/2 ≤ 2*m_0^3/(3*k^2)" (is "?lhs ≤ _")
proof-
let ?f = "λt. -k*t^3/6 + m_0*t^2/2"
```

```

and ?f' = "\lambda t. -k*t^2/2 + m_0*t"
and ?f'' = "\lambda t. -k*t + m_0"
have "2*m_0^3/(3*k^2) = -k*(2*m_0/k)^3/6 + m_0*(2*m_0/k)^2/2" (is "_ = ?rhs")
  by (auto simp: field_simps power3_eq_cube)
moreover have "?lhs ≤ ?rhs"
proof(cases "x ≤ 2 * m_0 / k")
  case True
  have ge_0_left: "0 ≤ y ⇒ y ≤ m_0/k ⇒ ?f' 0 ≤ ?f' y" for y
    apply (rule has_vderiv_mono_test(1)[of "{0..m_0/k}" ?f' ?f'' 0])
    using ⟨k > 1⟩ ⟨m_0 > k⟩
    by (auto intro!: vderiv_intros simp: field_simps)
  moreover have ge_0_right: "m_0/k ≤ y ⇒ y ≤ 2*m_0/k
    ⇒ ?f' (2*m_0/k) ≤ ?f' y" for y
    apply (rule has_vderiv_mono_test(2)
      [of "{m_0/k..2*m_0/k}" ?f' ?f'' _ "2*m_0/k"])
    using ⟨k > 1⟩ ⟨m_0 > k⟩
    by (auto intro!: vderiv_intros simp: field_simps)
  ultimately have ge_0: "∀y∈{0..2*m_0/k}. 0 ≤ ?f' y"
    using ⟨k > 1⟩ ⟨m_0 > k⟩
    by (fastforce simp: field_simps)
  show ?thesis
    apply (rule has_vderiv_mono_test(1)[of "{0..2*m_0/k}" ?f' ?f'' _ "2*m_0/k"])
    using ge_0 True ⟨x ∈ {0..}⟩ ⟨k > 1⟩ ⟨m_0 > k⟩
    by (auto intro!: vderiv_intros simp: field_simps)
next
  case False
  have "2*m_0/k ≤ y ⇒ ?f' y ≤ ?f' (2*m_0/k)" for y
    apply (rule has_vderiv_mono_test(2)[of "{m_0/k..}" ?f' ?f''])
    using ⟨k > 1⟩ ⟨m_0 > k⟩ by (auto intro!: vderiv_intros simp: field_simps)
  hence obs: "∀y∈{2*m_0/k..}. ?f' y ≤ 0"
    using ⟨k > 1⟩ ⟨m_0 > k⟩
    by (clarsimp simp: field_simps)
  show ?thesis
    apply (rule has_vderiv_mono_test(2)[of "{2*m_0/k..}" ?f' ?f''])
    using False ⟨k > 1⟩ obs
    by (auto intro!: vderiv_intros simp: field_simps)
qed
ultimately show ?thesis
  by simp
qed

```

Our proof of `rocket_arith` analyses the altitude behaviour to the left ($x \leq 2m_0/k$) and right ($x > 2m_0/k$) of the maximum altitude. Using the first derivative test, we show that, to the left, the altitude is increasing, and to the right, it is decreasing. Thus, the rocket achieves a maximum altitude at $\langle 2m_0/k, 2m_0^3/(3k^2) \rangle$. Accordingly, we repeat our use of the derivative test to show that the velocity remains above 0 in the interval $[0, 2m_0/k]$.

This example illustrates the relevance of our additions of forward diamonds and derivative tests. The diamonds allow us to do reachability proofs and show progress for hybrid systems. The tests enable us to prove emerging real-arithmetic proof obligations after VCG and provide the basis for increasing automation in this subtask of the verification process.

We have showcased various examples illustrating the automation added to our hybrid system verification framework. We refer readers interested in more examples using, for instance, our integration of vectors and matrices to previous publications and our participation in ARCH competition reports [27, 54, 56, 38].

9 Related Work

We know of two complementary approaches to the verification of hybrid systems: reachability analysis and deductive verification. Reachability analysis [3, 6, 15, 17, 19, 29, 30, 51] approximates the set of all reachable states of a hybrid system via the iteration of its transition relation until reaching a fixed point or a specification-violating state. This approach iteratively explores a hybrid system's state space and finds states that violate specified properties.

Our focus is on the deductive verification of hybrid systems with interactive theorem provers (ITPs) [1, 7, 27, 31, 63, 71, 75, 80]. It uses mathematical proofs to establish adherence to safety specifications for all system states. Examples of this approach in relevant general-purpose ITPs include a formalisation of hybrid automata and invariant reasoning for them in the PVS prover [1], a shallowly embedded implementation of a Logic of Events for hybrid systems reasoning in the Coq prover [7], or the use of the Coquelicot library for formalising a temporal logic of actions with the same purpose [71]. All these approaches have different semantics from our predicate transformer one and arguably employ less automated ITPs than our choice. In Isabelle/HOL, Hoare-style verification and refinement frameworks have appeared [9, 48, 73] and fewer have been specialised to hybrid systems [25]. These frameworks could be combined with our own in the spirit of working towards a full hybrid systems development environment within Isabelle/HOL.

The formalisms to describe and prove hybrid systems' correctness specifications are as diverse as the tools to implement such deductive verification systems. The HHL and HHLPy provers [75, 80] employ a Hybrid Hoare Logic (HHL) [86] for reasoning about Hybrid Communicating Sequential Processes (HCSPs) with the duration calculus in Isabelle/HOL and Python respectively. Analogously, the KeYmaera and KeYmaera X provers implement versions of differential dynamic logic $d\mathcal{L}$, a logic to reason about hybrid systems [31, 63]. Our work has been compared with these provers in hybrid system verification competitions [54, 56]. Yet, the tools are very different in nature and implementation. While both families of provers implement specific logics, our development is flexible and includes a differential Hoare logic, a refinement calculus [25], rules from differential dynamic logic [27], and linear systems (matrix) integrations [38]. Moreover, both the HHL and KeYmaera families have integrated unverified tools into their certification process. The HHLPy prover is mainly written in Python, while KeYmaera X uses the Wolfram Engine and/or Z3 as black-box solvers for quantifying elimination procedures. Albeit, the first HHL prover is verified with Isabelle/HOL, and there is work towards verifying real arithmetic algorithms for integration into KeYmaera X [72]. In contrast, our work has taken a stricter approach where every input from external tools must be certified by Isabelle/HOL. This illustrates our long-term goal of enabling general-purpose ITPs with fully automated verification capabilities.

We built our IsaVODEs framework on top of the Archive of Formal Proofs (AFP) entry for Ordinary Differential Equations [43] and our own extensions to it [38, 40]. Together with Isabelle's HOL-Analysis library, they provide a thorough basis for stating real analysis theorems in Isabelle/HOL. In particular, the library already has a different formalisation of the fact that C^1 -differentiation implies Lipschitz continuity. Nevertheless, that version depends on a type of bounded linear continuous functions while our implementation avoids creating a new type and the corresponding abstraction functions. As a result, our version is more manageable within IsaVODEs.

Formalisations of $d\mathcal{L}$ and related logics have recently appeared in the AFP [16, 66] but they are not intended as verification tools and, therefore, they are incomparable with our framework. Despite the implementation differences, we compare KeYmaera X and our support for $d\mathcal{L}$ reasoning. Our personal experience indicates that using (one-step) differential cuts, weakenings, and inductions is similar to their use in KeYmaera X. However, supplying solutions works differently because our framework requires certifying or assuming their uniqueness. In contrast, the uniqueness of solutions is guaranteed by $d\mathcal{L}$'s syntax. Our work in this paper automates this certification process. Finally, our verification of the soundness of the differential ghost rule presented here could be generalised further to match all the cases prescribed by $d\mathcal{L}$'s syntactic implementation.

10 Conclusions and Future Work

In this paper, we have described IsaVODEs, our framework for verifying cyber-physical systems in Isabelle/HOL. This substantial development includes both strong theoretical foundations and practical verification support, provided by automated theorem provers and an integration with computer algebra systems. Our language and verification technique extends $d\mathcal{L}$'s hybrid programs in several ways, notably with matrices to support engineering mathematics, and frames to support modular reasoning. We have validated our tool with a substantial library of benchmarks and examples.

Here we have improved our framework by formalising and proving VCG laws about forward diamonds which enable reasoning about the reachability or progress of hybrid systems. We have generalised our frame laws and $d\mathcal{L}$ -style differential ghosts rule, allowing us to derive related Darboux rules. We have formalised foundational theorems like the fact that differentiable functions are Lipschitz-continuous, and the first and second derivative test laws. These support the practical goal of increasing automation via our proof methods for performing differential induction or VCG via supplying flows. Our integration of CASs into this process makes the verification experience with flows seamless. Finally, we have evaluated the benefits of all these additions with various verification examples.

Our Isabelle-based approach is inherently extensible. We can add syntax and semantics for bespoke program operators and associated Hoare-logic rules to support tailoring for particular models. Overall, verifying CPSs using IsaVODEs benefits from the wealth of technology provided by Isabelle, notably the frontend, asynchronous document processing, the theory library, proof automation, and support for code generation. We need not be limited to a single notation but can provide semantics for established engineering notations. IsaVODEs benefits from the fact that Isabelle is a gateway for a variety of other verification tools through “hammers”, such as SMT solvers, model generators, and computer algebra systems. Our additions in this paper increase IsaVODEs' usability for complex verifications. We believe these advantages can allow the integration of our technology into software engineering workflows.

A limitation of our current approach occurs when the arithmetic obligations at the end of the verification are too complex for SMT solvers [61]. Currently, there are two options: users can manually prove these obligations themselves, or they can assert them at the cost of increasing uncertainty in their verification. In these cases, the ideal approach would connect tools deciding these expressions, e.g. CAS systems or domain-specific automated provers [4] in a way that IsaVODEs certifies the underlying reasoning. We leave this development for future work.

Another avenue of improvement, following our introduction of the forward diamond in our framework, is the addition of the remaining modal operators and their VCG rules [77]. Currently, we have only formalised a backward diamond but its VCG rules remain to be proved. Their implementation could lead to a framework for incorrectness analysis [58] of hybrid systems complementing current testing and simulation techniques. We will also consider supporting further extensions to hybrid programs, such as quantified $d\mathcal{L}$ and differential-algebraic logic (DAL), which can both provide additional modelling capacity.

In terms of alternative uses of our framework, we expect $d\mathcal{L}$ -style security analysis about hybrid systems [85] to be easily done in IsaVODEs too. Similarly, IsaVODEs foundations have been used as semantics for other model-based robot development technologies [12, 18]. Therefore, IsaVODEs proofs could be integrated into these tools for increased confidence in the performed analysis.

Acknowledgements This preprint has not undergone peer review (when applicable) or any post-submission improvements or corrections.

Funding statements A Novo Nordisk Fonden Start Package Grant (NNF20OC0063462) and a Horizon MSCA 2022 Postdoctoral Fellowship (project acronym DeepIsaHOL and number 101102608) partially supported the first author during the development of this article. Views and opinions ex-

pressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Executive Agency. Neither the European Union nor the European Research Executive Agency can be held responsible for them. The work was also funded by UKRI-EP SRC project CyPhyAssure (grant reference EP/S001190/1), the Assuring Autonomy International Programme (AAIP; grant CSI:Cobot), a partnership between Lloyd's Register Foundation and the University of York, and Labex DigiCosme through an invited professorship of the fourth author at the Laboratoire d'informatique de l'École polytechnique.

References

- [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, New Jersey, 2001. IEEE Computer Society.
- [2] E. Ackerman, L. C. Gatewood, J. W. Rosevear, and G. D. Molnar. Model studies of blood-glucose regulation. *The bulletin of mathematical biophysics*, 27(5):21–37, 1965. <https://doi.org/10.1007/BF02477259>.
- [3] A. S. Adimoolam and T. Dang. Safety verification of networked control systems by complex zonotopes. *Leibniz Trans. Embed. Syst.*, 8(2):01:1–01:22, 2022.
- [4] B. Akbarpour and L. C. Paulson. MetiTarski: An automatic theorem prover for real-valued special functions. *JAR*, 44(3):175–205, 2010.
- [5] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub. Jasmin: High-assurance and high-speed cryptography. In *CCS 2017*, pages 1807–1823, New York, 2017. ACM.
- [6] M. Althoff. An introduction to CORA 2015. In *ARCH 2015*, volume 34, pages 120–151, EasyChair, 2015. EasyChair.
- [7] A. Anand and R. A. Knepper. Roscoq: Robots powered by constructive reals. In *ITP*, volume 9236 of *LNCS*, pages 34–50, Heidelberg, 2015. Springer.
- [8] J. Aransay and J. Divasón. A formalisation in HOL of the fundamental theorem of linear algebra and its application to the solution of the least squares problem. *JAR*, 58(4):509–535, 2017.
- [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] R. Back and J. von Wright. *Refinement Calculus—A Systematic Introduction*. Springer, Heidelberg, 1998.
- [11] D. A. Basin, T. Dardinier, N. Hauser, L. Heimes, J. J. H. y Munive, N. Kaletsch, S. Krstic, E. Marsicano, M. Raszyk, J. Schneider, D. L. Tiore, D. Traytel, and S. Zingg. VeriMon: A formally verified monitoring tool. In H. Seidl, Z. Liu, and C. S. Pasareanu, editors, *ICTAC 2022*, volume 13572 of *LNCS*, pages 1–6, Heidelberg, 2022. Springer.
- [12] J. Baxter, G. Carvalho, A. Cavalcanti, and F. R. Júnior. Roboworld: Verification of robotic systems with environment in the loop. *FAC*, 2023. Just Accepted.
- [13] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1), 2016.

- [14] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffei, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *POPL 2014*, pages 87–100, New York, 2014. ACM.
- [15] S. Bogomolov, M. Forets, G. Frehse, K. Potomkin, and C. Schilling. JuliaReach: a toolbox for set-based reachability. In N. Ozay and P. Prabhakar, editors, *HSCC 2019*, pages 39–44, New York, 2019. ACM.
- [16] R. Bohrer, V. Rahli, I. Vukotic, M. Völpl, and A. Platzer. Formally verified differential dynamic logic. In *CPP*, pages 208–221, New York, 2017. ACM.
- [17] 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, New Jersey, 2010. IEEE Computer Society.
- [18] A. Cavalcanti, Z. Attala, J. Baxter, A. Miyazawa, and P. Ribeiro. *Model-Based Engineering for Robotics with RoboChart and RoboTool*, pages 106–151. Springer, Heidelberg, 2023.
- [19] 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, Heidelberg, 2013. Springer.
- [20] K. Cordwell, K. T. Yong, and P. A. A verified decision procedure for univariate real arithmetic with the BKR algorithm. In L. Cohen and C. Kaliszyk, editors, *ITP*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:20, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [21] J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.
- [22] S. Foster and J. Baxter. Automated algebraic reasoning for collections and local variables with lenses. In *RAMiCS*, volume 12062 of *LNCS*, Heidelberg, 2020. Springer.
- [23] S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying semantic foundations for automated verification tools in Isabelle/UTP. *Science of Computer Programming*, 197, October 2020.
- [24] S. Foster, M. Gleirscher, and R. Calinescu. Towards deductive verification of control algorithms for autonomous marine vehicles. In *ICECCS*, New Jersey, October 2020. IEEE.
- [25] S. Foster, J. J. Huerta y Munive, and G. Struth. Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In *RAMiCS[postponed]*, volume 12062 of *LNCS*, pages 169–186, 2020.
- [26] S. Foster, Y. Nemouchi, M. Gleirscher, R. Wei, and T. Kelly. Integration of formal proof into unified assurance cases with Isabelle/SACM. *Formal Aspects of Computing*, 2021.
- [27] S. Foster, J. J. H. y Munive, M. Gleirscher, and G. Struth. Hybrid systems verification with Isabelle/HOL: Simpler syntax, better models, faster proofs. In *FM 2021*, volume 13047 of *LNCS*, pages 367–386, Heidelberg, 2021. Springer.
- [28] S. Foster and F. Zeyda. Optics. *Archive of Formal Proofs*, May 2017.
- [29] 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, Heidelberg, 2011. Springer.
- [30] G. Frehse, B. H. Krogh, R. A. Rutenbar, and O. Maler. Time domain verification of oscillator circuit properties. In O. Maler, editor, *FAC 2005*, volume 153 of *ENTCS*, pages 9–22, Amsterdam, 2005. Elsevier.

- [31] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völpl, and A. Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In A. P. Felty and A. Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 527–538, Heidelberg, 2015. Springer.
- [32] J. Gallicchio, Y. K. Tan, S. Mitsch, and A. Platzer. Implicit definitions with differential equations for keymaera X - (system description). In *IJCAR 2022*, pages 723–733, 2022.
- [33] V. B. F. Gomes and G. Struth. Modal Kleene algebra applied to program correctness. In *FM*, volume 9995 of *LNCS*, pages 310–325, 2016.
- [34] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *10th Intl. Symp. on Functional and Logic Programming (FLOPS)*, volume 6009 of *LNCS*, pages 103–117, Heidelberg, 2010. Springer.
- [35] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Massachusetts, 2000.
- [36] J. Hölzl. Proving inequalities over reals with computation in Isabelle/HOL. In *PLMMS*, pages 38–45, New York, 2009. ACM.
- [37] J. Hölzl, F. Immler, and B. Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In *ITP*, volume 7998 of *LNCS*, pages 279–294, Heidelberg, 2013. Springer.
- [38] J. J. Huerta y Munive. Affine systems of ODEs in Isabelle/HOL for hybrid-program verification. In *SEFM*, volume 12310 of *LNCS*, pages 77–92, Heidelberg, 2020. Springer.
- [39] J. J. Huerta y Munive. *Algebraic verification of hybrid systems in Isabelle/HOL*. PhD thesis, The University of Sheffield, 2021.
- [40] J. J. Huerta y Munive and G. Struth. Predicate transformer semantics for hybrid systems. *JAR*, 66(1):93–139, 2022.
- [41] F. Immler and J. Hölzl. Ordinary differential equations. *Archive of Formal Proofs*, 2012.
- [42] F. Immler and C. Traut. The flow of ODEs: Formalization of variational equation and poincaré. In *ITP 2016*, volume 9807 of *LNCS*, pages 184–199, Heidelberg, 2016. Springer.
- [43] F. Immler and C. Traut. The flow of ODEs: Formalization of variational equation and Poincaré map. *JAR*, 62(2):215–236, 2019.
- [44] 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.
- [45] G. Klein et al. seL4: Formal verification of an OS kernel. In *Proc. 22nd Symp. on Operating Systems Principles (SOSP)*, pages 207–220, New York, 2009. ACM.
- [46] O. Kuncar and A. Popescu. A consistent foundation for Isabelle/HOL. *JAR*, 62:531–555, 2019.
- [47] P. Lammich. Generating verified LLVM from Isabelle/HOL. In *ITP 2019*, volume 141 of *LIPICs*, pages 22:1–22:19, Germany, 2019. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
- [48] P. Lammich. Refinement to imperative HOL. *JAR*, 62(4):481–503, 2019.
- [49] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. CompCert – a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems*, France, 2016. SEE.
- [50] W. Li, G. Passmore, and L. Paulson. Deciding univariate polynomial problems using untrusted certificates in Isabelle/HOL. *J. Autom. Reasoning*, 62:29–91, 2019.

- [51] Y. Li, H. Zhu, K. Braught, K. Shen, and S. Mitra. Verse: A python library for reasoning about multi-agent hybrid system scenarios. In *CAV 2023*, volume 13964 of *LNCS*, pages 351–364, Heidelberg, 2023. Springer.
- [52] 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, Heidelberg, 2011. Springer.
- [53] D. Matichuk, T. C. Murray, and M. Wenzel. Eisbach: A proof method language for Isabelle. *J. Automated Reasoning*, 56(3):261–282, 2016.
- [54] 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.*, volume 74, pages 141–161, EasyChair, 2020. EasyChair.
- [55] S. Mitsch and A. Platzer. Verified runtime validation for partially observable hybrid systems. *CoRR*, abs/1811.06502, 2018.
- [56] S. Mitsch, B. Zhan, H. Sheng, A. Bentkamp, X. Jin, S. Wang, S. Foster, C. P. Laursen, and J. J. H. y Munive. ARCH-COMP22 category report: Hybrid systems theorem proving. In *ARCH22*, volume 90 of *EPiC Series in Computing*, pages 185–203, Munich, 2022. EasyChair.
- [57] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software and Systems Modelling*, 18:3097–3149, January 2019.
- [58] P. W. O’Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, 2020.
- [59] F. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, 1982.
- [60] L. C. Paulson. *ML for the working programmer (2. ed.)*. Cambridge University Press, Cambridge, 1996. <https://www.cl.cam.ac.uk/~lp15/MLbook/>.
- [61] L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In G. Sutcliffe, S. Schulz, and E. Ternovska, editors, *IWIL 2010*, volume 2 of *EPiC Series in Computing*, pages 1–11, EasyChair, 2010. EasyChair.
- [62] A. Platzer. The structure of differential invariants and differential cut elimination. *Logical Methods in Computer Science*, 8(4), 2008.
- [63] A. Platzer. *Logical Analysis of Hybrid Systems*. Springer, Heidelberg, 2010.
- [64] A. Platzer. Logics of dynamical systems. In *LICS*, pages 13–24, 2012. <https://doi.org/10.1109/LICS.2012.13>.
- [65] A. Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, Heidelberg, 2018.
- [66] A. Platzer. Differential game logic. *Archive of Formal Proofs*, 2019, 2019.
- [67] A. Platzer and J. Quesel. European train control system: A case study in formal verification. In *ICFEM*, volume 5885 of *LNCS*, pages 246–265, Heidelberg, 2009. Springer.
- [68] A. Platzer and Y. K. Tan. Differential equation axiomatization: The impressive power of differential ghosts. In *LICS*, pages 819–828, New Jersey, 2018. ACM.
- [69] J. Quesel, S. Mitsch, S. M. Loos, N. Arechiga, and A. Platzer. How to model and prove hybrid systems with KeYmaera: a tutorial on safety. *STTT*, 18(1):67–91, 2016.

- [70] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, New Jersey, 2002. IEEE.
- [71] D. Ricketts, G. Malecha, M. M. Alvarez, V. Gowda, and S. Lerner. Towards verification of hybrid systems in a foundational proof assistant. In *MEMOCODE*, pages 248–257, New Jersey, 2015. IEEE.
- [72] M. Scharager, K. Cordwell, S. Mitsch, and A. Platzer. Verified quadratic virtual substitution for real arithmetic. In M. Huisman, C. S. Pasareanu, and N. Zhan, editors, *FM 2021*, volume 13047 of *LNCS*, pages 200–217, Heidelberg, 2021. Springer.
- [73] N. Schirmer. *Verification of sequential imperative programs in Isabelle-HOL*. PhD thesis, Technical University Munich, Germany, 2006.
- [74] N. Schirmer and M. Wenzel. State spaces – the locale way. In *SSV 2009*, volume 254 of *ENTCS*, pages 161–179, 2009.
- [75] H. Sheng, A. Bentkamp, and B. Zhan. HHLPy: Practical verification of hybrid systems using hoare logic. In M. Chechik, J. Katoen, and M. Leucker, editors, *FM 2023*, volume 14000 of *LNCS*, pages 160–178, Heidelberg, 2023. Springer.
- [76] D. Stanescu and B. M. Chen-Charpentier. Random coefficient differential equation models for bacterial growth. *Mathematical and Computer Modelling*, 50(5):885–895, 2009. <https://doi.org/10.1016/j.mcm.2009.05.017>.
- [77] G. Struth. Transformer semantics. *Archive of Formal Proofs*, 2018.
- [78] G. Teschl. *Ordinary Differential Equations and Dynamical Systems*. AMS, Rhode Island, 2012.
- [79] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.0)*, 2020.
- [80] S. Wang, N. Zhan, and L. Zou. An improved HHL prover: An interactive theorem prover for hybrid systems. In *ICFEM*, volume 9407 of *LNCS*, pages 382–399, 2015.
- [81] F. Weinert. *Radioactive Decay Law (Rutherford–Soddy)*. Springer, Berlin, Heidelberg, 2009. https://doi.org/10.1007/978-3-540-70626-7_183.
- [82] M. Wenzel, C. Ballarin, S. Berghofer, J. Blanchette, T. Bourke, L. Bulwahn, A. Chaieb, L. Dixon, F. Haftmann, B. Huffman, L. Hupel, G. Klein, A. Krauss, O. Kuncar, A. Lochbihler, T. Nipkow, L. Noschinski, D. von Oheimb, L. Paulson, S. Skalberg, C. Sternagel, and D. Traytel. The Isabelle/Isar reference manual, 2023. <https://isabelle.in.tum.de/doc/isar-ref.pdf>.
- [83] Wolfram Research. Expressions - wolfram language.
- [84] Wolfram Research. Wolfram engine.
- [85] J. Xiang, N. Fulton, and S. Chong. Relational analysis of sensor attacks on cyber-physical systems. In *CSF 2021*, pages 1–16, New Jersey, 2021. IEEE.
- [86] N. Zhan, B. Zhan, S. Wang, D. P. Guelev, and X. Jin. A generalized hybrid Hoare logic. *CoRR*, abs/2303.15020, 2023.