This is a repository copy of *Predicate transformer semantics for hybrid systems: verification components for Isabelle/HOL*.

Predicate Transformer Semantics for Hybrid Systems

Verification Components for Isabelle/HOL

Jonathan Julián Huerta y Munive[1] · Georg Struth[1]

Abstract

We present a semantic framework for the deductive verification of hybrid systems with Isabelle/HOL. It supports reasoning about the temporal evolutions of hybrid programs in the style of differential dynamic logic modelled by flows or invariant sets for vector fields. We introduce the semantic foundations of this framework and summarise their Isabelle formalisation as well as the resulting verification components. A series of simple examples shows our approach at work.

Keywords Hybrid systems · Predicate transformers · Modal Kleene algebra · Hybrid program verification · Interactive theorem proving

1 Introduction

Hybrid systems combine continuous dynamics with discrete control. Their verification is receiving increasing attention as the number of computing systems controlling real-world physical systems is growing. Mathematically, hybrid system verification requires integrating continuous system dynamics, often modelled by systems of differential equations, and discrete control components into hybrid automata, hybrid programs or similar domain-specific modelling formalisms, and into analysis techniques for these. Such techniques include state space exploration, reachability or safety analyses by model checking and deductive verification with domain-specific logics [10].

One of the most prominent deductive approaches is differential dynamic logic d$\mathcal{L}$ [47], an extension of dynamic logic [21] to hybrid programs for reasoning with autonomous systems of differential equations, their solutions and invariant sets. It is supported by the KeYmaera X tool [14] and has proved its worth in numerous case studies [33,40,47]. KeYmaera X verifies partial correctness specifications for hybrid programs using a combination of domain-specific sequent and Hilbert calculi, which itself is based on an intricate uniform substitution calculus.

✉ Jonathan Julián Huerta y Munive
jonathan.munive@di.ku.dk

Georg Struth
g.struth@sheffield.ac.uk

[1] Department of Computer Science, University of Sheffield, Sheffield, UK

For pragmatic reasons, its language is restricted to differential terms of real arithmetic [14] (that of hybrid automata is usually restricted to polynomial or linear constraints [10]).

Our initial motivation for this work has been the formalisation of a d$\mathcal{L}$-style approach to hybrid program verification in the Isabelle/HOL proof assistant [28] by combining Isabelle's mathematical components for analysis and ordinary differential equations [23,29–31] with verification components for modal Kleene algebras [17]. We are using a shallow embedding that, in general, encodes semantic representations of domain-specific formalisms within a host-language (deep embeddings start from syntactic representations using data types to program abstract syntax trees). This benefits not only from the well-known advantages of shallowness: more rapid developments and simpler, more adaptable components. It has also shifted our focus from encoding d$\mathcal{L}$'s complex syntactic proof system to developing denotational semantics for hybrid systems and supporting the natural style in which mathematicians, physicists or engineers reason about them—without proof-theoretic baggage. After all, we get Isabelle's own proof system and proof methods for free, and our expressive power is only limited by its type theory and higher-order logic.

Our main contribution is an open compositional semantic framework for the deductive verification of hybrid programs in a general purpose proof assistant. In a nutshell, hybrid programs are while programs, or simply programs with control loops, in which an evolution command for the continuous system dynamics complements the standard assignment command for the discrete control. Evolution commands roughly specify vector fields (via systems of ordinary differential equations) together with guards that model boundary conditions. Here, we restrict our attention to abstract predicate transformer algebras using modal Kleene algebras [9], quantales of lattice endofunctions or quantaloids of functions between lattices [5]. They are instantiated first to intermediate relational or state transformer semantics for d$\mathcal{L}$-style hybrid programs, and then to concrete semantics over program stores for hybrid programs: for dynamical systems with global flows, Lipschitz continuous vector fields with local flows and continuous vector fields with multiple solutions. Another verification component is based directly on flows. This array of components demonstrates the compositionality and versatility of our framework. Figure 1 shows its basic anatomy.
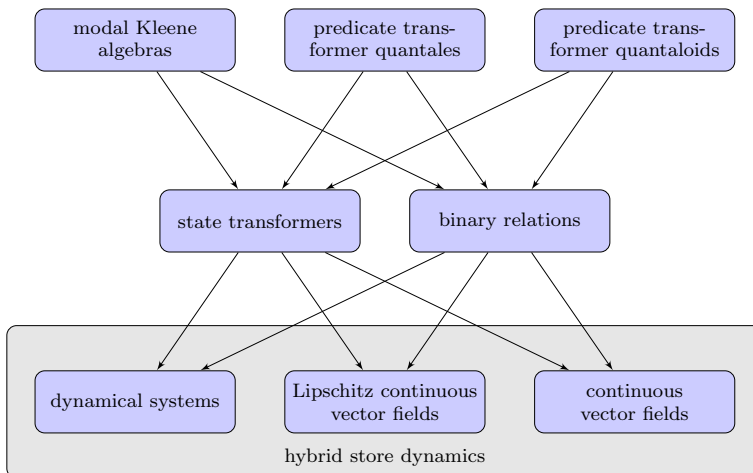

Fig. 1 Isabelle framework for hybrid systems verification

Our framework benefits from compositionality and algebra in various ways. Using algebra allows us to derive most of the semantic properties needed for verification by equational reasoning, and it reduces the overhead of developing different concrete semantics to a minimum. Using modal Kleene algebras and predicate transformer algebras, in particular, makes large parts of verification condition generation equational, and thus accessible to Isabelle's simplifiers. Compositionality of our extant framework for classical programs allows us to localise the development of concrete semantics for hybrid programs to the specification and formalisation of a semantics for evolutions commands. We only need to replace standard models of the program store by a hybrid store model. In our denotational state transformer semantics, evolution commands are interpreted as unions of all orbits of solutions of the vector field at some initial value, subject to the guards constraining the durations of evolutions. This covers situations beyond the remits of the Picard–Lindelöf theorem [22,66] and supports general reasoning about guarded invariant sets. Ultimately, we can simply plug the predicate transformers for evolution commands into the generic algebras for while programs and their rules for verification condition generation.

Verification condition generation for evolution commands is supported by three workflows that are inspired by $d\mathcal{L}$, but work differently in practice:

– The first one asks users to supply a flow and a Lipschitz constant for the vector field specified by the evolution command. We usually obtain this data using an external computer algebra system (integrating one into Isabelle seems routine and is left for future work). After certifying the flow conditions and checking Lipschitz continuity of the vector field, as dictated by the Picard–Lindelöf theorem, the orbit for the flow can be used to compute the weakest liberal preconditions for the evolution command. This workflow deviates from $d\mathcal{L}$ in allowing users to supply an interval of interest as domain of the flow.
– The second workflow works more generally in situations where unique solutions need not exist or are difficult to work with. It requires users to supply an invariant set for the vector field in the sense of dynamical systems theory [22,66]. After certifying the properties for invariant sets, a correctness specification for the evolution command and the invariant set is used in place of a weakest liberal precondition. Here, beyond $d\mathcal{L}$, we support working with solutions defined over chosen intervals and using $d\mathcal{L}$-style inference rules as well as arbitrary higher-order logic.
– The third workflow uses flows ab initio in the specification and semantic analysis of evolution commands. This circumvents checking any continuity, existence, uniqueness or invariant conditions of vector fields mentioned. This is not at all supported by $d\mathcal{L}$.

With all three workflows, hybrid program verification is ultimately performed within the concrete semantics. But, as with classical program verification, verification condition generation eliminates all structural conditions automatically so that proof obligations are entirely about the dynamics of the hybrid program store. They can be calculated in mathematical textbook style by equational reasoning, and of course by external solvers and decision procedures for arithmetic. (Their integration, as oracles or as verified components, is very important, but left for future work.) For the introductory examples presented, we have merely formalised some simple tactics that help automating the computation of derivatives in multivariate Banach spaces or that of polynomials and transcendental functions. Yet for those who prefer $d\mathcal{L}$-style reasoning, we have formalised a rudimentary set of its inference rules that are sound relative to our semantics. Overall, unlike $d\mathcal{L}$, which prescribes its domain-specific set of inference rules, we grant users the freedom of choice between various workflows and even of developing their own one within our semantic framework.

The entire framework, including the mathematical development in this article, has been formalised with Isabelle/HOL. All Isabelle components can be found in the Archive of Formal Proofs [16,18,25,65]. We are currently using them to verify hybrid programs post hoc in the standard weakest liberal precondition style outlined above. Yet the approach is flexible enough to support the verification of hybrid systems using Hoare logic [11], symbolic execution with strongest postconditions, program refinement with predicate transformers in the style of Back and von Wright [5] and Morgan [11], and reasoning about hybrid program equivalences in the elegant equational style of Kleene algebra with tests [37].

While our approach is powerful enough to tackle most problems of a recent systems competition [45], the work documented in this article focuses mainly on the semantic foundations and the proof of concept that the approach works. A more user-friendly specification language, a less simplistic hybrid store model, enhanced tactics for reasoning with flows and invariants, and mathematical background theories for reasoning about affine and linear systems of differential equations have been added, while this article has been under review [12,26]. The doctoral dissertation of the first author contains a more comprehensive description of the framework and further generalisations [27].

The remainder of this article is organised as follows: Sects. 2–6 introduce the algebras of relations, state and predicate transformers needed. Section 7 explains the shallow embedding used to formalise verification components for while programs. After recalling the basics of differential equations in Sect. 8, we introduce our semantics for evolution commands in Sects. 9–11 and explain our procedures for computing weakest liberal preconditions and reasoning with differential invariants for them. Sections 13–15 summarise the corresponding Isabelle components. Sections 12 and 16 briefly list the derivation and formalisation of semantic variants of $d\mathcal{L}$ inference rules. Section 17 presents four verification examples in our framework using the main two workflows. Section 18 presents our third workflow and a brief example for it. Sections 19 and 20 discuss related work and conclude the article. A glossary of cross-references between theorems in the text and the Isabelle theories is presented in Appendix A.

2 Kleene Algebra

This section summarises the mathematical foundations of our simplest and most developed predicate transformer algebra—modal Kleene algebra. It introduces the basics of Kleene algebras, and the state transformer model and relational model used. The relational model is standard for Kleene algebra. The state transformer model has so far received less attention and is therefore explained in detail.

A *dioid* $(S, +, \cdot, 0, 1)$ is an additively idempotent semiring, $\alpha + \alpha = \alpha$ holds for all $\alpha \in S$. The underlying abelian monoid $(S, +, 0)$ is therefore a semilattice with order defined by $\alpha \le \beta \leftrightarrow \alpha + \beta = \beta$. The order is preserved by $\cdot$ and $+$ in both arguments; 0 is its least element.

A *Kleene algebra* $(K, +, \cdot, 0, 1, {}^*)$ is a dioid expanded by the Kleene star $(-)^* : K \to K$ that satisfies the left and right unfold and induction axioms

$$1 + \alpha \cdot \alpha^* \le \alpha^*, \qquad \gamma + \alpha \cdot \beta \le \beta \to \alpha^* \cdot \gamma \le \beta,$$
$$1 + \alpha^* \cdot \alpha \le \alpha^*, \qquad \gamma + \beta \cdot \alpha \le \beta \to \gamma \cdot \alpha^* \le \beta.$$

By these axioms, $\alpha^* \cdot \gamma$ is the least fixpoint of the function $\gamma + \alpha \cdot (-)$ and $\gamma \cdot \alpha^*$ that of $\gamma + (-) \cdot \alpha$, where we use $-$ as a wildcard for function arguments. The fixpoint α^* arises as a

special case. The more general induction axioms combine its definition with sup-preservation or continuity of left and right multiplication.

Opposition is an important duality of Kleene algebras: swapping the order of multiplication in any Kleene algebra yields another one. The class of Kleene algebras is therefore closed under opposition.

Kleene algebras were conceived as algebras of regular expressions. But here we interpret their elements as programs. Addition models their nondeterministic choice, multiplication their sequential composition and the Kleene star their unbounded finite iteration. The element 0 models abort; 1 models the ineffective program. Two programs are deemed equal if they lead from the same inputs to the same outputs. These intuitions are grounded in concrete program semantics.

With the relational composition of $R \subseteq X \times Y$ and $S \subseteq Y \times Z$ defined as $(R; S) \, x \, z$ if $R \, x \, y$ and $S \, y \, z$ for some $y \in Y$, with $Id_X \, x \, y$ if $x = y$, and the reflexive-transitive closure of $R \subseteq X \times X$ defined as $R^* = \bigcup_{i \in \mathbb{N}} R^i$, where $R^0 = Id_X$ and $R^{i+1} = R; R^i$, where we write $R \, x \, y$ instead of $(x, y) \in R$, the following holds.

Proposition 2.1 *Let X be a set. Then, Rel $X = (\mathcal{P}(X \times X), \cup, ; , \emptyset, Id_X, ^*)$ forms a Kleene algebra—the* full relation Kleene algebra *over X.*

A *relation Kleene algebra* over X is thus any subalgebra of Rel X.

Opposition can be expressed in Rel X by conversion, where the *converse* of relation R is defined by $R^{\smile} \, x \, y \leftrightarrow R \, y \, x$. It satisfies in particular $(R; S)^{\smile} = S^{\smile}; R^{\smile}$.

The isomorphism $\mathcal{P}(X \times Y) \cong (\mathcal{P} \, Y)^X$ between categories of relations and non-deterministic functions—so-called *state transformers*—yields an alternative representation. It is given by the bijections $\mathcal{F} : \mathcal{P}(X \times Y) \to (\mathcal{P} \, Y)^X$ and $\mathcal{R} : (\mathcal{P} \, Y)^X \to \mathcal{P}(X \times Y)$ defined by $\mathcal{F} \, R \, x = \{y \in Y \mid R \, x \, y\}$ and by $\mathcal{R} \, f \, x \, y \Leftrightarrow y \in f \, x$. Following Isabelle syntax, we use juxtaposition with a space to denote function application. State transformers $f : X \to \mathcal{P} \, Y$ and $g : Y \to \mathcal{P} \, Z$ are composed by the (forward) Kleisli composition of the powerset monad

$$(f \circ_K g) \, x = \bigcup \{g \, y \mid y \in f \, x\}.$$

The function $\eta_X = \{-\}$ is a unit of this monad. The functors $\mathcal{F}$ and $\mathcal{R}$ preserve arbitrary sups and infs, extended pointwise to state transformers, and stars $f^{*_K} \, x = \bigcup_{i \in \mathbb{N}} f^{i_K} \, x$, which are defined with respect to Kleisli composition.

Proposition 2.2 *Let X be a set. Then, Sta $X = ((\mathcal{P} \, X)^X, \cup, \circ_K, \lambda x. \, \emptyset, \eta_X, ^{*_K})$ forms a Kleene algebra—the* full state transformer Kleene algebra *over X.*

A *state transformer Kleene algebra* over X is any subalgebra of Sta X. Opposition is now expressed using the (contravariant) functor $(-)^{op} = \mathcal{F} \circ (-)^{\smile} \circ \mathcal{R}$ that associates $f^{op} : Y \to \mathcal{P} \, X$ with every $f : X \to \mathcal{P} \, Y$.

The category **Rel**, with relations of type $X \times Y$ or state transformers of type $X \to \mathcal{P} \, Y$ as arrows, is beyond mono-type Kleene algebra.

For a more refined hierarchy of variants of Kleene algebras, their calculational properties and the most important computational models, see our formalisation in the Archive of Formal Proofs [3]. The state transformer model has been formalised with Isabelle for this article.

3 Modal Kleene Algebra

Kleene algebras must be extended to express conditionals or while loops more faithfully. This requires tests, which are not prima facie actions, but propositions. Assertions and correctness specifications cannot be expressed directly either.

Two standard extensions bring Kleene algebra closer to program semantics. Kleene algebra with tests [37] yields a simple algebraic semantics for while programs and a partial correctness semantics for these in terms of an algebraic propositional Hoare logic—ignoring assignments. Predicate transformer semantics, however, cannot be expressed [61]. Alternatively, Kleene algebras can be enriched by modal box and diamond operators in the style of propositional dynamic logic (PDL), which yields test and assertions as well as predicate transformers. Yet once again, assignments cannot be expressed within the algebra. We outline the second approach.

An *antidomain semiring* [9] is a semiring S expanded by an *antidomain operation* ad : $S \to S$ axiomatised by

$$ad\,\alpha \cdot \alpha = 0, \qquad ad\,\alpha + ad^2\,\alpha = 1, \qquad ad\,(\alpha \cdot \beta) \leq ad\,(\alpha \cdot ad^2\,\beta).$$

By opposition, an *antirange semiring* [9] is a semiring S expanded by an *antirange operation* $ar : S \to S$ axiomatised by

$$\alpha \cdot ar\,\alpha = 0, \qquad ar\,\alpha + ar^2\,\alpha = 1, \qquad ar\,(\alpha \cdot \beta) \leq ar\,(ar^2\,\alpha \cdot \beta).$$

Antidomain and antirange semirings are a fortiori dioids.

The antidomain $ad\,\alpha$ of program α models the set of those states from which α cannot be executed. The operation $d = ad^2$ thus defines the *domain* of a program: the set of those states from which it can be executed. By opposition, the antirange $ar\,\alpha$ of α yields those states into which α cannot be executed and $r = ar^2$ defines the *range* of α: those states into which it can be executed.

A *modal Kleene algebra* (MKA) [9] is a Kleene algebra that is both an antidomain and an antirange Kleene algebra in which $d \circ r = r$ and $r \circ d = d$.

In a MKA K, the set $\mathcal{P}\,ad\,K$—the image of K under ad—models the set of all tests or propositions. We henceforth often write $p, q, \dots$ for its elements. Moreover, $\mathcal{P}\,ad\,K = \mathcal{P}\,d\,K = \mathcal{P}\,r\,K = \mathcal{P}\,ar\,K = K_d = K_r$, where $K_f = \{\alpha \in S \mid f\,\alpha = \alpha\}$ for $f \in \{d, r\}$. Hence, $p \in \mathcal{P}\,ad\,K \leftrightarrow d\,p = p$. It follows that the class MKA is closed under opposition. In addition, K_d forms a Boolean algebra with least element 0, greatest element 1, join $+$, meet $\cdot$ and complementation ad—the algebra of propositions, assertions or tests.

Axiomatising MKA based on domain and range would lack the power to express complementation: K_d would only be a distributive lattice.

The programming intuitions for MKA are once again grounded in concrete semantics.

Proposition 3.1 *If X is a set, then* Rel X *is the full relation MKA over X with*

$$ad\,R\,x\,x \leftrightarrow \neg\exists y \in X.\,R\,x\,y \qquad and \qquad ar\,R = ad\,R^{\smile}.$$

Every subalgebra of a full relation MKA is a *relation* MKA.

Similarly, $ar = ad \circ (-)^{\smile}$, $d = r \circ (-)^{\smile}$ and $r = d \circ (-)^{\smile}$. Furthermore,

$$(\mathcal{P}\,(X \times X))_d = \{P \mid P \subseteq Id_X\}.$$

Henceforth, we often identify such relational subidentities, sets and predicates and their types via the isomorphisms $(\mathcal{P}\,(X \times X))_d \cong X \to \mathbb{B} \cong \mathcal{P}\,X$.

Proposition 3.2 *Let X be a set. Then, Sta X is the full state transformer MKA over X with*

$$ad\ f\ x = \begin{cases} \eta_X\ x, & \text{if } f\ x = \emptyset, \\ \emptyset, & \text{otherwise,} \end{cases} \quad and \quad ar\ f = ad\ f^{op}.$$

Every subalgebra of a full relation MKA is a *state transformer* MKA. Similarly,

$$d\ f\ x = \begin{cases} \emptyset, & \text{if } f\ x = \emptyset, \\ \eta_X\ x, & \text{otherwise,} \end{cases} \quad \text{and} \quad r\ f = d\ f^{op}.$$

These propositions generalise again beyond mono-types, but algebras of such typed relations and state transformers cannot be captured by MKA.

In every MKA, $p \cdot \alpha$ and $\alpha \cdot p$ model the domain and range restriction of α to states satisfying p. Conditionals and while loops can thus be expressed:

$$\textbf{if } p \textbf{ then } \alpha \textbf{ else } \beta = p \cdot \alpha + \bar{p} \cdot \beta \quad \text{and} \quad \textbf{while } p \textbf{ do } \alpha = (p \cdot \alpha)^* \cdot \bar{p},$$

where we write $\bar{p} = ad\ p = ar\ p$. Together with sequential composition $\alpha; \beta = \alpha \cdot \beta$, this yields an algebraic semantics of while programs without assignments. It is grounded in the relational and the state transformer semantics. A more refined hierarchy of variants of MKAs, starting from domain and antidomain semigroups, their calculational properties and the most important computational models, can be found in the Archive of Formal Proofs [16]. The state transformer model of MKA has been formalised with Isabelle for this article.

4 Modal Kleene Algebra, Predicate Transformers and Invariants

MKA can express the modal operators of PDL, both with a relational Kripke semantics and a coalgebraic state transformer semantics.

$$|\alpha\rangle p = d\ (\alpha \cdot p), \qquad |\alpha]p = ad\ (\alpha \cdot ad\ p), \qquad \langle\alpha|p = r\ (p \cdot \alpha), \qquad [\alpha|p = ar\ (ar\ p \cdot \alpha).$$

This is consistent with Jónsson and Tarski's Boolean algebras with operators [35]: Each of $|\alpha\rangle$, $\langle\alpha|$, $|\alpha]$ and $[\alpha|$ is an endofunction $K_d \to K_d$ on the Boolean algebra K_d. Yet another view of modal operators is that of predicate transformers. The function $|-]-$ yields the *weakest liberal precondition* operator wlp; $\langle-|-$ the *strongest postcondition* operator.

The boxes and diamonds of MKA are related by De Morgan duality:

$$|\alpha\rangle p = \overline{|\alpha]\bar{p}}, \qquad |\alpha]p = \overline{|\alpha\rangle\bar{p}}, \qquad \langle\alpha|p = \overline{[\alpha|\bar{p}}, \qquad [\alpha|p = \overline{\langle\alpha|\bar{p}};$$

their dualities are captured by the adjunctions and conjugations

$$|\alpha\rangle p \leq q \leftrightarrow p \leq [\alpha|q, \qquad \langle\alpha|p \leq q \leftrightarrow p \leq |\alpha]q,$$
$$|\alpha\rangle p \cdot q = 0 \leftrightarrow p \cdot \langle\alpha|q = 0, \qquad |\alpha]p + q = 1 \leftrightarrow p + [\alpha|q = 1.$$

In Rel X, as in standard Kripke semantics of modal logics in general, and of PDL in particular,

$$|R\rangle P = \{x \mid \exists y \in X.\ R\ x\ y \wedge P\ y\} \quad \text{and} \quad |R]P = \{x \mid \forall y \in X.\ R\ x\ y \to P\ y\},$$

where we identify predicates and subidentity relations. For the remaining two modalities, $\langle-| = |-\rangle \circ (-)^{\smile}$ and $[-| = |-] \circ (-)^{\smile}$. Hence, $|R\rangle P$ is the preimage of P under R and $\langle R|P$ the image of P under R. The isomorphism between subidentities, predicates and sets

also allows us to see $|R\rangle$, $\langle R|$, $|R]$ and $[R|$ as operators on the complete atomic Boolean algebra $\mathcal{P} X$, which carries algebraic structure beyond K_d that is reminiscent of a module.

In Sta X, alternatively,

$$\langle f|P = \{y \mid \exists x.\ y \in f\, x \wedge P\, x\} \quad \text{and} \quad |f]P = \{x \mid f\, x \subseteq P\}.$$

Moreover, $|-\rangle = \langle -| \circ (-)^{op}$ and $[-| = |-] \circ (-)^{op}$. Here, $\langle f|$ is the Kleisli extension of f for the powerset monad and $|f\rangle$ that of the opposite function (see Sect. 6).

The isomorphism $\mathcal{P}\,(X \times X) \cong (\mathcal{P}\,X)^X$ makes the approaches coherent:

$$|f\rangle = |\mathcal{R}\,f\rangle, \qquad |R\rangle = |\mathcal{F}\,R\rangle, \qquad |f] = |\mathcal{R}\,f], \qquad |R] = |\mathcal{F}\,R],$$

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

Predicate transformers are useful for specifying program correctness conditions and for verification condition generation. The identity

$$p \le |\alpha]q$$

captures the standard partial correctness specification for programs: if α is executed from states where precondition p holds, and if it terminates, then postcondition q holds in the states where it does. Verifying it amounts to computing $|\alpha]q$ recursively over the program structure from q and checking that the result is greater or equal to p. Intuitively, $|\alpha]q$ represents the largest set of states from which one must end up in set q when executing α, or alternatively the weakest precondition from which postcondition q must hold when executing α.

Calculating $|\alpha]q$ for straight-line programs is completely equational, but loops require invariants. To this end, one usually adds annotations to loops,

$$\textbf{while } p \textbf{ inv } i \textbf{ do } \alpha = \textbf{while } p \textbf{ do } \alpha,$$

where i is the *loop invariant* for α and calculates *wlp*s as follows [17,18]. For all $p, q, i, t \in K_d$ and $\alpha, \beta \in K$,

$$|\alpha \cdot \beta]q = |\alpha]|\beta]q, \tag{wlp-seq}$$

$$|\textbf{if } p \textbf{ then } \alpha \textbf{ else } \beta]q = (\bar{p} + |\alpha]q) \cdot (p + |\beta]q) = p \cdot |\alpha]q + \bar{p} \cdot |\beta]q, \tag{wlp-cond}$$

$$i \le |\alpha]i \rightarrow i \le |\alpha^*]i, \tag{wlp-star}$$

$$p \le i \wedge i \cdot t \le |\alpha]i \wedge i \cdot \bar{t} \le q \ \rightarrow\ p \le |\textbf{while } t \textbf{ inv } i \textbf{ do } \alpha]q. \tag{wlp-while}$$

In the rule (wlp-star), i is an invariant for the star as well. In addition, we support a while rule without an invariant annotation.

More generally, beyond loops, an element $i \in K_d$ is an *invariant* for α if it is a postfixpoint of $|\alpha]$ in K_d:

$$i \le |\alpha]i.$$

By the adjunction between boxes and diamonds, this is the case if and only if $\langle \alpha|i \le i$; that is, i is a prefixpoint of $\langle \alpha|$ in K_d. We return to this equivalence in the context of differential invariants and invariant sets of vector fields in Sect. 11. We write $\text{Inv}\,\alpha$ for the set of invariants of α.

Lemma 4.1 *In every MKA, if $i, j \in \text{Inv}\,\alpha$, then $i + j, i \cdot j \in \text{Inv}\,\alpha$.*

As a generalisation of the rule (wlp-while) for annotated while loops, we can derive a rule for commands annotated with tentative invariants α **inv** $i = \alpha$. For all $i, p, q \in K_d$ and $\alpha \in K$,

$$p \le i \wedge i \le |\alpha]i \wedge i \le q \to p \le |\alpha \text{ **inv** } i]q. \qquad \text{(wlp-cmd)}$$

Combining (wlp-cmd) with (wlp-star) then yields, for **loop** α **inv** $i = \alpha^*$,

$$p \le i \wedge i \le |\alpha]i \wedge i \le q \to p \le |\textbf{loop}\,\alpha\,\textbf{inv}\,i]q. \qquad \text{(wlp-loop-inv)}$$

We use such annotated commands for reasoning about differential invariants and loops of hybrid programs below.

The modal operators of MKA have, of course, a much richer algebra beyond verification condition generation. For a comprehensive list, see the Archive of Formal Proofs [16]. We have already derived the rules of propositional Hoare logic, which ignores assignments, and those for verification condition generation for symbolic execution with strongest postconditions in this setting [18]. A component for total correctness is also available. It supports refinement proofs in the style of Back and von Wright [5]. But this is beyond the scope of this article. The other two abstract predicate transformer algebras from Fig. 1 are surveyed in the following two sections.

5 Predicate Transformers à la Back and von Wright

While MKA has so far been our most developed setting for verifying (hybrid) programs, our framework is compositional and supports other predicate transformer algebras as well. Two of them are outlined in this and the following section. Their Isabelle formalisation [65] is discussed in Sect. 6.

The first approach follows Back and von Wright [5] in modelling predicate transformers, or simply *transformers*, as functions between complete lattices. Readers not familiar with lattice theory can freely skip this section. To obtain useful laws for program construction or verification, conditions are imposed.

A function $f : L_1 \to L_2$ between two complete lattices $(L_1, \le_1)$ and $(L_2, \le_2)$ is *order-preserving* if $x \le_1 y \to f\,x \le_2 f\,y$, *sup-preserving* if $f \circ \bigsqcup = \bigsqcup \circ \mathcal{P}\,f$ and *inf-preserving* if $f \circ \bigsqcap = \bigsqcap \circ \mathcal{P}\,f$. All sup- or inf-preserving functions are order-preserving.

We write $\mathcal{T}(L)$ for the set of transformers over the complete lattice L, and $\mathcal{T}_\le(L)$, $\mathcal{T}_\sqcup(L)$, $\mathcal{T}_\sqcap(L)$ for the subsets of order-, sup- and inf-preserving transformers. Obviously, $\mathcal{T}_\sqcap(L) = \mathcal{T}_\sqcup(L^{op})$. The following fact is well known [5,15].

Proposition 5.1 *Let X be a set, and let L be a complete lattice. Then, L^X forms a complete lattice with order and sups extended pointwise.*

Infs, least and greatest elements can then be defined from sups on L^X as usual. Function spaces L^L, in particular, form monoids with respect to function composition $\circ$ and id_L. In addition, $\circ$ preserves sups and infs in its first argument, but not necessarily in its second one. Algebraically, this is captured as follows.

A *near-quantale* $(Q, \le, \cdot)$ is a complete lattice $(Q, \le)$ with an associative composition $\cdot$ that preserves sups in its first argument. It is *unital* if composition has a unit 1. A *prequantale* is a near-quantale in which composition is order-preserving in its second argument. A *quantale* is a near quantale in which composition preserves sups in its second argument. See [56] for more information about quantales.

Proposition 5.2 *Let L be a complete lattice. Then,*

1. *$\mathcal{T}(L)$ and $\mathcal{T}(L^{op})$ form unital near-quantales;*
2. *$\mathcal{T}_{\leq}(L)$ ($\mathcal{T}_{\leq}(L^{op})$) forms a unital sub-prequantale of $\mathcal{T}(L)$ ($\mathcal{T}(L^{op})$);*
3. *$\mathcal{T}_{\sqcup}(L)$ ($\mathcal{T}_{\sqcap}(L)$) forms a unital subquantale of $\mathcal{T}_{\leq}(L)$ ($\mathcal{T}_{\leq}(L^{op})$).*

Transformers for while loops are obtained by connecting quantales with Kleene algebras. This requires fixpoints of $\varphi_{\alpha\gamma} = \gamma \sqcup \alpha \cdot (-)$ and $\varphi_{\alpha} = 1 \sqcup \alpha \cdot (-)$ as well as the Kleene star $\alpha^* = \bigsqcup_{i \in \mathbb{N}} \alpha^i$. A *left Kleene algebra* is a dioid in which φ has a least fixpoint that satisfies $lfp\, \varphi_{\alpha\gamma} = lfp\, \varphi_{\alpha} \cdot \gamma$. Hence, φ_{α} satisfies the left unfold and left induction axioms $1 \sqcup \alpha \cdot \varphi_{\alpha} \leq \varphi_{\alpha}$ and $\gamma \sqcup \alpha \cdot \beta \leq \beta \rightarrow \varphi_{\alpha} \cdot \gamma \leq \beta$. By opposition, a *right Kleene algebra* is a dioid in which the least fixpoint of a dual function $1 \sqcup (-) \cdot \alpha$ satisfies the right unfold and right induction axioms.

Proposition 5.3

1. *Every near-quantale is a right Kleene algebra with $lfp\, \varphi_{\alpha} = \alpha^*$.*
2. *Every prequantale is also a left Kleene algebra.*
3. *Every quantale is a Kleene algebra with $lfp\, \varphi_{\alpha} = \alpha^*$.*

The proofs of (1) and (3) use sup-preservation and Kleene's fixpoint theorem. That of (2) uses the Knaster–Tarski fixpoint theorem to show that $\varphi_{\alpha\gamma}$ has a least fixpoint, and fixpoint fusion [44] to derive $lfp\, \varphi_{\alpha\gamma} = lfp\, \varphi_{\alpha} \cdot \gamma$, which yields the left Kleene algebra axioms. In prequantales, $lfp\, \varphi_{\alpha} \cdot \gamma \leq \alpha^* \cdot \gamma$; equality generally requires sup-preservation in the first argument of composition.

The fixpoint and iteration laws on functions spaces, which follow from Propositions 5.3 and 5.2, still need to be translated into laws for transformers operating on the underlying lattice. This is achieved again by fixpoint fusion [5]. In $\mathcal{T}_{\leq}(L)$,

$$lfp\,(\lambda g.\, id_L \sqcup f \circ g)\, x = lfp\,(\lambda y.\, x \sqcup f\, y),$$

and *lfp* preserves isotonicity. In $\mathcal{T}_{\sqcup}(L)$, moreover,

$$f\, x \leq x \rightarrow f^*\, x \leq x,$$

$id_L \sqcup f \circ f^* = f^* = f^* \circ f \sqcup id_L$ and $(-)^*$ preserves sups. All results dualise to inf-preserving transformers.

Relative to MKA, backward diamonds correspond to sup-preserving forward transformers and forward boxes to inf-preserving backward transformers in the opposite quantale, where the lattice has been dualised and the order of composition been swapped. An analogous correspondence holds for forward diamonds and backward boxes. Sup- and inf-preserving transformers over complete lattices are less general than MKA in that preservation of arbitrary sups or infs is required, whereas that of MKA is restricted to finite sups and infs. Isotone transformers, however, are more general, as not even finite sups or infs need to be preserved, and finite sup- or inf-preservation implies order preservation.

We are mainly using the *wlp* operator for verification condition generation and hence briefly outline *wlp*s for conditionals and loops in this setting. We assume that the underlying lattice L is a complete Boolean algebra, that is, a complete lattice as well as a complemented distributive lattice. We can then lift elements of L to *wlp*s as $|p]q = p \rightarrow q$ and define, in $\mathcal{T}_{\leq}(L^{op})$,

if p **then** f **else** $g = |p] \circ f \sqcap |\bar{p}] \circ g$ and   **while** p **do** $f = lfp\, \varphi_{|p]f} \circ |\bar{p}]$.

In $\mathcal{T}_\sqcap(L)$, we even obtain

$$\textbf{while } p \textbf{ do } f = (|p] \circ f)^* \circ |\bar{p}].$$

These equations allow generating verification conditions as with (wlp-cond) and (wlp-while) from Sect. 4. Overall, our Isabelle components for lattice-based predicate transformers in the Archive of Formal Proofs [65] contain essentially the same equations and rules for verification condition generation as those for MKA.

We have so far restricted the approach to endofunctions on a complete lattice to relate it to MKA. Yet it generalises to functions in $L_2^{L_1}$ and hence to categories [5]. The corresponding poly-typed generalisations of quantales are known as *quantaloids* [55]. In particular, composition is then a partial operation.

6 Predicate Transformers from the Powerset Monad

A second, more coalgebraic approach to predicate transformers starts from monads [41]. In addition, it details the relational and state transformer semantics of MKA in a more modern algebraic approach. We need to assume basic knowledge of categories and monads. Once again, readers unfamiliar with these concepts can freely skip it.

Recall that $(\mathcal{P}, \eta_X, \mu_X)$, for $\mathcal{P} : \textbf{Set} \to \textbf{Set}$, $\eta_X : X \to \mathcal{P}\,X$ defined by $\eta_X = \{-\}$ and $\mu_X : \mathcal{P}^2\,X \to \mathcal{P}\,X$ defined by $\mu_X = \bigcup$ is the monad of the powerset functor in the category **Set** of sets and functions. The morphisms η and μ are natural transformations. They satisfy, for every $f : X \to Y$,

$$\eta_Y \circ id\,f = \mathcal{P}\,f \circ \eta_X \qquad \text{and} \qquad \mu_Y \circ \mathcal{P}^2\,f = \mathcal{P}\,f \circ \mu_X.$$

From the monadic point of view, state transformers $X \to \mathcal{P}\,Y$ are arrows $X \to Y$ in the Kleisli category $\textbf{Set}_{\mathcal{P}}$ of $\mathcal{P}$ over **Set**. They are composed by (forward) Kleisli composition $f \circ_K g = \mu \circ \mathcal{P}\,g \circ f$ as explained before Proposition 2.2 in Sect. 2. The category $\textbf{Set}_{\mathcal{P}}$ is known to be isomorphic to **Rel**, the category of sets and binary relations.

The isomorphism between state and forward predicate transformers is based on the contravariant functor $(-)^\dagger : \textbf{Set}_{\mathcal{P}}(X, \mathcal{P}\,Y) \to \textbf{Set}_{\mathcal{P}}(\mathcal{P}\,X, \mathcal{P}\,Y)$—the Kleisli extension. Its definition $f^\dagger = \mu \circ \mathcal{P}\,f$ implies that $(-)^\dagger = \langle-|$ on morphisms, which is the strongest postcondition operator.

The structure of state spaces—Boolean algebras for MKA, complete lattices in Back and von Wright's approach—is captured by the Eilenberg–Moore algebras of the powerset monad. It is well known that $(-)^\dagger$ embeds $\textbf{Set}_{\mathcal{P}}$ into their category. Its objects are complete (sup-semi)lattices; its morphisms sup-preserving functions, hence transformers. More precisely, $(-)^\dagger$ embeds into powerset algebras, complete atomic Boolean algebras that are the free objects in this category.

The isomorphism $\textbf{Set}_{\mathcal{P}}(X, \mathcal{P}\,Y) \cong \textbf{Set}^{\sqcup}(\mathcal{P}\,X, \mathcal{P}\,Y)$ between state transformers and sup-preserving predicate transformers then arises as follows. The embedding $\langle-|$ has an injective inverse $\langle-|^{-1}$ on the subcategory of sup-preserving transformers. It is defined by $\langle-|^{-1} = (-) \circ \eta$, which can be spelled out as $\langle\varphi|^{-1}\,x = \{y \mid y \in \varphi\,\{x\}\}$. The isomorphism preserves the quantaloid structures of state and predicate transformers that is, compositions (contravariantly), units and sups, hence least elements, but not necessarily infs and greatest elements. These results extend to $\textbf{Set}^{\sqcup}(\mathcal{P}\,X, \mathcal{P}\,Y) \cong \textbf{Rel}(X, Y)$ via $\textbf{Set}_{\mathcal{P}} \cong \textbf{Rel}$. In addition, predicate transformers $\langle f| : \mathcal{P}\,X \to \mathcal{P}\,Y$ preserve of course sups in powerset lattices, hence least elements, but not necessarily infs and greatest elements.

Forward boxes or *wlp*s can be obtained from state transformers via a (covariant) functor $|-| : \mathbf{Set}_{\mathcal{P}}(X, \mathcal{P}\,Y) \to \mathbf{Set}(\mathcal{P}\,Y, \mathcal{P}\,X)$, embedding Kleisli arrows into the opposite of the category of Eilenberg–Moore algebras formed by complete (inf-semi)lattices and inf-preserving functions. It is defined on morphisms as $|-| = \partial_F \circ \langle -| \circ (-)^{op}$, where $\partial_F\,f = \partial \circ f \circ \partial$ and ∂ dualises the lattice. Unfolding definitions, once again $|f|\,P = \{x \mid f\,x \subseteq P\}$.

Furthermore, its inverse $|-|^{-1}$ on the subcategory of inf-preserving transformers is $|\varphi|^{-1}\,x = \bigcap\{P \mid x \in \varphi\,P\}$. The duality $\mathbf{Set}_{\mathcal{P}}(X, \mathcal{P}\,Y) \cong \mathbf{Set}^{\sqcap}(\mathcal{P}\,Y, \mathcal{P}\,X)$ reverses Kleisli arrows and preserves the quantaloid structures up to lattice duality, mapping sups to infs and vice versa. It extends to relations as before. In addition, predicate transformers $|f|$ preserve of course infs of powerset lattices, hence greatest elements, but not necessarily sups and least elements.

The remaining transformers $|-\rangle$ and $[-|$ and their inverses arise from $\langle -|$ and $|-|$ by opposition: $|-\rangle = \langle -| \circ (-)^{op}$, $|-\rangle^{-1} = (-)^{op} \circ \langle -|^{-1}$, $[-| = |-| \circ (-)^{op}$ and $[-|^{-1} = (-)^{op} \circ |-|^{-1}$. Taken together, the four modal operators satisfy the laws of the MKA modalities outlined in Sect. 4 and those of the abstract sup/inf-preserving transformers discussed in Sect. 5. They give in fact semantics to the algebraic developments, when restricted to mono-types, and once again yield the same rules for verification condition in the state transformer and the relational semantics, albeit in a more general categorical setting.

The categorical approach to predicate transformers outlined is not new, apart perhaps from the emphasis on quantales and quantaloids. The emphasis on monads is due at least to Manes [43]. More recently, Jacob's work on state-and-effect triangles [32] has explored similar connections and their generalisation beyond sequential programs. A formalisation with Isabelle, which is further discussed in Sect. 13, is a contribution of this article.

7 Assignments

Two important ingredients for concrete program semantics and verification condition generation are still missing: a mathematical model of the program store and program assignments, and rules for calculating *wlp*s for these basic commands. To prepare for hybrid programs (see Sect. 9 for a syntax), we model stores and assignments as discrete dynamical systems over state spaces.

Formally, a *dynamical system* [4,66] is an *action* of a monoid $(M, \star, u)$ on a set or state space S, that is, a monoid morphism $\varphi : M \to S \to S$ into the *transformation monoid* $(S^S, \circ, id_S)$ on S^S. Thus, by definition,

$$\varphi\,(m \star n) = (\varphi\,m) \circ (\varphi\,n) \qquad \text{and} \qquad \varphi\,u = id_S.$$

The first action axiom captures the inherent determinism of dynamical systems. Conversely, each transformation monoid $(S^S, \circ, id_S)$ determines a monoid action in which the action $\varphi : S^S \to S \to S$ is function application.

States of simple while programs can be modelled simply as maps $s : V \to E$ from program variables in V to values in E. State spaces for such discrete dynamical systems are function spaces $S = E^V$.

An update function $f_a : V \to (S \to E) \to S \to S$ for assignment commands can be defined as

$$f_a\,v\,e\,s = s[v \mapsto e\,s],$$

where $f[a \mapsto b]$ updates $f : A \to B$ by associating $a \in A$ with b and every $y \neq a$ with $f\,y$. The "expression" $e : S \to E$ is evaluated in state s to $e\,s$. The maps $f_a\,v\,e$ generate a

transformation monoid, hence a monoid action $S^S \to S \to S$ on S^S. They also connect the concrete program store semantics with the *wlp* semantics used for verification condition generation.

We lift $f_a\, v\, e : S \to S$ to a state transformer $v :=_{\mathcal{F}} e : S \to \mathcal{P}\, S$ as

$$(v :=_{\mathcal{F}} e) = \eta_S \circ (f_a\, v\, e) = \lambda s.\, \{f_a\, v\, e\, s\},$$

thus creating a semantic illusion for syntactic assignment commands in the MKA Sta S. For Rel S, the isomorphism between $\mathbf{Set}_{\mathcal{P}}$ and $\mathbf{Rel}$ yields

$$(v :=_{\mathcal{R}} e) = \mathcal{R}\, (v :=_{\mathcal{F}} e),$$

hence $(v :=_{\mathcal{R}} e) = \{(s, f_a\, v\, e\, s) \mid s \in E^V\} = \{(s, s[v \mapsto e\, s]) \mid s \in E^V\}$. Alternatively, we could have defined the state transformer semantics from the relational one via $(v :=_{\mathcal{F}} e) = \mathcal{F}\, (v :=_{\mathcal{R}} e)$.

The *wlp*s for assignment commands in Rel S and Sta S are of course the same. Hence, we drop the indices $\mathcal{F}$ and $\mathcal{R}$ and write

$$|v := e]Q = \lambda s.\, Q\, (s[v \mapsto e\, s]) = \lambda s.\, Q\, (f_a\, v\, e\, s). \qquad \text{(wlp-asgn)}$$

Adding the *wlp* law for assignments in either semantics to the algebraic ones for the program structure suffices to generate data-level verification conditions for while programs.

The approach outlined so far is suited for building verification components via shallow embeddings with proof assistants such as Isabelle. The predicate transformer algebras of the previous sections, as shown in the first row of Fig. 1, can all be instantiated to intermediate state transformer and relational semantics, as shown in Propositions 3.1 and 3.2 for MKA. These form the second row in Fig. 1. Each of these can be instantiated further to concrete semantics with predicate transformers for assignments, as described in this section.

In Isabelle, these instantiations are enabled by type polymorphism. If modal Kleene algebras have type $'a$, then the intermediate semantics have the type of relations or state transformers over $'a$, and Propositions 3.1 and 3.2 can be formalised, so that all facts known for MKA are available in the intermediate semantics. The concrete semantics then require another simple instantiation of the types of relations or state transformers to those of program stores. All facts known for MKA and the two intermediate semantics are then available in the concrete predicate transformer semantics for while programs. A particularity of the semantic approach and the shallow embedding is that assignment semantics are based on function updates instead of substitutions—see the rule (wlp-asgn)—so that an explicit substitution calculus like that of d$\mathcal{L}$ is not needed. We can simply rely on that of Isabelle/HOL.

The use of algebra and the modularity of the shallow semantics simplify the construction of program verification components [18] considerably. The overall approach discussed has been developed initially for Hoare logics in [2]. It has been extended to predicate transformer semantics based on MKA in [17].

8 Ordinary Differential Equations

Before developing relational and state transformer models for the basic evolution commands of hybrid programs in the next section, we briefly review some basic facts about continuous dynamical systems and ordinary differential equations.

Continuous dynamical systems $\varphi : T \to S \to S$ are *flows*, which often represent solutions to systems of ordinary differential equations (ODEs) [4,22,66]. They are called continuous

because T, which models time, is assumed to form a non-discrete submonoid of $(\mathbb{R}, +, 0)$, and the state space or phase space S is usually a manifold with topological structure. By definition, flows are monoid actions. Hence, φ satisfies, for all $t_1, t_2 \in T$,

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

We always assume that T is an open interval in $\mathbb{R}$ and S an open subset of $\mathbb{R}^n$. Beyond that, one usually assumes that actions are compatible with the structure on S. As S is a manifold, we assume that flows are continuously differentiable.

The *trajectory* of φ through state $s \in S$ is the function $\varphi_s : T \to S$ defined by $\varphi_s = \lambda t.\,\varphi\,t\,s$, that is, $\varphi_s\,t = \varphi\,t\,s$. It describes the system's evolution in time passing through state s.

The *orbit* of s is the set of all states on the trajectory passing through s, but not necessarily starting in this state. We model it as the function $\gamma^\varphi : S \to \mathcal{P}\,S$ defined by

$$\gamma^\varphi\,s = \mathcal{P}\,\varphi_s\,T,$$

the canonical map sending each $s \in S$ to its equivalence class $\gamma^\varphi\,s$. Orbit functions are state transformers, as their type indicates. They form our basic semantics for evolution commands and hybrid programs.

Flows arise from ODEs as follows. In a system of ODEs

$$x_i'\,t = f_i\,(t, (x_1\,t), \ldots, (x_n\,t)), \qquad (1 \le i \le n),$$

each f_i is a continuous real-valued function and $t \in T \subseteq \mathbb{R}$. Any such system can be made time-independent—or *autonomous*—by adding the equation $x_0'\,t = 1$. We henceforth restrict our attention to autonomous systems and write

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

The continuous function $f : S \to S$ on $S \subseteq \mathbb{R}^n$ is a *vector field*. It assigns a vector to each point in S.

An autonomous system of ODEs is thus simply a vector field f, and a *solution* a continuously differentiable function $X : T \to S$ that satisfies $X'\,t = f\,(X\,t)$ for all $t \in T$, or more briefly $X' = f \circ X$.

An *initial value problem* (IVP) is a pair (f, s) of a vector field f and an initial value $(0, s) \in T \times S$ [22,66], where $t_0 = 0$ and s represent the initial time and initial state of the system. A solution to the IVP (f, s) satisfies

$$X' = f \circ X \qquad \text{and} \qquad X\,0 = s.$$

If solutions X to an IVP (f, s) are unique and $T = \mathbb{R}$, then it is easy to show that $X = \varphi_s^f$ is the trajectory of the flow φ^f through s.

Geometrically, φ_s^f is the unique curve in S that is parametrised by t, passes through s and is tangential to f at any point. As trajectories arise from integrating both sides of $(\varphi_s^f)' = f \circ \varphi_s^f$, they are also called *integral curves*. We henceforth write φ_s when the dependency on f is clear.

The following example provides some physical intuition for readers unfamiliar with these concepts.

Example 8.1 *(Particles in fluid)* We use the autonomous system of ODEs

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

where $v \in \mathbb{R} \setminus \{0\}$ is a constant, as a simple model for the movement of particles in a three-dimensional fluid. Its vector field $f : \mathbb{R}^3 \to \mathbb{R}^3$,

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

associates a velocity vector with each point of $S = \mathbb{R}^3$ (vectors in Fig. 2).

For each point $s = (s_1, s_2, s_3)^T$, the solutions $\varphi_s : \mathbb{R} \to \mathbb{R}^3$ of the IVP (f, s) are uniquely defined. They are the trajectories of particles through time passing through state s (dot and line in Fig. 2), given by

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

where we use juxtaposition without spaces as multiplication of real numbers.

Checking that they are indeed solutions to the IVP requires simple calculations:

$$\varphi_s'\, t = \begin{pmatrix} v \\ 0 \\ -\sin(s_1+vt) \end{pmatrix} = f \begin{pmatrix} s_1 + vt \\ s_2 \\ s_3 + \frac{\cos(s_1+vt)}{v} - \frac{\cos s_1}{v} \end{pmatrix} = f(\varphi_s\, t),$$

$$\varphi_s\, 0 = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \end{pmatrix} + \begin{pmatrix} v0 \\ 0 \\ \frac{\cos(s_1+v0)}{v} - \frac{\cos s_1}{v} \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \end{pmatrix} = s.$$

Checking that $\varphi : \mathbb{R} \to \mathbb{R}^3 \to \mathbb{R}^3$, $\varphi\, t\, s = \varphi_s\, t$, is a flow is calculational, too:

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

$$= \begin{pmatrix} s_1 \\ s_2 \\ s_3 \end{pmatrix} + \begin{pmatrix} v(t_1 + t_2) \\ 0 \\ \frac{\cos(s_1+v(t_1+t_2))}{v} - \frac{\cos s_1}{v} \end{pmatrix}$$

$$= \varphi\, (t_1 + t_2)\, s.$$

The condition $\varphi\, 0\, s = s$ has already been checked. $\qquad\square$

It is well known that not all IVPs admit flows: not all of them have unique solutions, and in many situations, flows exist locally on a subset of $\mathbb{R}$ that does not form a submonoid. Peano's theorem guarantees the local existence of solutions for systems of ODEs whose associated vector field is continuous. Conditions for local existence and uniqueness are provided by the Picard–Lindelöf theorem [22,66], which we briefly discuss, as we use it for our first workflow.

By the fundamental theorem of calculus, any solution to an IVP must satisfy

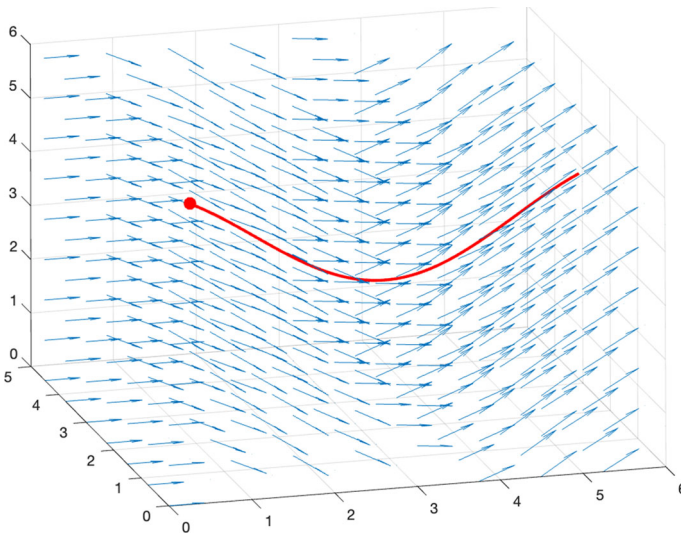$$X\, t - X\, 0 = \int_0^t f(X\, \tau)\, d\tau.$$

Fig. 2 Vector field and trajectory for a particle in a fluid (Example 8.1)

It can be shown that this equation holds if, for $X\,0 = s$, the function

$$h\,x\,t = s + \int_0^t f\,(x\,\tau)\,\mathrm{d}\tau$$

has a fixpoint. This, in turn, is the case if the limit X of the sequence $(h^n)_{n\in\mathbb{N}}$, defined by $h^0\,x\,t = s$ and $h^{n+1} = h \circ h^n$, exists. Indeed, with this assumption,

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

using continuity of addition, integration and f in the second step. Finally, existence of the limit of $(h^n)_{n\in\mathbb{N}}$ is guaranteed by constraining the domain of the h^n, and by Banach's fixpoint theorem, there must be a *Lipschitz constant* $\ell \geq 0$ such that

$$\|f\,s_1 - f\,s_2\| \leq \ell\|s_1 - s_2\|,$$

for any $s_1, s_2 \in S$, where $\|-\|$ is the Euclidean norm on $\mathbb{R}^n$. Vector fields satisfying this condition are called *Lipschitz continuous*.

Theorem 8.2 (Picard–Lindelöf) *Let $S \subseteq \mathbb{R}^n$ be an open set and $f : S \to S$ a Lipschitz continuous vector field. The IVP (f, s) then has a unique solution $X : T_s \to S$ on some open interval $T_s \subseteq \mathbb{R}$.*

The Picard–Lindelöf theorem makes it possible to patch together intervals T_s to a set $U = \bigcup_{s\in S} T_s \times \{s\} \subseteq \mathbb{R} \times S$, from which a largest interval of existence $T = \bigcup_{s\in S} T_s$ can be extracted. One can then define a *local flow* $\varphi : T \to S \to S$ such that $\varphi_s\,t$ is the maximal integral curve at s. The monoid action identities $\varphi\,0 = id$ and $\varphi\,(t_1 + t_2)\,s = \varphi\,t_1\,(\varphi\,t_2\,s)$ can thus be shown for all $t_2, t_1 + t_2 \in T_s$ [66], but U need not be closed under addition. The Picard–Lindelöf theorem, in the form presented, thus provides sufficient conditions for the existence and uniqueness of local flows for autonomous systems of ODEs. Flows are global and hence monoid actions if T is equal to $\mathbb{R}$ or its nonnegative or non-positive subset.

Hybrid systems often deal with dynamical systems where $T = T_s = \mathbb{R}$ for any $s \in S$ and S is isomorphic to $\mathbb{R}^n$ for some $n \in \mathbb{N}$. Our approach supports local flows with $T \subset \mathbb{R}$ and $S \subset \mathbb{R}^n$ as well, and even IVPs with multiple solutions beyond the realm of Picard–Lindelöf.

9 Evolution Commands for Lipschitz Continuous Vector Fields

Simple hybrid programs of $\mathsf{d}\mathcal{L}$ [47] are defined by the syntax

$$\mathcal{C} ::= \; x := e \mid x' = f \;\&\; G \mid ?P \mid \mathcal{C}; \mathcal{C} \mid \mathcal{C} + \mathcal{C} \mid \mathcal{C}^*,$$

which adds *evolution commands* $x' = f \;\&\; G$ to the program syntax of dynamic logic. Intuitively, evolution commands introduce a vector field f for an autonomous system of ODEs and a *guard* G, which models boundary conditions or similar constraints that restrict temporal evolutions. Guards are also known as *evolution domain restrictions* or *invariants* in the hybrid automata literature [10], but henceforth we consistently refer to them as "guards". Nondeterministic choice and finite iteration can be adapted for modelling conditionals and while loops as with MKA or predicate transformer semantics.

We are only interested in the semantics of hybrid programs. Relative to the semantics of standard while programs, it thus remains to define the *wlp*s for evolution commands. This requires relational and state transformer semantics for evolution commands over hybrid program stores. In this section, we describe our first workflow that certifies solutions using the Picard–Lindelöf theorem. We thus assume that vector fields are Lipschitz continuous, such that the Picard–Lindelöf theorem guarantees at least local flows. This is more general than needed for dynamical systems. A further generalisation to continuous vector fields is presented in the next section in preparation for our second, more powerful workflow.

We begin with hybrid program stores for $\mathsf{d}\mathcal{L}$ [46]. These are maps $s : V \to \mathbb{R}$ that assign real numbers to program variables in V. Variables may appear both in differential equations and the discrete control of a hybrid system. One usually assumes that $|V| = n$ for some $n \in \mathbb{N}$, which makes $\mathbb{R}^V$ isomorphic to the vector space $\mathbb{R}^n$. The results from Sect. 8 then apply to any state space $S \subseteq \mathbb{R}^V$.

Next, we describe a state transformer semantics and a $\mathsf{d}\mathcal{L}$-style relational semantics of evolution commands with Lipschitz continuous vector fields. Intuitively, the semantics of $x' = f \;\&\; G$ in state $s \in S \subseteq \mathbb{R}^V$ is the longest segment of the trajectory φ_s^f at s along which all points satisfy G.

For the remainder of this section, we fix a Lipschitz continuous vector field $f : S \to S$ and a guard $G : S \to \mathbb{B}$, for $S \subseteq \mathbb{R}^V$. We freely consider G, and any other function of that type, as a set or a predicate. As explained in Sect. 8, there is a (local) flow $\varphi : T \to S \to S$ defined on a maximal interval $T \subseteq \mathbb{R}$ with $0 \in T$. Thus, we can pick any interval $U \subseteq T$ with $0 \in U$ to compute *wlp*s over subintervals of the interval of existence T. In examples, we typically use the subinterval $[0, t]$, from the time at which the system dynamics starts to a maximal time t of interest, or the subinterval $\mathbb{R}_+$, the set of nonnegative real numbers.

For each $t \in U$, let $\downarrow t = \{t' \in U \mid t' \leq t\}$. The *G-guarded orbit* on U at $s \in S$ is then defined as $\gamma_{G,U}^\varphi : S \to \mathcal{P} S$ by

$$\gamma_{G,U}^\varphi \, s = \bigcup \{\mathcal{P} \, \varphi_s \downarrow t \mid t \in U \wedge \mathcal{P} \, \varphi_s \downarrow t \subseteq G\}.$$

Intuitively, $\gamma_{G,U}^\varphi \, s$ is the orbit at s defined along the longest interval of time in U that satisfies guard G. This intuition is more apparent in the following lemma.

Lemma 9.1 *Let $s \in S$. Then,*

1. $\gamma_{G,U}^{\varphi} \, s = \bigcup \{ \gamma^{\varphi | \downarrow t} \, s \mid t \in U \wedge \gamma^{\varphi | \downarrow t} \, s \subseteq G \}$,
2. $\gamma_{G,U}^{\varphi} \, s = \{ \varphi_s \, t \mid t \in U \wedge \forall \tau \in \downarrow t. \, G \, (\varphi_s \, \tau) \}$.

We have not formalised (1) with Isabelle because reasoning with partial functions may be tedious. As a special case, for $U = T_+$, any subinterval of $\mathbb{R}_+$,

$$\gamma_{G,T_+}^{\varphi} \, s = \{ \varphi_s \, t \mid t \in T_+ \wedge \forall \tau \in [0, t]. \, G \, (\varphi_s \, \tau) \}.$$

We can now define the state transformer semantics of $x' = f \, \& \, G$ simply as

$$(x' =_{\mathcal{F}} f \, \& \, G)_U = \gamma_{G,U}^{\varphi}.$$

Hence, the denotation of an evolution command in state s is the guarded orbit at s in time interval U. Alternatively, in Rel S,

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

like in Sect. 7. Restricting this further to $U = \mathbb{R}_+$ yields the standard semantics of evolution commands of d$\mathcal{L}$.

It remains to derive the *wlp*s for evolution commands. These are the same in Rel S and Sta S, so we drop $\mathcal{F}$ and $\mathcal{R}$.

Proposition 9.2 *Let $Q : S \to \mathbb{B}$. Then,*

$$|(x' = f \, \& \, G)_U] Q = \lambda s \in S. \, \{ s \mid \forall t \in U. \, \mathcal{P} \, \varphi_s \, \downarrow t \subseteq G \to \mathcal{P} \, \varphi_s \, \downarrow t \subseteq Q \}.$$

By Lemma 9.1, alternatively,

$$|(x' = f \, \& \, G)_U] Q = \lambda s \in S. \, \{ s \mid \forall t \in U. \, \gamma^{\varphi | \downarrow t} \, s \subseteq G \to \gamma^{\varphi | \downarrow t} \, s \subseteq Q \}.$$

For verification condition generation, the following variant is most useful.

Lemma 9.3 *Let $Q : S \to \mathbb{B}$. Then,*

$$|(x' = f \, \& \, G)_U] Q = \lambda s \in S. \forall t \in U. \, (\forall \tau \in \downarrow t. \, G \, (\varphi_s \, \tau)) \to Q \, (\varphi_s \, t). \qquad \text{(wlp-evl)}$$

In particular, for $T = \mathbb{R}$ and $U = \mathbb{R}_+$,

$$|(x' = f \, \& \, G)_{\mathbb{R}_+}] Q = \lambda s \in S. \forall t \in \mathbb{R}_+. \, (\forall \tau \in [0, t]. \, G \, (\varphi_s \, \tau)) \to Q \, (\varphi_s \, t).$$

Accordingly, and consistently with d$\mathcal{L}$, Q is no longer a postcondition in the traditional sense: by definition, it is supposed to hold along the trajectory and therefore on any orbit at any particular initial condition s guarded by G.

For a more categorical view on the *wlp* of evolution commands, remember from Sect. 6 that $\langle (x' = f \, \& \, G)_U | = (\gamma_{G,U}^{\varphi})^{\dagger}$, where $(-)^{\dagger}$ is the Kleisli extension map, and that the *wlp* of $(x' = f \, \& \, G)_U$ is its right adjoint. It therefore satisfies

$$|(x' = f \& G)_U] P = \bigcup \{Q \mid (\gamma_{G,U}^{\varphi})^\dagger \, Q \subseteq P\} = \{s \mid \gamma_{G,U}^{\varphi} \, s \subseteq P\}.$$

The identity in Proposition 9.2 can then be calculated from there.

The *wlp* laws in Proposition 9.2 and Lemma 9.3 complete the laws for verification condition generation for hybrid programs in the relational and state transformer semantics. In practice, Proposition 9.2, Lemma 9.3 and the Picard–Lindelöf theorem support our first workflow for computing the *wlp* of an evolution command $x' = f \& G$ on a set U for a Lipschitz continuous vector field:

1. check that the vector field f is indeed Lipschitz continuous and $S \subseteq R^V$ open;
2. supply the (local) flow φ for f with U, a subinterval of the interval of existence around 0;
3. certify that φ_s is indeed the unique solution for (f, s) for any $s \in S$ and for U:

   (a) $\varphi_s' = f \circ \varphi_s$ on U for any $s \in S$,
   (b) $\varphi_s \, 0 = s$ for any $s \in S$,
   (c) U is subset of open set T with $0 \in U$;

4. if successful, apply the identity in Proposition 9.2 or Lemma 9.3.

In practice, computer algebra tools are helpful for finding flows. Their integration into proof assistants for this purpose is routine and therefore not pursued in this article. The existence of unique solutions can be guaranteed uniformly, for instance, for affine or linear systems of ordinary differential equations. See [26] for the formalisation of such an approach with Isabelle.

The following classical example illustrates our algebraic approach and gives a first glimpse of the mathematics involved. It should be noted that we are not embellishing our natural semantical notation with any façade program syntax in this article; see [12] for such an extension. A formal verification with Isabelle can be found in Example 17.1.

Example 9.4 (Bouncing ball) A ball of mass m is dropped from height $h \geq 0$. Its state space is $s \in \mathbb{R}^V$ for $V = \{x, v\}$, where x denotes its position and v its velocity. Its kinematics is specified by the vector field $f : \mathbb{R}^V \to \mathbb{R}^V$ with

$$f \begin{pmatrix} s_x \\ s_v \end{pmatrix} = \begin{pmatrix} s_v \\ -g \end{pmatrix},$$

where g is the acceleration due to gravity and we abbreviate $s_x = s\,x$ and $s_v = s\,v$. The ball is assumed to bounce back from the ground in an elastic collision. This is modelled using a discrete control, which checks for $s_x = 0$ and then flips the velocity. A guard $G = (\lambda s. \, s_x \geq 0)$ precludes any motion below the ground. The system is modelled by the hybrid program [47]

$$\begin{aligned} \mathsf{Cntrl} &= \mathbf{if} \ (\lambda \, s. \, s_x = 0) \ \mathbf{then} \ v := (\lambda \, s. \, - s_v) \ \mathbf{else} \ skip, \\ \mathsf{Ball} &= (x' = f \& G \, ; \mathsf{Cntrl})^*, \end{aligned}$$

where *skip* denotes the program that maps each state to itself (represented by 1 in MKA). Its correctness specification is

$$P \leq |\mathsf{Ball}] Q \qquad \text{for} \quad P = (\lambda s. \, s_x = h \wedge s_v = 0) \quad \text{and} \quad Q = (\lambda s. \, 0 \leq s_x \leq h).$$

We also need the loop invariant

$$I = \left(\lambda s.\ 0 \le s_x \wedge \frac{1}{2} s_v^2 = g(h - s_x) \right),$$

which uses a variant of energy conservation with m cancelled out.

The first step of our verification proof shows that $P \le I$ and $I \le Q$. The first inequality holds because $\frac{1}{2} 0^2 = 0 = h - h$; the second one because $0 \le s_x$ appears both in I and in Q and because $s_x \le h$ is guaranteed by $g(h - s_x) \ge 0$, which holds as $\frac{1}{2} s_v^2 \ge 0$. With transitivity and isotonicity of boxes, we can thus bring the correctness specification into the form $I \le |\mathsf{Ball}]I$.

Applying (wlp-star) then yields the proof obligation $I \le |x' = f\ \&\ G\ ;\ \mathsf{Cntrl}]I$. To discharge it, we use (wlp-seq) to calculate the $wlps$

$$J = |\mathbf{if}\ (\lambda s.\ s_x = 0)\ \mathbf{then}\ v := (\lambda s.\ -s_v)\ \mathbf{else}\ skip]I,$$
$$K = |x' = f\ \&\ G]J$$

incrementally and finally show that $I \le K$.

For the first wlp, we calculate, with (wlp-cond) and for $T = (\lambda s.\ s_x = 0)$,

$$J = (T \to |v := (\lambda s.\ -s_v)]I) \cdot (\overline{T} \to I)$$
$$= \left(T \to |v := (\lambda s.\ -s_v)] \left(\lambda s.\ 0 \le s_x \wedge \frac{1}{2} s_v^2 = g(h - s_x) \right) \right) \cdot (\overline{T} \to I)$$
$$= \left(T \to \left(\lambda s.\ 0 \le s_x \wedge \frac{1}{2} (-s_v)^2 = g(h - s_x) \right) \right) \cdot (\overline{T} \to I)$$
$$= (T \to I) \cdot (\overline{T} \to I)$$
$$= I.$$

For the second wlp, we wish to apply (wlp-evl). This requires checking that f is Lipschitz continuous—$\ell = 1$ does the job, supplying a flow and checking that it solves the IVP (f, s) for all $s \in S$ and satisfies the flow conditions for $T = \mathbb{R}$ and $S = \mathbb{R}^V$. We leave it to the reader to verify that $\varphi : \mathbb{R} \to \mathbb{R}^V \to \mathbb{R}^V$ defined by

$$\varphi_s\, t = \begin{pmatrix} s_x \\ s_v \end{pmatrix} + \begin{pmatrix} s_v \\ -g \end{pmatrix} t - \frac{1}{2} \begin{pmatrix} g \\ 0 \end{pmatrix} t^2$$

meets the requirements in the procedure outlined above, cf. Example 8.1. Then, expanding definitions and applying (wlp-evl) from Lemma 9.3,

$K\, s$
$$= \left(\forall t \in \mathbb{R}_+.\ (\forall \tau \in [0, t].\ 0 \le \varphi_s\, \tau\, x) \to 0 \le \varphi_s\, t\, x \wedge \frac{1}{2} (\varphi_s\, t\, v)^2 = g(h - \varphi_s\, t\, x) \right)$$
$$= \left(\forall t.\ (\forall \tau \in [0, t].\ 0 \le \varphi_s\, \tau\, x) \to \frac{1}{2} (\varphi_s\, t\, v)^2 = g(h - \varphi_s\, t\, x) \right)$$
$$= \left(\forall t.\ \left(\forall \tau \in [0, t].\ 0 \le s_x + s_v t - \frac{1}{2} g \tau^2 \right) \right.$$
$$\left. \to \frac{1}{2} (s_v - gt)^2 = g \left(h - s_x - s_v t + \frac{1}{2} g t^2 \right) \right).$$

Finally, for $I \leq K$, suppose $0 \leq s_x$, $\frac{1}{2}s_v^2 = g\left(h - s_x\right)$ and $0 \leq s_x + s_v\tau - \frac{1}{2}g\tau^2$ for all $\tau \in [0, t]$. It remains to show that $\frac{1}{2}\left(s_v - gt\right)^2 = g\left(h - s_x - s_v t + \frac{1}{2}gt^2\right)$. Indeed, using the second assumption in the second step,

$$
\begin{aligned}
\frac{1}{2}(s_v - gt)^2 &= \frac{1}{2}s_v^2 - g\left(s_v t + \frac{1}{2}gt^2\right) \\
&= g(h - s_x) - g\left(s_v t + \frac{1}{2}gt^2\right) \\
&= g\left(h - s_x + s_v t + \frac{1}{2}gt^2\right).
\end{aligned}
$$

The verification with Isabelle described in Example 17.1 is far more automatic than this proof on paper suggests, and there is ample scope for further automation. As already pointed out, the main purpose of this example is to illustrate our first workflow and give an impression of the mathematical reasoning involved.   □

Certifying solutions of systems of ODEs can be tedious and hard to automate, and many ODEs do not admit analytic solutions. It is possible to circumvent these obstacles to practical verification applications in various ways. One approach, using invariant sets for systems of ODEs, is pursued by $d\mathcal{L}$ and described in the following sections. It constitutes the second workflow supported by our framework. Another approach aims at particular types of vector fields for which (global) flows always exist and are easy to compute. A classical example is linear systems of ODEs [22,66], for which the first author has already developed methods in a successor article [26]. A final approach abandons differential equations and vector fields altogether and starts from flows—as known from hybrid automata [10]. This requires changing the syntax of hybrid programs. The approach is outlined in Sect. 18. It constitutes the third workflow supported by our framework.

10 Evolution Commands for Continuous Vector Fields

As the semantic approach to evolution commands developed in the previous section depends mainly on orbits, which are nothing but sets of states, it can be generalised beyond trajectories and flows. In this section, we drop the requirement of uniqueness of solutions to IVPs and hence assume that vector fields are merely continuous. In fact, if vector fields are non-continuous, the set of solutions defined below will simply be empty. We therefore generalise the definitions in the previous section to obtain weakest liberal preconditions for evolution commands that do not admit unique solutions, for instance, IVPs of the form $x' t = k\sqrt{x t}$ with $x\, 0 = 0$ for any $k \in \mathbb{R}$ [24]. Our second workflow using invariant sets is based on this generalisation.

Consider the IVP (f, s) for continuous vector field $f : S \rightarrow S$ and initial state $s \in S \subseteq \mathbb{R}^V$. Let

$$
\mathsf{Sols}\, f\, T\, s = \{X \mid \forall t \in T.\ X' t = f\, (X\, t) \wedge X\, 0 = s\}
$$

denote its set of solutions on $T \subseteq \mathbb{R}$ with $0 \in T$. Here, T is no longer the maximal interval of existence defined by the Picard–Lindelöf theorem; it can be changed like the set U in the previous section. Then, each solution X is still continuously differentiable and thus $f \circ X$ integrable in T.

For all $X \in \mathsf{Sols}\ f\ T\ s$ and $G : S \to \mathbb{B}$, we define the *G-guarded orbit* of X along T in s via the function $\gamma_G^X : S \to \mathcal{P}\ S$ as

$$\gamma_G^X\ s = \bigcup \{\mathcal{P}\ X \downarrow t \mid t \in T \wedge \mathcal{P}\ X \downarrow t \subseteq G\},$$

which simplifies to $\gamma_G^X\ s = \{X\ t \mid t \in T \wedge \forall \tau \in \downarrow t.\ G\ (X\ \tau)\}$. By Kneser's theorem [36], when non-uniqueness occurs at some point, infinitely many solutions exist for it. Thus, we define the *G-guarded orbital* of f along T in s via the function $\gamma_G^f : S \to \mathcal{P}\ S$ as

$$\gamma_G^f\ s = \bigcup \{\gamma_G^X\ s \mid X \in \mathsf{Sols}\ f\ T\ s\}.$$

We thus patch the guarded orbit of each solution to the associated IVP together so that $\gamma_G^f\ s$ represents all possible evolutions in time that pass through s. This is evident from the following result.

Lemma 10.1 *Let $f : S \to S$ be continuous and $G : S \to \mathbb{B}$. Then,*

$$\gamma_G^f\ s = \{X\ t \mid t \in T \wedge \mathcal{P}\ X \downarrow t \subseteq G \wedge X \in \mathsf{Sols}\ f\ T\ s\}.$$

If $G = \top$, the constantly true predicate on S or the set S itself, we simply write γ^f instead of $\gamma_\top^f$.

The state transformer semantics of the evolution command for a continuous vector field f can then be defined as

$$(x' =_{\mathcal{F}}\ f\ \&\ G) = \gamma_G^f.$$

The corresponding relational semantics is

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

Once again, $\langle x' = f\ \&\ G| = (\gamma_G^f)^\dagger$. This leads to a *wlp* for evolution commands.

Proposition 10.2 *Let $S \subseteq \mathbb{R}^V$ and $T \subseteq \mathbb{R}$. Let $f : S \to S$ be a continuous vector field and $G, Q : S \to \mathbb{B}$. Then,*

$$|x' = f\ \&\ G]Q = \lambda s \in S.\ \{s \mid \forall X \in \mathsf{Sols}\ f\ T\ s.\forall t \in T.\ \mathcal{P}\ X \downarrow t \subseteq G \to \mathcal{P}\ X \downarrow t \subseteq Q\}.$$

This identity can be rewritten, for predicates, as

$$|x' = f\ \&\ G]Q = \lambda s \in S.\forall X \in \mathsf{Sols}\ f\ T\ s.\forall t \in T.\ (\forall \tau \in \downarrow t.\ G\ (X\ \tau)) \to Q\ (X\ t).$$

Whether this fact is useful for verification applications, as outlined above, remains to be seen. Yet the next section shows that it is certainly useful for reasoning with invariant sets. The following corollary is important for verification proofs with invariants as well.

Corollary 10.3 *Let $f : S \to S$, $S \subseteq \mathbb{R}^V$, be a continuous vector field, $T \subseteq \mathbb{R}$ and $G, Q : S \to \mathbb{B}$. Then,*

$$|x' = f\ \&\ G]Q = |x' = f\ \&\ G](G \cdot Q).$$

11 Invariants for Evolution Commands

In d$\mathcal{L}$, differential invariants are predicates I that satisfy $I \leq |x' = f \& G]I$ [46]. In the terminology of Sect. 4, they are simply invariants for evolution commands. They play a crucial role in d$\mathcal{L}$ and KeYmaera X because of the limited support for solving ODEs and their greater generality.

In dynamical systems theory, when all guards are $\top$ and global flows exist, and in (semi)group theory, *invariant sets* for actions or flows $\varphi : T \rightarrow S \rightarrow S$ are sets $I \subseteq S$ satisfying $\gamma^\varphi s \subseteq I$ for all $s \in I$ [66]. Based on the results from Sect. 10, we generalise both notions uniformly.

A predicate or set $I : S \rightarrow \mathbb{B}$ is an *invariant* of the continuous vector field $f : S \rightarrow S$ and guard $G : S \rightarrow \mathbb{B}$ *along* $T \subseteq \mathbb{R}$ if

$$(\gamma_G^f)^\dagger I \subseteq I.$$

Note that the parameter T is hidden in the definition of γ_G^f. For $G = \top$, when $(\gamma^f)^\dagger I \subseteq I$, we call I simply an *invariant* of f along T.

The following proposition yields a structural insight in the relationship between invariant sets of dynamical systems and differential invariants of d$\mathcal{L}$ in terms of an adjunction.

Proposition 11.1 *Let* $f : S \rightarrow S$ *be continuous,* $G : S \rightarrow \mathbb{B}$ *and* $T \subseteq \mathbb{R}$. *Then, the following are equivalent.*

1. *I is an invariant for* f *and* G *along* T;
2. $\langle x' = f \& G|I \subseteq I$;
3. $I \subseteq |x' = f \& G]I$.

Proof

$$(\gamma_G^f)^\dagger I \subseteq I \leftrightarrow \langle x' = f \& G|I \subseteq I \leftrightarrow I \subseteq |x' = f \& G]I.$$

The first step uses the definition of backward diamonds as Kleisli extensions in Sect. 6 and that of the semantics of evolution commands in Sect. 10. The final step uses the adjunction between boxes and diamonds from Sect. 4. □

For our *wlp*-calculus, condition (3) is of course most useful. Yet instead of checking that a flow is a solution to a vector field, as previously, we now need to check whether a predicate is an invariant—without having to solve the system of ODEs. This may in some case be a condicio sine qua non and in others a considerable simplification of reasoning. The following lemmas lead to our second workflow. We show some proofs although they have been formalised with Isabelle, as they explain why the approach works.

First, towards Corollary 11.4, we may ignore guards when checking for invariants and we can use a simple second-order formula.

Lemma 11.2 *Let* $f : S \rightarrow S$ *be continuous and* $I : S \rightarrow \mathbb{B}$. *Then,*

1. $I \subseteq |x' = f \& \top]I \rightarrow I \subseteq |x' = f \& G]I$,
2. $I \subseteq |x' = f \& \top]I \leftrightarrow (I\,s \rightarrow \forall X \in \text{Sols}\, f\, T\, s. \forall t \in T.\, I\,(X\,t))$.

Proof For (1), $\gamma_G^f \subseteq \gamma^f$ for all G and hence $\langle x' = f \& G|I \subseteq \langle x' = f \& \top|I \subseteq I$. The proof of (2) is a simple calculation. □

Second, we can recurse over predicates as follows.

Lemma 11.3 *Let $f : S \to S$ be a continuous vector field, $\mu, \nu : S \to \mathbb{R}$ differentiable and $T \subseteq \mathbb{R}$ with $0 \in T$.*

1. *If $(\mu \circ X)' = (\nu \circ X)'$ for all X such that $X' t = f t (X t)$ and $G (X t)$ when $t \in T$, then $\mu = \nu$ is an invariant for f along T,*
2. *if $(\mu \circ X)' \tau \le (\nu \circ X)' \tau$ when $\tau > 0$, and $(\mu \circ X)' \tau \ge (\nu \circ X)' \tau$ when $\tau < 0$, for all X such that $X' t = f t (X t)$ and $G (X t)$, then both $\mu < \nu$ and $\mu \le \nu$ are invariants for f along T,*
3. *if $\mu < \nu$ and $\nu < \mu$ are invariants for f along T, then $\mu \ne \nu$ is too (and conversely if 0 is the least element in T),*
4. *$\mu \not\le \nu$ is an invariant for f along T if and only if $\nu < \mu$ is too.*

Proof We only show the proof of (1), as it reveals the main idea of the procedure outlined below. By definition, $\mu = \nu$ is an invariant for f along T if and only if $\mu s = \nu s$ implies $\mu (X t) = \nu (X t)$ for all $X \in \mathsf{Sols}\, f\, T\, s$. It is a well-known consequence of the mean value theorem that two continuously differentiable functions are the same if and only if they intersect at some point and have the same derivative. Hence, $(\mu \circ X)' = (\nu \circ X)'$ and $\mu s = \nu s$ imply $\mu (X t) = \nu (X t)$ for all $X \in \mathsf{Sols}\, f\, T\, s$.   □

Proposition 10.3, the properties in this section—in particular Lemma 11.3—and Lemma 4.1 about invariants that are conjunctions or disjunctions support our second workflow for proving a correctness specification $P \le |x' = f\, \& \, G]Q$.

1. Check whether a candidate predicate I is a differential invariant:

   (a) transform I into negation normal form;
   (b) if I is complex, reduce it with Lemma 4.1, and Proposition 11.3(3) and (4);
   (c) if I is atomic, apply Proposition 11.3(1) and (2);
       (if successful, $I \le |x' = f\, \& \, G]I$ holds by Proposition 11.1(3) and Lemma 11.2);

2. if successful, prove $P \le I$ and $|x' = f\, \& \, G](G \cdot I) \le |x' = f\, \& \, G]Q$.

   For $G = \top$ and Lipschitz continuous vector fields, the notions of invariant can be strengthened.

Corollary 11.4 *Let $f : S \to S$ be Lipschitz continuous. Then, the following are equivalent.*

1. *I is an invariant for f along T;*
2. *$\langle x' = f\, \& \, \top | I = I$;*
3. *$I = |x' = f\, \& \, \top] I$.*

The identities (2) and (3) hold because $0 \in T$.

Next, we revisit the bouncing ball example from Sect. 9 to illustrate our second work flow that reasons with differential invariants. Once again, we give detailed mathematical calculations to indicate the kind of mathematical reasoning involved. A verification with Isabelle, which is much more automatic, can be found in Example 17.2.

Example 11.5 *(Bouncing ball with differential invariant)* We can avoid solving the system of ODEs in Example 9.4 using a differential invariant to show that

$$I \le |x' = f\, \& \, G]I$$

for the loop invariant I and vector field $f (s_x, s_v)^T = (s_v, -g)^T$. The most natural candidate for a differential invariant is of course energy conservation. Cancelling the mass, we use

$$I_d = \left(\lambda s. \, \frac{1}{2} s_v^2 = g (h - s_x) \right).$$

We now apply our procedure for reasoning with differential invariants.

1. We use Proposition 11.3 with $\mu s = \frac{1}{2}s_v^2$ and $\nu s = g(h - s_x)$ to check that I_d is indeed an invariant. We thus need to show that $(\mu \circ X)' = (\nu \circ X)'$ for all $X \in \mathsf{Sols}\, f\, T\, s$, which unfolds to

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

because $s = X\,t$ and therefore $s_v = X\,t\,v$ and $s_x = X\,t\,x$. And indeed,

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

$$= -g(f\,(X\,t)\,x) = -g(X'\,t\,x) = (g(h - X\,t\,x))'.$$

By Proposition 11.3(1), I_d is thus an invariant for f along $\mathbb{R}^V$. Proposition 11.1(3) and Lemma 11.2 then imply that

$$I_d \le |x' = f\ \&\ G]I_d.$$

2. It remains to show that $I \le I_d$ and $|x' = f\ \&\ G]I_d \le |x' = f\ \&\ G]I$.

   – The first inequality is trivial.
   – For the second one, we calculate

$$(G \cdot I_d)\,s = \left(0 \le s_x \wedge \frac{1}{2}s_v^2 = g(h - s_x)\right) = I\,s.$$

   By Corollary 10.3, therefore,

$$|x' = f\ \&\ G]I_d = |x' = f\ \&\ G](G \cdot I_d) = |x' = f\ \&\ G]I.$$

This shows that $I \le |x' = f\ \&\ G]I$. The remaining proof of $P \le |\mathsf{Ball}]Q$ is the same as in Example 9.4. □

This example shows that one can reason about invariants of evolution commands in a natural mathematical style as it can be found in textbooks on differential equations [4,22, 66]. By contrast, $d\mathcal{L}$ relies on syntactic substitution-based reasoning in the term algebra of differential rings [46] to check invariants, and complex domain-specific inference rules to manipulate them. The following section shows that we can derive semantic variants of most of the $d\mathcal{L}$ inference rules for those who like this style of reasoning, see [12] for a complete list.

Next, we briefly specialise our approach to $d\mathcal{L}$-invariants, the invariants sets used in dynamical systems theory and those in (semi)group theory. We assume a setting where global flows exist and indices U can be dropped.

Corollary 11.6 *Let $f : S \to S$ be Lipschitz continuous. Then, $I : S \to \mathbb{B}$ is a $d\mathcal{L}$-invariant for $x' = f\ \&\ \top$ if and only if I is an invariant set for φ^f.*

Proof It is easy to check that $(\forall s \in I.\ I\,s \to \gamma^\varphi s \subseteq I) \leftrightarrow (\gamma^\varphi)^\dagger I \subseteq I$. The claim then follows from Proposition 11.1. In the Lipschitz continuous case, of course, $\mathsf{Sols}\, f\, T\, s = \{\varphi^f\}$. □

It remains to point out that the difference between the definition of invariant sets for dynamical systems and that for (semi)group actions is merely notational: In group theory,

an invariant set I of a (semi)group action $\varphi : T \rightarrow S \rightarrow S$ satisfies $T \cdot I \subseteq I$, where $T \cdot I = \{\varphi\, t\, s \mid t \in T \wedge s \in I\}$. In the presence of a unit, therefore $T \cdot I = I$. Yet of course $(\gamma^{\varphi})^{\dagger} I = \{\varphi\, t\, s \mid t \in T \wedge s \in I\}$ as well.

At the end of this section, we summarise the two main workflows presented. Both use the standard laws for predicate transformer algebras for automating verification condition generation with respect to the structural part of hybrid programs. For straight-line programs, this requires only equational reasoning and can be dealt with by Isabelle's simplifiers. The remaining verification conditions for basic commands—evolution and assignment commands—are generated by equational reasoning in the concrete semantics of the hybrid program store. In fact, only this concrete semantics had to be added to a standard Isabelle verification component to make our verification components work.

The verification conditions generated are then at the level of reasoning with functions over $\mathbb{R}^n$, and in some cases in linear algebra [26]. At this level, by contrast with dℒ, we do not require any domain-specific inference rules and can rely on Isabelle's support for semantic reason about the hybrid dynamics within its higher-order logic, an approach that has allowed us to verify a large number of benchmark examples [45]. Yet our approach is versatile enough to derive inference rules in the style of dℒ, as the following section shows.

12 Derivation of dℒ Inference Rules

As a proof of concept, we derive semantic variants of some axioms and inference rules of dℒ, thus proving their soundness with respect to our semantics. The first one introduces solutions of IVPs with constant vector fields [6]. It is a trivial instance of Proposition 9.2 with $f = \lambda s.\, c$ for some $c \in \mathbb{R}$. Such vector fields are Lipschitz continuous; their flows are $\varphi\, t\, s = s + ct$. Hence,

$$|x' = (\lambda s.\, c)\, \&\, G]Q = \lambda s \in S.\, \forall t \in T.\, (\forall \tau \leq t.\, G\, (s + c\tau)) \rightarrow Q\, (s + ct). \qquad \text{(DS)}$$

For a second dℒ inference rule, we simply rewrite the *wlp* in Proposition 9.2 as a Hoare-style inference rule.

Lemma 12.1 *Let $S \subseteq \mathbb{R}^V$ and $T = \mathbb{R}$. Let $\varphi : T \rightarrow S \rightarrow S$ be the flow for the Lipschitz continuous vector field $f : S \rightarrow S$, and $G, Q : S \rightarrow \mathbb{B}$. Then,*

$$\frac{\forall s \in S.\, P\, s \rightarrow (\forall t \in T.\, (\forall \tau \leq t.\, G(\varphi_s\, \tau)) \rightarrow Q\, (\varphi_s\, t))}{P \leq |x' = f\, \&\, G]Q} \qquad \text{(dSolve)}$$

To apply this rule in our setting, the procedure in Sect. 9 must be followed.

Next, we derive five semantic counterparts of the dℒ axioms and inference rules for differential invariants in the setting of Sect. 11. The *differential cut* axiom (DC) and rule (dC), *differential weakening*, (DW) and (dW), and the *differential induction* rule (dI). These rules are typically applied backwards as follows: *dC* introduces an invariant. Its left premise is discharged via *dI*, Proposition 11.1 and logical reasoning, while its right premise is discharged via *dW*. Note that the conclusions of all these rules are semantically equivalent to Hoare triples. Verification examples using these rules and the dℒ approach can be found in our Isabelle components.

Lemma 12.2 *Let $P, G, I, Q : S \rightarrow \mathbb{B}$, $T \subseteq \mathbb{R}$ and $f : S \rightarrow S$ be a continuous vector field. Then, with η_S the unit of the powerset monad,*

$$|x' = f\, \&\, G]I = \eta_S \rightarrow |x' = f\, \&\, (\lambda s.\, G\, s \wedge I\, s)]Q = |x' = f\, \&\, G]Q, \qquad \text{(DC)}$$

$$\frac{P \le |x' = f \mathbin{\&} G]I \qquad P \le |x' = f \mathbin{\&} (\lambda s.\, G\,s \wedge I\,s)]Q}{P \le |x' = f \mathbin{\&} G]Q} \tag{dC}$$

$$|x' = f \mathbin{\&} G](\lambda s.\, G\,s \to Q\,s) = |x' = f \mathbin{\&} G]Q, \tag{DW}$$

$$\frac{G \le Q}{P \le |x' = f \mathbin{\&} G]Q} \tag{dW}$$

Finally, if I is a differential invariant for f along T, then

$$\frac{P \le I \qquad I \le Q}{P \le |x' = f \mathbin{\&} G]Q} \tag{dI}$$

Axiom (DC) and rule (dC) introduce differential invariants in guards of evolution commands. Axiom and rule (DW) and (dW) summarise the fact that if a guard is strong enough to imply a postcondition, then no invariant or solution needs to be found. Finally, the differential induction rule follows from Proposition 11.1(3), transitivity and isotonicity of boxes.

A differential ghost rule [51] (dG) and sometimes a differential effect axiom [48] have also been proposed for reasoning with invariants in d$\mathcal{L}$. Our semantics approach has so far no need for these [45]—we do not anticipate any reason why we should not be able to freely introduce ghost variables for the continuous dynamics as we have so far done for the discrete one using Isabelle's higher-order logic—but see [12] for a derivation of (dG) within our semantic framework.

13 Isabelle Components for MKA and Predicate Transformers

The entire mathematical development of MKA in Sects. 2–4 has been formalised with Isabelle [3,16]. Verification components for Isabelle and the relational store model in Sect. 7 have been developed, too [17,18], using the shallow embedding approach discussed in Sects. 1 and 7. Predicate transformers à la Back and von Wright have been formalised previously in Isabelle by Preoteasa [52,53]. Our alternative formalisation emphasises the quantalic structure of transformers [64,65], as in Sect. 5, and we have added a third component based on quantaloids [65]. It is based on a formalisation of the powerset monad [65], as outlined in Sect. 6. Our formalisation is compositional in that all three approaches to predicate transformers can be combined with relational and state transformer semantics and different models of the (hybrid) program store, as shown in Fig. 1.

This section summarises the Isabelle components for predicate transformers and the verification component based on MKA. More detailed information can be found in the proof documents for these components [18,65].

The MKA component is integrated into the Kleene algebra hierarchy that formalises variants of Kleene algebras [3] and modal Kleene algebras [16], as outlined in Sects. 2 and 3. In these mathematical components, algebras are formalised as type classes, their models via instantiation and interpretation statements. For Kleene algebras, many computationally interesting models have been formalised; for MKA, only the relational model is present in the Archive of Formal Proofs. The state transformer model has been formalised for quantales in a different component [65].

Instantiation and interpretation statements have several purposes in Isabelle. They make algebraic facts available in all models, establish soundness of algebraic hierarchies and ultimately make the axiomatic approaches consistent with respect to Isabelle's small trustworthy core. Finally, they unify developments of multiple concrete semantics.

In our MKA-based verification components [18], program syntax is absent and semantic illusions of program syntax are provided in the concrete program semantics, as outlined in Sect. 7. Consequently, verification conditions for the control structure of programs are generated within the algebra; those for assignments in the concrete store semantics. We currently model stores simply as functions from strings representing variables to values of arbitrary type. Expressions are simulated by functions from stores to values, as outlined in Sect. 7; stores with poly-typed values are modelled via sum-types. An extension to verification components for hybrid programs is described in the following sections.

A second component is based on predicate transformers à la Back and von Wright [5], for which we have built special purpose components with advanced features for orderings and lattices [63] and for quantales [64]. These structures are once again formalised as type classes. Predicate transformers, however, are modelled as global functions that may have different source and target types. Isabelle's simple type system can infer most general types for definitions. These can be associated with predicate transformers by sort constraints; definitions can often be declared in the point-free style of functional programming. This makes the formalisation of quantaloids of transformers with partial compositions straightforward. Mono-typed transformer algebras are obtained from these via subtyping. They are linked with quantales and Kleene algebras by interpretation or instantiation.

Isabelle's type system is too weak for a deep embedding of general categorical concepts, but formalising instances such as the powerset monad, its Kleisli category and Eilenberg–Moore algebras is straightforward. We have formalised the isomorphisms and dualities between relations, state transformers and the four predicate transformers corresponding to backward and forward boxes and diamonds in this setting. Using these dualities to transport theorems automatically requires Isabelle's transfer package, which is ongoing work.

We have created a second verification component for hybrid systems based on Back and von Wright's approach, using the monadic transformers to obtain a concrete semantics. Finally, we have once again restricted the categorical approach to the mono-typed case in a third component. Via subtyping, we can then show that the categorical transformers form quantales, and more specifically MKAs. Everything Isabelle knows about MKA is then available in this instance.

14 Isabelle Components for ODEs and Orbits

This section and the two following ones describe the formalisation of the material in Sects. 8–12 in Isabelle, from mathematical components for ODEs and orbits to verification components for hybrid programs based on (local) flows, differential invariants and d$\mathcal{L}$-style inference rules.

We begin with summarising Immler and Hölzl's formalisation of the Picard–Lindelöf theorem based on the Isabelle hierarchy for analysis and ordinary differential equations [23, 29–31]. We have adapted their results to show that unique solutions to IVPs for autonomous systems of ODEs guaranteed by this theorem satisfy the local flow conditions, as discussed in previous sections.

Hölzl and Immler have proved the Picard–Lindelöf theorem for time-dependent vector fields of type $real \Rightarrow ('a::\{heine\text{-}borel, banach\}) \Rightarrow 'a$ [30]. They have called their theorem *unique-solution* and have formalised it within a locale called *unique-on-bounded-closed* to bundle the assumptions for the local existence of unique solutions within a closed interval in $\mathbb{R}$. They have specialised and hence extended this locale in various ways.

Our approach builds on top of their extension *ll-on-open-it* that bundles more or less the conditions of Theorem 8.2, but for the time-dependent case. In our formalisation, we add the

condition $t_0 \in T$ to have this parameter available in the following developments. Thus, we have generated the following variant.

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

sublocale *ll-on-open-it T f S* t_0
  $\langle proof \rangle$

lemma *unique-solution*:
  assumes *xivp*: $D\, X = (\lambda t. f\, t\, (X\, t))$ *on* $\{t_0 - -t\}\, X\, t_0 = s\, X \in \{t_0 - -t\} \to S$
    and $t \in T$
    and *yivp*: $D\, Y = (\lambda t. f\, t\, (Y\, t))$ *on* $\{t_0 - -t\}\, Y\, t_0 = s\, Y \in \{t_0 - -t\} \to S$
    and $s \in S$
  shows $X\, t = Y\, t$
  $\langle proof \rangle$

end

The locale declaration lists the assumptions of the Picard–Lindelöf theorem: the vector field f—which is still time-dependent—is defined on an open time interval T that contains the initial time t_0, and an open subset S of the state space. The vector field f is continuous in time and, for each $(t, s) \in T \times S$, Lipschitz continuous on a closed subset of $T \times S$ around (t, s). The sublocale statement shows that these assumptions imply those of the locale *ll-on-open-it*. Lemma *unique-solution* ensures that the Picard–Lindelöf theorem is derivable within this locale. The notation $D\, X$ stands for X', and $g \in A \to B$ indicates that function g maps from the set A into the set B, as opposed to the type of g, which can be larger. The notation $\{t_0 - -t\}$ indicates the set of real numbers between t_0 and t (including both), where t may be above or below t_0. The formalisation of the Picard–Lindelöf theorem comprises a formal definition of solutions to IVPs of system of ODEs in Isabelle. As an abbreviation, we have defined the set $\mathsf{Sols}\, f\, T\, s$ of Sect. 10 with the additional requirement that $X \in T \to S$.

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

We restrict locale *picard-lindeloef* to autonomous systems and to $t_0 = 0$, while introducing the variable φ for the local flow of the vector field. In support of our open approach to hybrid program verification, this allows users to supply any characterisation of the flow that suits them best, as a successor paper illustrates [26].

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

The assumptions *ivp* force *T* to coincide with its largest subinterval (*ex-ivl*) where solutions exist (lemma *ex-ivl-eq* below). Thus, φ is the unique solution on the whole of *T* —and not only on its subsets $\{0--t\}$ unlike *picard-lindeloef* or *ll-on-open-it*. This allows users of the locale to choose *T* as small as they wish.

lemma *ex-ivl-eq*: $s \in S \Longrightarrow ex\text{-}ivl\, s = T$
⟨*proof*⟩

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

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

lemma *eq-solution*: $X \in Sols\,(\lambda t.\,f)\,T\,S\,0\,s \Longrightarrow t \in T \Longrightarrow s \in S \Longrightarrow X\,t = \varphi\,t\,s$
⟨*proof*⟩

Finally, in this locale we can prove that if the maximal interval of existence *T* equals $\mathbb{R}$, then the flow φ is global and hence a proper monoid action.

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

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

We have not generated a locale for this case, as the assumptions needed to remain unchanged. Locale *picard-lindeloef* thus guarantees the existence of unique solutions for IVPs of time-dependent systems. Locale *local-flow* specialises it to autonomous systems with Lipschitz continuous vector fields and local flows. It covers dynamical systems with global flows and thus the verification of hybrid systems. This provides the basic Isabelle infrastructure for formalising the concrete semantics for hybrid systems with Lipschitz continuous vector fields from Fig. 1.

Next, we describe our formalisation of the orbits and orbitals from Sect. 10. These form the basis for our verification components for continuous vector fields beyond the scope of the Picard–Lindelöf theorem, as shown in Fig. 1. Yet we can instantiate all concepts to settings where (local) flows exist. First, we have formalised the *G*-guarded orbit γ_G^X of *X* along *T*, with *down T t* standing for $\downarrow t$.

definition *g-orbit* :: $(real \Rightarrow {}'a) \Rightarrow ({}'a \Rightarrow bool) \Rightarrow real\ set \Rightarrow {}'a\ set$ (γ)
  where $\gamma\,X\,G\,T = \bigcup\{\mathcal{P}\,X\,(down\,T\,t)\,|t.\,\mathcal{P}\,X\,(down\,T\,t) \subseteq \{s.\,G\,s\}\}$

lemma *g-orbit-eq*: $\gamma\,X\,G\,T = \{X\,t\,|t.\,t \in T \wedge (\forall\,\tau{\in}down\,T\,t.\,G\,(X\,\tau))\}$
⟨*proof*⟩

We have also formalised the *G*-guarded orbital of *f* along *T* in *s* (as $\gamma_G^f\,s$) together with Lemma 10.1.

definition *g-orbital* :: $({}'a \Rightarrow {}'a) \Rightarrow ({}'a \Rightarrow bool) \Rightarrow real\ set \Rightarrow real \Rightarrow$
  $({}'a::real\text{-}normed\text{-}vector) \Rightarrow {}'a\ set$
  where $g\text{-}orbital\,f\,G\,T\,S\,t_0\,s = \bigcup\{\gamma\,X\,G\,T\,|X.\,X \in Sols\,(\lambda t.\,f)\,T\,S\,t_0\,s\}$

lemma *g-orbital-eq*: *g-orbital f G T S t_0 s* =
  {*X t* |*t X*. *t* ∈ *T* ∧ $\mathcal{P}$ *X* (*down T t*) ⊆ {*s*. *G s*} ∧ *X* ∈ *Sols* (λ*t*. *f*) *T S t_0 s* }
  ⟨*proof*⟩

We have shown that their counterparts from dynamical systems are special cases by instanti-ating our definitions to the parameters of the locale *local-flow*. Hence, the ⊤-guarded orbital of *f* along *T* in *s* becomes the standard orbit of *s*, and its *G*-guarded version is the set in Lemma 9.1.

context *local-flow*
begin

definition *orbit* :: $'a \Rightarrow {}'a$ *set* (γ^φ)
  where γ^φ *s* = *g-orbital f* (λ*s*. *True*) *T S 0 s*

lemma *orbit-eq*[*simp*]: *s* ∈ *S* ⟹ γ^φ *s* = {φ *t s* |*t*. *t* ∈ *T*}
  ⟨*proof*⟩

lemma *g-orbital-collapses*:
  s ∈ *S* ⟹ *g-orbital f G T S 0 s* = {φ *t s* |*t*. *t* ∈ *T* ∧ (∀ τ∈*down T t*. *G* (φ τ *s*))}
  ⟨*proof*⟩

end

Overall, the set-theoretic concepts introduced in Sect. 10 are easily definable in Isabelle. Similarly, lemmas formalising their properties and relating them are often proved automati-cally in one or two lines. Analytical properties like the existence of derivatives in a region of space or the uniqueness of solutions for IVPs are harder to prove. Such lemmas often require long structured proofs with proofs by cases and explicit calculations, that is, a considerable amount of user interaction. Yet most proofs remain at least roughly at the level of textbook reasoning.

15 Isabelle Components for Hybrid Programs

This section describes the integration of the state transformer and relational semantics for dynamical systems and Lipschitz-continuous vector fields from Sect. 9 and the continuous vector fields from Sect. 10 into the three verification components for predicate transformers outlined in Sect. 13 and Fig. 1. This requires formalising hybrid stores and the semantics of evolution commands for dynamical systems, Lipschitz continuous vector fields with local flows and continuous vector fields. As explained in Sect. 9 and 11, this supports two different workflows using the procedures introduced in these sections: the first one is for reasoning with (local) flows and orbits, the second, more general one, for reasoning with invariants.

First, we explain our formalisation of the hybrid store type $\mathbb{R}^V$. We use Isabelle's type (*real*, $'n$) *vec* (abbreviated as *real^$'n$*) of real valued vectors of dimension *n*, formalised as the type $'n \Rightarrow real$ of functions from the finite type $'n$ into $\mathbb{R}$. This represents hybrid stores in $\mathbb{R}^V$ with $|V| = n$. Isabelle uses the notation *s$i* for the *i*th coordinate of a vector *s* and hence the value of store *s* at variable *i*. More mathematically, $ is the bijection from *real^$'n$* to $'n \Rightarrow real$. Its inverse is written using a binder χ that replaces λ-abstraction. Thus, (χ*i*. *s*)$*i* = *s* for any *s*::*real^$'n$* and (χ*i*. *x*)$*i* = *x* for any *x*::*real*. As a consequence of this simple approach, variables are formalised as natural numbers. More general namespaces have been included in our framework more recently [11,12] to make it more user friendly.

Our state transformer semantics uses functions of type $real\hat{}'n \Rightarrow (real\hat{}'n)$ *set*, which we abbreviate as $(real\hat{}'n)$ *nd-fun* (for non-deterministic functions). These are instances of the more general type $'a$ *nd-fun* of nondeterministic endofunctions.

Alternatively, we use relations of type $(real\hat{}'n)$ *rel*, which are instances of $'a$ *rel*. For both intermediate semantics, we have shown with Isabelle that they form MKAs, but we have also integrated them into the two quantalic predicate transformer semantics in Fig. 1.

interpretation *rel-aka*: *antidomain-kleene-algebra Id* {} (∪) (;) (⊆) (⊂) *rtrancl rel-ad*
⟨*proof*⟩

instantiation *nd-fun* :: (*type*) *antidomain-kleene-algebra*
⟨*proof*⟩

After these proofs, all statements proved in Isabelle's MKA components are available for state transformers and relations. We have formalised *wlp*s for both models, where $\lceil - \rceil$ ambiguously denotes the isomorphism between predicates and binary relations or nondeterministic functions.

lemma *wp-rel*: *wp R* $\lceil P \rceil = \lceil \lambda x. \forall y. (x,y) \in R \longrightarrow P y \rceil$
⟨*proof*⟩

lemma *wp-nd-fun*: *wp F* $\lceil P \rceil = \lceil \lambda x. \forall y. y \in (F x) \longrightarrow P y \rceil$
⟨*proof*⟩

Alternatively, we use the categorical forward box operator $fb_{\mathcal{F}}$ for Kleisli arrows of type $F :: 'a \Rightarrow 'b$ *set* described in Sect. 6,

lemma *ffb-eq*: $fb_{\mathcal{F}} F X = \{x. \forall y. y \in F x \longrightarrow y \in X\}$
⟨*proof*⟩

or its relational counterpart $fb_{\mathcal{R}}$.

We now switch to the categorical approach to predicate transformers based on state transformers and the Kleisli monad of the powerset functor, as a preliminary MKA-based one with relations has already been described elsewhere [28]. Apart from typing and some minor syntactic differences, the other approaches—predicate transformers based on MKA and quantales, and an intermediate relational semantics for these—yield analogous results and are equally suitable for verification. This evidences the compositionality of our approach.

The state and predicate transformer semantics of assignment commands is based on store update functions, as described in Sect. 7. For hybrid programs, it must be adapted to type $'a\hat{}'n$.

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

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

lemma *ffb-assign*[*simp*]: $fb_{\mathcal{F}}$ (*x* ::= *e*) *Q* = {*s*. (χ *j*. ((($) *s*)(*x* := (*e s*))) *j*) ∈ *Q*}
⟨*proof*⟩

The ($) applies the bijection $ as a function in prefix notation.

We write (*x* ::= *e*) for the semantic illusion for a syntactic assignment commands, as Isabelle uses *f* (*i* := *a*) for function update *f* [*i* ↦ *a*]. Lemma *ffb-assign* is then a direct consequence of *ffb-eq*, and it coincides with (wlp-asgn) in Sect. 7 up to minor syntactic

differences. In the verification examples that feature in this article, we have not attempted to hide the functions that impersonate syntactic expressions and the lambda abstractions they require. This may be unwieldy for users. It is nevertheless routine to program more elegant notation with Isabelle [12].

Similarly, *wlp*s for the control structure commands of hybrid programs (Eqs. wlp-seq, wlp-cond, wlp-star) are easily derivable.

lemma *ffb-kcomp*[*simp*]: $fb_{\mathcal{F}}\ (G\ ;\ F)\ P = fb_{\mathcal{F}}\ G\ (fb_{\mathcal{F}}\ F\ P)$
⟨*proof*⟩

lemma *ffb-if-then-else*[*simp*]: $fb_{\mathcal{F}}\ (IF\ T\ THEN\ X\ ELSE\ Y)\ Q =$
$\{s.\ T\ s \longrightarrow s \in fb_{\mathcal{F}}\ X\ Q\} \cap \{s.\ \neg\ T\ s \longrightarrow s \in fb_{\mathcal{F}}\ Y\ Q\}$
⟨*proof*⟩

lemma *ffb-loopI*: $P \leq \{s.\ I\ s\} \implies \{s.\ I\ s\} \leq Q \implies \{s.\ I\ s\} \leq fb_{\mathcal{F}}\ F\ \{s.\ I\ s\} \implies$
$P \leq fb_{\mathcal{F}}\ (LOOP\ F\ INV\ I)\ Q$
⟨*proof*⟩

In these lemmas, ; is syntactic sugar for the forward Kleisli composition $\circ_K$ and *LOOP* stands for the Kleene star for state transformers with its annotated loop-invariant after the keyword *INV*, along the lines of Sect. 4.

As in Sect. 10, the general semantics of evolution commands for continuous vector fields is given by G-guarded orbitals of f along T. We have formalised the *wlp*s in Proposition 10.2, and a specialisation to local flows in the context of our locale *local-flow* given by Lemma 9.3 [Eq. (wlp-evl)].

notation *g-orbital* $((1x' =\!\!-\ \&\ \text{-}\ on\ \text{-}\ \text{-}\ @\ \text{-}))$

lemma *ffb-g-orbital*:
$fb_{\mathcal{F}}\ (x' = f\ \&\ G\ on\ T\ S\ @\ t_0)\ Q =$
$\{s.\ \forall X \in Sols\ (\lambda t.\ f)\ T\ S\ t_0\ s.\ \forall t \in T.\ (\forall \tau \in down\ T\ t.\ G\ (X\ \tau)) \longrightarrow (X\ t) \in Q\}$
⟨*proof*⟩

lemma (**in** *local-flow*) *ffb-g-ode*:
$fb_{\mathcal{F}}\ (x' = f\ \&\ G\ on\ T\ S\ @\ 0)\ Q =$
$\{s.\ s \in S \longrightarrow (\forall t \in T.\ (\forall \tau \in down\ T\ t.\ G\ (\varphi\ \tau\ s)) \longrightarrow (\varphi\ t\ s) \in Q)\}$
⟨*proof*⟩

As Lemma *ffb-g-ode* is defined in locale *local-flow*, users are required to check the conditions of the Picard–Lindelöf theorem to access this locale and certify that φ is indeed a solution of the IVP as part of our first workflow.

Finally, we describe our component for reasoning with differential invariants in the general setting of continuous vector fields, using our second workflow. We start with definitions and a basic property from Proposition 11.1.

definition *diff-invariant* :: $('a \Rightarrow bool) \Rightarrow (('a::real\text{-}normed\text{-}vector) \Rightarrow 'a) \Rightarrow real\ set \Rightarrow$
$'a\ set \Rightarrow real \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where *diff-invariant* $I\ f\ T\ S\ t_0\ G = (((g\text{-}orbital\ f\ G\ T\ S\ t_0)^{\dagger})\ \{s.\ I\ s\} \subseteq \{s.\ I\ s\})$

lemma *ffb-diff-inv*:
diff-invariant $I\ f\ T\ S\ t_0\ G = (\{s.\ I\ s\} \leq fb_{\mathcal{F}}\ (x' = f\ \&\ G\ on\ T\ S\ @\ t_0)\ \{s.\ I\ s\})$
⟨*proof*⟩

We have formalised the most important rules for reasoning with differential invariants, including those for the procedure of Sect. 11 via Corollary 10.3 and Lemmas 4.1 and 11.3.

The formalisation of the first two is straightforward. We have proved the clauses of 11.3 in various lemmas and bundled them under the name *diff-invariant-rules*. We show one of these clauses as an example.

named-theorems *diff-invariant-rules compilation of rules for differential invariants.*

lemma [*diff-invariant-rules*]:
  assumes *is-interval T*
  and $t_0 \in T$
  and $\forall X. (D X = (\lambda\tau. f (X \tau)) \text{ on } T) \longrightarrow (D (\lambda\tau. \mu (X \tau) - \nu (X \tau)) = ((*_R) 0) \text{ on } T)$
  shows *diff-invariant* $(\lambda s. \mu\ s = \nu\ s) f T S t_0 G$
  $\langle proof \rangle$

lemma *ffb-g-odei*: $P \leq \{s. I s\} \Longrightarrow \{s. I s\} \leq fb_{\mathcal{F}} (x'{=}f \& G \text{ on } T S @ t_0) \{s. I s\} \Longrightarrow$
  $\{s. I s \land G s\} \leq Q \Longrightarrow P \leq fb_{\mathcal{F}} (x'{=}f \& G \text{ on } T S @ t_0 \text{ DINV } I) Q$
  $\langle proof \rangle$

Lemma *ffb-g-odei* completes the procedure of Sect. 11 by formalising step 2, which annotates invariants in evolution commands, following the approach outlined for loops and general commands in MKA at the end of Sect. 4. With Isabelle, we use the *DINV* keyword.

The two workflows for proving partial correctness specifications with evolution commands require users to discharge proof obligations for derivatives. In the case of flows, these must be solutions for vector fields; in the case of differential invariants, the procedure of Sect. 11 requires proving the assumptions of Lemma 11.3. To increase proof automation when reasoning about derivatives, we have bundled several derivative properties under the name *poly-derivatives* as a proof method.

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

declare *has-vderiv-on-const* [*poly-derivatives*]
    and *has-vderiv-on-id* [*poly-derivatives*]
    and *has-vderiv-on-add* [*THEN has-vderiv-on-eq-rhs, poly-derivatives*]
    and *has-vderiv-on-diff* [*THEN has-vderiv-on-eq-rhs, poly-derivatives*]
    and *has-vderiv-on-mult* [*THEN has-vderiv-on-eq-rhs, poly-derivatives*]

lemma [*poly-derivatives*]: $D f = f' \text{ on } T \Longrightarrow g = (\lambda t. - f' t) \Longrightarrow D (\lambda t. - f t) = g \text{ on } T$
  $\langle proof \rangle$

lemma [*poly-derivatives*]: $(a{::}real) \neq 0 \Longrightarrow D f = f' \text{ on } T \Longrightarrow g = (\lambda t. (f' t)/a) \Longrightarrow$
  $D (\lambda t. (f t)/a) = g \text{ on } T$
  $\langle proof \rangle$

lemma [*poly-derivatives*]: $n \geq 1 \Longrightarrow D (f{::}real \Rightarrow real) = f' \text{ on } T \Longrightarrow$
  $g = (\lambda t. n * (f' t) * (f t)\hat{\ }(n-1)) \Longrightarrow D (\lambda t. (f t)\hat{\ }n) = g \text{ on } T$
  $\langle proof \rangle$

lemma [*poly-derivatives*]: $D (f{::}real \Rightarrow real) = f' \text{ on } T \Longrightarrow$
  $g = (\lambda t. - (f' t) * sin (f t)) \Longrightarrow D (\lambda t. cos (f t)) = g \text{ on } T$
  $\langle proof \rangle$

lemma [*poly-derivatives*]: $D (f{::}real \Rightarrow real) = f' \text{ on } T \Longrightarrow g = (\lambda t. (f' t) * cos (f t)) \Longrightarrow$
  $D (\lambda t. sin (f t)) = g \text{ on } T$
  $\langle proof \rangle$

lemma [*poly-derivatives*]: $D (f{::}real \Rightarrow real) = f' \text{ on } T \Longrightarrow g = (\lambda t. (f' t) * exp (f t)) \Longrightarrow$
  $D (\lambda t. exp (f t)) = g \text{ on } T$
  $\langle proof \rangle$

Isabelle can now apply rules iteratively and check, for pairs of functions, if one is a derivative of the other. This is often fully automatic. The following lemma shows an example that involves a mix of polynomials and transcendental functions beyond differential fields with a_0 to a_5 being constants and t the polynomial variable.

lemma $c \neq 0 \Longrightarrow D \ (\lambda t. \ a_5 * t\hat{\ }5 + a_3 * (t\hat{\ }3 \ / \ c) - a_2 * exp \ (t\hat{\ }2) + a_1 * cos \ t + a_0)$
$= (\lambda t. \ 5 * a_5 * t\hat{\ }4 + 3 * a_3 * (t\hat{\ }2 \ / \ c) - 2 * a_2 * t * exp \ (t\hat{\ }2) - a_1 * sin \ t) \ on \ T$
   by(*auto intro*!: *poly-derivatives*)

The formalisation of more advanced heuristics for such functions, and the integration of decision procedures for suitable classes, is left for future work.

The complete Isabelle formalisation, including the other two predicate transformer algebras and the relational semantics, can be found in the Archive of Formal Proofs [25].

We briefly reflect on our experience with the Isabelle formalisation of our framework. MKA, its relational model and the concrete relational semantics for traditional while programs are so far the most developed and versatile starting point for our hybrid systems verification components. The full formalisation of a rudimentary Hoare logic component for this setting using a generalised Kleene algebra from Isabelle's main libraries fits on two A4 pages [62]; a similar development for a Hoare logic for hybrid programs is discussed in a successor paper [11]. Our standalone MKA-based verification component for traditional while programs fills about seven A4 pages. For hybrid programs, in theory, only a concrete semantics for hybrid programs needs to be plugged in as a replacement of the semantics described in Sect. 7. In practice, however, Isabelle's instantiations often make theory hierarchies non-compositional as each type can only be instantiated in one way. We faced such a clash of instances between Isabelle's Kleene algebra and analysis hierarchies and hence had to customise the former for our purposes.

Replacing the intermediate relational semantics by state transformers required some background work, simply because the former are well supported by Isabelle, whereas the latter are new. Interestingly, it is possible to propagate theorems automatically along the isomorphisms between these semantics like for type classes, locales and their instantiations and interpretation. Isabelle's transfer and lifting packages provide an infrastructure for this, which remains by and large unexplored. We leave this for future work.

The categorical approach to transformer quantaloids is more complex—both conceptually and from a formalisation point of view—than the MKA based one, in particular when state transformers are integrated via the powerset monad. At the level of verification conditions generation, however, there are almost no differences. Once again a stripped down component can be generated that just suffices for verification condition generation, and we are using it in subsequent work [12]. Relative to Isabelle's main libraries, it fills merely four pages [25]. Working with quantales instead of quantaloids might seem mathematically simpler, but with Isabelle it is actually more tedious, as subtypes for endofunctions need to be created.

In sum, for simple verification tasks, the lightweight stripped down predicate transformer algebras obtained from MKA or quantaloids seem preferable; for more complex program transformations or refinements, the integration into the full MKA hierarchy or categorical predicate transformer component is certainly beneficial.

16 Isabelle Support for d$\mathcal{L}$-Style Reasoning

This section lists our formalisation of semantic variants of the most important axioms and inference rules of d$\mathcal{L}$ in Isabelle outlined in Sect. 12. It covers all three predicate transformer

semantics as well as the relational and state transformer model. Once again, we only show state transformers in the categorical approach.

We have formalised a generalised version of the d$\mathcal{L}$-rules with parameters T, S and t_0 with intervals U and for orbitals. We can easily instantiate them to $\mathbb{R}$, $\mathbb{R}^V$ and 0, respectively. This enables users to perform verification proofs in the style of d$\mathcal{L}$ and establishes soundness of these rules relative to our semantics as a side effect. First, we show our formalisations of (DS) and (dSolve).

lemma *DS*:
  fixes *c*::$'a$::{*heine-borel*, *banach*}
  shows $fb_\mathcal{F}$ ($x' = (\lambda s.\ c)$ & G) $Q = \{x.\ \forall t.\ (\forall \tau \leq t.\ G\ (x + \tau *_R c)) \longrightarrow (x + t *_R c) \in Q\}$
  ⟨*proof*⟩

lemma *solve*:
  assumes *local-flow f UNIV UNIV* φ
    and $\forall s.\ s \in P \longrightarrow (\forall t.\ (\forall \tau \leq t.\ G\ (\varphi\ \tau\ s)) \longrightarrow (\varphi\ t\ s) \in Q)$
  shows $P \leq fb_\mathcal{F}$ ($x' = f$ & G) Q
  ⟨*proof*⟩

Next, we list semantic variants of the five d$\mathcal{L}$ axioms and inference rules for reasoning with differential invariants discussed in Sect. 12. Recall that due to our semantic approach, evolution commands in these rules only require the vector field f :: $'a \Rightarrow 'a$ and guard G :: $'a \Rightarrow bool$, while the $x' =$ is just syntactic sugar to resemble ODEs.

lemma *DW*: $fb_\mathcal{F}$ ($x' = f$ & G) $Q = fb_\mathcal{F}$ ($x' = f$ & G) $\{s.\ G\ s \longrightarrow s \in Q\}$
  ⟨*proof*⟩

lemma *dW*: $\{s.\ G\ s\} \leq Q \Longrightarrow P \leq fb_\mathcal{F}$ ($x' = f$ & G) Q
  ⟨*proof*⟩

lemma *DC*:
  assumes $fb_\mathcal{F}$ ($x' = f$ & G) $\{s.\ C\ s\} = UNIV$
  shows $fb_\mathcal{F}$ ($x' = f$ & G) $Q = fb_\mathcal{F}$ ($x' = f$ & ($\lambda s.\ G\ s \wedge C\ s$)) Q
  ⟨*proof*⟩

lemma *dC*:
  assumes $P \leq fb_\mathcal{F}$ ($x' = f$ & G) $\{s.\ C\ s\}$
    and $P \leq fb_\mathcal{F}$ ($x' = f$ & ($\lambda s.\ G\ s \wedge C\ s$)) Q
  shows $P \leq fb_\mathcal{F}$ ($x' = f$ & G) Q
  ⟨*proof*⟩

lemma *dI*:
  assumes $P \leq \{s.\ I\ s\}$
    and *diff-invariant I f UNIV UNIV 0 G*
    and $\{s.\ I\ s\} \leq Q$
  shows $P \leq fb_\mathcal{F}$ ($x' = f$ & G) Q
  ⟨*proof*⟩

Additional d$\mathcal{L}$ rules can easily be formalised. More recent work features, for instance, a ghost rule [12], which is heavily used for reasoning with invariants in d$\mathcal{L}$, but seems less relevant to our semantic approach [45].

17 Verification Examples

This section explains the formalisation of the bouncing ball examples from Sect. 9 and 11 with Isabelle, and we add two further verification examples using a simple circular pendulum. All four of them use Isabelle's type *2* of two elements. It denotes the set of variables *V* of hybrid programs over the state space $\mathbb{R}^V$ for $|V| = 2$. We follow Isabelle's notation and write *0::2* and *1::2* for the two variables and their type. As such a formalisation of variables is rather unwieldy, more recent extensions to our framework support more general name spaces, more sophisticated store models and a more user-friendly specification language for hybrid programs and assertions [12]. The examples in this section should therefore be taken cum grano salis.

Example 17.1 *(Bouncing Ball via Flow)* First, we formalise Example 9.4 with our verification components for flows, using our first workflow. We write *0::2* for the ball's position starting from height *h*, *1::2* for its velocity, and *s$0* and *s$1* for s_x and s_v. We formalise the vector field $f\ (s_x, s_v)^T = (s_v, -g)^T$ for the ball as

abbreviation *fball :: real* $\Rightarrow$ *real^2* $\Rightarrow$ *real^2* (*f*)
  where *f g s* $\equiv$ (χ *i. if i=0 then s$1 else g*)

We can now state the partial correctness specification for the bouncing ball in Isabelle, where the loop invariant *I* is that of Sect. 9, but written slightly differently to enhance proof automation.

lemma *bouncing-ball*: *g < 0* $\Longrightarrow$ *h* $\geq$ *0* $\Longrightarrow$
  {*s. s$0 = h* $\wedge$ *s$1 = 0*} $\leq$ *fb*$_\mathcal{F}$
  (*LOOP* (
    (*x´=(f g)* & (λ *s. s$0* $\geq$ *0*)) ;
    (*IF* (λ *s. s$0 = 0*) *THEN* (*1 ::= ($\lambda$s. − s$1*)) *ELSE skip*))
  INV (λ*s. 0* $\leq$ *s$0* $\wedge$ *2 · g · s$0 − 2 · g · h − s$1 · s$1 = 0*))
  {*s. 0* $\leq$ *s$0* $\wedge$ *s$0* $\leq$ *h*}

The proof of this lemma is shown below. It follows that in Example 9.4, but requires some intermediate lemmas. For example, if we first apply rule *ffb-loopI* (wlp-star), the subgoals $P \leq I$ and $I \leq Q$, for $P = (\lambda s.\ s_x = h \wedge s_v = 0)$ and $Q = (\lambda s.\ 0 \leq s_x \leq h)$, need to be proven. They can be discharged automatically after supplying some lemmas about real arithmetic, which have been bundled under the name *bb-real-arith*. We show one of them below to give an impression.

named-theorems *bb-real-arith real arithmetic properties for the bouncing ball*.

lemma [*bb-real-arith*]: *0 > g* $\Longrightarrow$ *2 · g · x − 2 · g · h = v ·* $\Longrightarrow$ (*x::real*) $\leq$ *h*
  $\langle proof \rangle$

These properties depend on distributivity and commutativity properties that Isabelle cannot simplify immediately. As we are not working within a well-defined language, such as differential rings or fields, we have not attempted to automate them any further, so that proofs require some user interaction.

The remaining rules, that is, *ffb-kcomp* (wlp-seq), *ffb-if-then-else* (wlp-cond), and *ffb-assign* (wlp-asgn), have been added to Isabelle's automatic proof tools. It then remains to compute the *wlp* for the evolution command of the bouncing ball. To use *local-flow.ffb-g-ode* (wlp-evl), we follow the procedure in Sect. 9. We need to check that the vector field is Lip-

schitz continuous, supply the local flow as in Example 9.4 and check that it solves the IVP and satisfies the flow conditions.

abbreviation *ball-flow* :: *real* ⇒ *real* ⇒ *real^2* ⇒ *real^2* (φ)
  where φ *g t s* ≡ (χ *i*. *if i=0 then g · t ˆ2/2 + s$1 · t + s$0 else g · t + s$1*)

lemma *local-flow-ball*: *local-flow* (*f g*) *UNIV UNIV* (φ *g*)
  ⟨*proof*⟩

The arithmetic computations with real numbers at the end of Example 9.4 are then discharged automatically by adding the rules in *bb-real-arith* to Isabelle's automatic tools. The resulting two-line proof of the bouncing ball is shown below.

apply(*rule wp-loopI*, *simp-all add*: *local-flow.wp-g-ode*[*OF local-flow-ball*])
  by (*auto simp*: *bb-real-arith*)

Overall, the verification proof covers less than a page and a half in the proof document— and this is mainly due to the few arithmetic calculations in the background that require user interaction. All other proofs make heavy use of Isabelle's simplifiers and are by and large automatic.   □

Example 17.2 *(Bouncing Ball via Invariant)* This example formalises the invariant-based proof from Example 11.5 using our second workflow. The correctness specification changes in that we annotate the differential invariant ab initio.

lemma *bouncing-ball-invariants*: *g < 0* ⟹ *h ≥ 0* ⟹
  {*s. s$0 = h ∧ s$1 = 0*} ≤ *fb*$_\mathcal{F}$
  (*LOOP* (
    (*x´=(f g)*) & (λ *s. s$0 ≥ 0*) *DINV* (λ *s. 2 · g · s$0 − 2 · g · h − s$1 · s$1 = 0*)) ;
    (*IF* (λ *s. s$0 = 0*) *THEN* (*1* ::= (λ*s. − s$1*)) *ELSE skip*))
  INV (λ*s. 0 ≤ s$0* ∧*2 · g · s$0 − 2 · g · h − s$1 · s$1 = 0*))
  {*s. 0 ≤ s$0 ∧ s$0 ≤ h*}
  apply(*rule ffb-loopI*, *simp-all*)
    apply(*force*, *force simp*: *bb-real-arith*)
  by(*rule ffb-g-odei*) (*auto intro*!: *diff-invariant-rules poly-derivatives*)

As before, the first line of the proof applies the non-evolution *wlp*-rules; the second one discharges *P ≤ I* and *I ≤ Q* for loop invariant *I*. It remains to show that *I ≤ |x′ = f & G* DINV *I_d*]*I* for differential invariant *I_d*.

For this, we unfold the annotated invariant rule *ffb-g-odei*, which performs step (2) of Example 11.5 and generates the proof obligation *I_d ≤ |x′ = f & G]I_d*. The proof of this fact is automatic because the rule *ffb-diff-inv* (Lemma 11.2) has been added to Isabelle's simplifiers. Step (1) is checked with our rules for derivatives *poly-derivatives* and differential invariants *diff-invariant-rules* (Proposition 11.3). The full verification covers less than a page in the proof document.   □

Example 17.3 *(Circular Pendulum via Invariant)* The ODEs

$$x' t = y t \qquad \text{and} \qquad y' t = -x t,$$

which correspond to the vector field $f : \mathbb{R}^V \to \mathbb{R}^V$,

$$f \begin{pmatrix} s_x \\ s_y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} s_x \\ s_y \end{pmatrix},$$

for $V = \{x, y\}$, describe the kinematics of a circular pendulum. All orbits are "governed" by the separable differential equation

$$\frac{dy}{dx} = \frac{y'}{x'} = -\frac{x}{y},$$

obtained by parametric derivation. Rewriting it as $x\, dx + y\, dy = 0$ and integrating both sides yield $x^2 + y^2 = r^2$, for some constant $r > 0$, which describes the circular orbits of the ODEs. This leads to the differential invariant

$$I = \left(\lambda s.\ s_x^2 + s_y^2 = r^2\right), \qquad (r \geq 0).$$

Once again, we apply our procedure from Sect. 11 to prove

$$I = |x' = f \,\&\, \top] I$$

using Lemma 11.4, as the guard is trivial.

1. Using Proposition 11.3 with $\mu\, s = s_x^2$ and $\nu\, s = r^2 - s_y^2$, we check that I is an invariant, showing that $(\mu \circ X)' = (\nu \circ X)'$ for all $X \in \mathsf{Sols}\, f\, T\, s$, and hence

$$\left((X\, t\, x)^2\right)' = \left(r^2 - (X\, t\, y)^2\right)'.$$

We calculate

$$\left((X\, t\, x)^2\right)' = 2(X\, t\, x)(X'\, t\, x) = -2(X'\, t\, y)(X\, t\, y) = \left(r^2 - (X\, t\, y)^2\right)'.$$

It therefore follows from Proposition 11.3(1) that I is an invariant for f along $\mathbb{R}^V$; $I = |x' = f \,\&\, \top] I$ holds by Lemma 11.4.
2. As $P = I = Q$, there is nothing to show.

In the Isabelle formalisation, we introduce a name for the vector field and show that I is an invariant for it—as the invariant is the pre- and postcondition, an annotation is not needed. The verification is straightforward following the workflow of the previous example, and even simpler because the pre- and postconditions are just the differential invariant.

abbreviation *fpend* :: *real^2 ⇒ real^2* (*f*)
 where *f s ≡ (χ i. if i=0 then s$1 else −s$0)*

lemma *pendulum*: {*s. r² = (s$0)² + (s$1)²*} ≤ *fb_F (x´= f & G)* {*s. r² = (s$0)² + (s$1)²*}
 by (*auto intro!: diff-invariant-rules poly-derivatives*)

The Isabelle proof is automatic if we supply the tactic for derivative rules. □

Example 17.4 (*Circular Pendulum via Flow*) Alternatively, the kinematic equations for the circular pendulum from Example 17.4 can of course be solved using linear combinations of trigonometric functions. Yet first we need to show that the vector field f is Lipschitz continuous with constant 1. Next, we supply the flow

$$\varphi_s\, t = \begin{pmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{pmatrix} \begin{pmatrix} s_x \\ s_y \end{pmatrix}.$$

We need to check that it solves the IVP (f, s) for all $s \in \mathbb{R}^V$ and that it satisfies the flow conditions for $T = \mathbb{R}$ and $S = \mathbb{R}^V$. As an example calculation,

$$\varphi_s'\, t = \begin{pmatrix} -\sin t & \cos t \\ -\cos t & -\sin t \end{pmatrix} \begin{pmatrix} s_x \\ s_y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{pmatrix} \begin{pmatrix} s_x \\ s_y \end{pmatrix} = f\, (\varphi_s\, t).$$

The remaining conditions are left to the reader.

To compute $|x' = f \,\&\, \top]I$, we expand (wlp-evl). This yields

$$
\begin{aligned}
|x' = f \,\&\, \top]I \, s &= \forall t. \, I \, (\varphi_s \, t) \\
&= \left(\forall t. \, (\varphi_s \, t \, x)^2 + (\varphi_s \, t \, y)^2 = r^2 \right) \\
&= \left(\forall t. \, (s_x \cos t + s_y \sin t)^2 + (s_y \cos t - s_x \sin t)^2 = r^2 \right) \\
&= \left(\forall t. \, s_x^2 (\sin^2 t + \cos^2 t) + s_y^2 (\sin^2 t + \cos^2 t) = r^2 \right) \\
&= I \, s.
\end{aligned}
$$

In the Isabelle proof along these lines, we first prove that the vector field satisfies the conditions of the Picard–Lindelöf theorem. To this end, we need to unfold the locale definitions, then introduce the Lipschitz constant and call Isabelle's simplifiers. Next, to prove that the solution supplied is a flow and a solution to the IVP, we unfold definitions and finish the proof by checking that the derivative of the flow in each coordinate coincides with the vector field in that coordinate. The introduction of the flow and these lemmas are shown below.

abbreviation *pend-flow* :: *real* $\Rightarrow$ *real^2* $\Rightarrow$ *real^2* (φ)
 where $\varphi \, t \, s \equiv (\chi \, i.$ *if i = 0 then s$0 · cos t + s$1 · sin t else s$1 · cos t − s$0 · sin t*)

lemma *local-flow-pend*: *local-flow f UNIV UNIV* φ
 $\langle proof \rangle$

The proof of the correctness specification requires only an application of the *wlp* rule *local-flow.ffb-g-ode* (wlp-evl) and Isabelle's simplifier.

lemma *pendulum*: $\{s. \, r^2 = (s\$0)^2 + (s\$1)^2\} \le fb_{\mathcal{F}} \, (x'=f \,\&\, G) \, \{s. \, r^2 = (s\$0)^2 + (s\$1)^2\}$
 by (*force simp*: *local-flow.ffb-g-ode*[*OF local-flow-pend*])

□

All four examples have been based on the categorical approach and the state transformer semantics. Alternative formalisations for the other predicate transformer algebras and the relational semantics can be found in other verification components [25]. In the MKA-based component, the proofs using the relational and the state transformer semantics are precisely the same, which underpins the modularity of our approach. In the other components, we could certainly achieve the same effect by simply rewriting names and adjusting some types.

Transcendental functions cannot be expressed directly in d$\mathcal{L}$'s term language, yet we can use them smoothly and easily with Isabelle with the tactic outlined in Sect. 15. Both the differential invariant workflow and the flow-based workflow benefit from these rules. In fact, both approaches are very similar for the pendulum example: both need a handful of lemmas to prove the partial correctness specification $I = |x' = f \,\&\, \top]I$, and both require a creative step in the form of introducing a differential invariant or the flow for the system.

We have presented the pendulum example in matrix notation as this points to a common feature of many applications: their dynamics can be described by linear systems of ODEs that are representable by matrices and have uniform solutions given by a matrix exponential that can be computed with standard methods from linear algebra. The development of domain-specific techniques for linear systems with Isabelle has been the subject of a successor article [26]. Beyond these simple examples, our approach has successfully tackled a large set of benchmarks from a systems competition [45] and been fine-tuned for proof automation, so that the size of proofs and level of user interaction reported in this article is no longer representative. More information about the background theory development with Isabelle and

the methods and heuristics programmed can be found in the first author's doctoral dissertation [27]. A more far-reaching integration of solvers and decision procedures, or procedures for invariant learning, as oracles or with correctness guarantees, is of course crucial to the applicability of this framework, but beyond the semantic considerations of this article. It is left for future work.

18 Outlook: A Flow-Based Verification Component

The verification components presented so far adhered very much to the pessimistic interactive theorem proving mindset that prefers the internal reconstruction of all external results. This section briefly outlines a fourth, more optimistic verification component that deviates entirely from the vector-field-based approach of $d\mathcal{L}$ and works directly with flows or solutions to IVPs. It shifts responsibility for the correctness of solutions entirely to users—or the computer algebra system they could or should use. This is common practice for instance when working with hybrid automata [10], and of course it simplifies proofs considerably.

For this third workflow supported by our framework, the topological or differentiable structure of the underlying state space is of secondary interest. With Isabelle, this kind of structure and additional conditions can always be imposed by instantiating types with sort constraints as they arise. Hence, we start from a setting that covers both discrete and continuous evolutions and use a general type for time instead of *real*, *rat* or *int*. The evolution commands now specify arbitrary guarded φ-type functions instead of vector fields. The type of time needs to admit an order relation, which is indicated by the sort constraint *ord* below, yet specific properties, such as reflexivity or transitivity, need not be imposed ab initio.

Apart from that, the definition of the guarded-orbit semantics and the *wlp* rule are as before, but side conditions on Lipschitz continuity or the Picard–Lindelöf theorem are superfluous.

definition *g-evol* :: $((′a::ord) \Rightarrow ′b \Rightarrow ′b) \Rightarrow ′b\ pred \Rightarrow ′a\ set \Rightarrow (′b \Rightarrow ′b\ set)$ *(EVOL)*
  where *EVOL* φ *G T* = $(\lambda s.\ g\text{-}orbit\ (\lambda t.\ \varphi\ t\ s)\ G\ T)$

lemma *fbox-g-evol*[*simp*]:
  fixes φ :: $(′a::preorder) \Rightarrow ′b \Rightarrow ′b$
  shows $fb_\mathcal{F}$ *(EVOL φ G T)* $Q = \{s.\ (\forall t{\in}T.\ (\forall \tau{\in}down\ T\ t.\ G\ (\varphi\ \tau\ s)) \longrightarrow (\varphi\ t\ s) \in Q)\}$
  unfolding *g-evol-def g-orbit-eq ffb-eq* **by** *auto*

Using the flows of the bouncing ball and the circular pendulum from previous examples, verification proofs are now fully automatic.

lemma *pendulum-dyn*:
  $\{s.\ r^2 = (s\$0)^2 + (s\$1)^2\} \leq fb_\mathcal{F}$ *(EVOL φ G T)* $\{s.\ r^2 = (s\$0)^2 + (s\$1)^2\}$
  by *force*

lemma *bouncing-ball-dyn*: $g < 0 \Longrightarrow h \geq 0 \Longrightarrow$
  $\{s.\ s\$0 = h \land s\$1 = 0\} \leq fb_\mathcal{F}$
  (LOOP (
    (EVOL (φ g) (λ s. s$0 $\geq$ 0) T) ;
    (IF (λ s. s$0 = 0) THEN (1 ::= ($\lambda s.\ - s\$1$)) ELSE skip))
  INV ($\lambda s.\ 0 \leq s\$0 \land 2 \cdot g \cdot s\$0 - 2 \cdot g \cdot h - s\$1 \cdot s\$1 = 0$))
  $\{s.\ 0 \leq s\$0 \land s\$0 \leq h\}$
  by *(rule ffb-loopI) (auto simp: bb-real-arith)*

These examples no longer link flows with initial specifications in terms of system of ODEs, from which a user might have started. Hence, there is no longer any formal guarantee from Isabelle that the function φ specified satisfies any continuity of differentiability assumptions such as those of *local-flow*.

Further elaboration of this approach, in particular towards discrete systems or in the direction of hybrid automata, is left for future work.

19 Related Work

Methods for automated verification condition generation for partial and total correctness assertions with proof assistants date back to the early days of hardware and software verification by Gordon and colleagues [19]. Discussions on the benefits of shallow embeddings of verification methods in proof assistants—among them faster developments and increased modularity—can be traced back to the same group of researchers. We generally follow an approach described in [2] that starts from algebras of programs to generate verification conditions for the structural commands of programs, while developing those for basic commands in concrete semantics of the program store dynamics.

Mathematical components for classical real analysis have been developed for the Coq proof assistant in the Coquelicot library [7], others for constructive analysis in the CoRN library [8]. The Picard–Lindelöf theorem seems to be available only in the latter [42]. The proof assistant HOL-light includes a library with formalised n-dimensional Euclidean spaces. The first formalisation of the Picard–Lindelöf theorem in Isabelle, which we rewrite for our purposes and specialise to local flows, can be found in the AFP entry for ordinary differential equations [30].

Hybrid systems verification in general-purpose proof assistants has also been investigated. Examples in PVS include semantic invariant reasoning with hybrid automata [1] and, after submission of this article and publication of its precursor [28], $\mathsf{d}\mathcal{L}$-style verification with semi-algebraic sets and real analytic functions [59]. An earlier formalisation of the control function of an inverted pendulum [57] uses the Coquelicot library. Also in Coq, the robot operating system (ROSCoq) framework uses a shallowly embedded logic of events to reason about hybrid systems but only with $\mathsf{d}\mathcal{L}$'s differential induction rule. The HHL prover [67] formalises a Hoare-logic for hybrid systems verification within its calculus of hybrid communicating sequential processes in Isabelle [38]; part of their approach has been deeply embedded. Their semantics is very different from ours. An integration of their LZZ method [39] for finding semi-algebraic invariants for polynomial dynamical systems could probably be integrated into our framework to increase proof automation. Finally, a term-checker for KeYmaera X [6] and, after submission of this article, a formalisation of differential game logic ($\mathsf{d}\mathcal{G}\mathcal{L}$) [50] have been deeply embedded recently in Isabelle/HOL. None of them aim at hybrid program verification.

For an in-depth description of $\mathsf{d}\mathcal{L}$ see [49], a thorough study of differential invariants has been pursued in [46].

In theory, our own framework should therefore allow the integration of much of the related work mentioned, so long as it is consistent with our hybrid store semantics. It is not even necessary to delegate every task to the proof assistant. One can use external tools implementing decision procedures as oracles or at least certify their outputs with Isabelle. The oracle-based approach, however, may jeopardise the desirable conservative extension property relative to Isabelle's own kernel. Translations between different proof assistants may

not always be straightforward. For instance, it is yet to be seen if dependent types or multi-parameter type classes are needed for more flexible implementations of functions spaces (bounded, linear and continuous) or complex vector spaces, or if alternative formalisations of Picard–Lindelöf theorem and other existence theorems might help us to alleviate some of the requirements in our workflows.

20 Conclusion

We have presented a new semantic framework for the deductive verification of hybrid systems with the Isabelle/HOL proof assistant. The approach is inspired by differential dynamic logic, but the design of our verification components, the focus of our framework and the workflows for verifying hybrid systems are different.

First of all, as we use a shallow embedding, the basic verification components generated are quite minimalist and conceptually simple. They merely require the integration of a *wlp* semantics for basic evolution commands of hybrid programs into standard predicate transformer algebras. Our preferred semantics for such commands are state transformers, which in most cases simply map states to the guarded orbits of their temporal evolutions. Beyond that, no domain-specific inference rules are needed, verification condition generation is fully automatic, and even our approach to differential invariants is based entirely on general purpose algebraic invariance laws. Our examples show that mathematical reasoning about differential equations follows standard textbook style and hence comes close to the way mathematicians, physicists or control engineers have been trained to reason about such systems. Whether this is preferable to the proof-theoretic approach advocated by KeYmaera X remains to be seen.

Secondly, our approach aims at an open experimental platform that is only limited by Isabelle's ODE and analysis components, the expressivity of its higher-order logic and type system and the proof methods it provides. We could, for instance, have developed our semantics for time-dependent vector fields, but the restriction to autonomous systems, which does not affect generality, seems preferable in practice. The integration of internal or external solvers for differential algebras, transcendental functions or computer algebra systems for computing Lipschitz constants or flows in the style of Isabelle's Sledgehammer tactic is certainly interesting and a very important avenue for future work, but not a main concern in this article. So far, our open approach simply offers semantic alternatives that users may explore, adapt and extend.

Two specialisations of our framework are the topic of successor papers. The first one restricts our approach to linear systems of differential equations, where exponential solutions exist and can be computed with standard methods from linear algebra [26]. The second one [11] specialises the predicate transformer semantics to algebraic variants of Hoare logics and to refinement calculi for hybrid programs along the lines of previous components for traditional while programs [2]. This shows that our denotational semantics for hybrid programs are compatible with any Hoare logic, which constitutes another significant conceptual simplification relative to d$\mathcal{L}$ and KeYmaera X.

Beyond that we expect that a recent formalisation of Poincaré maps with Isabelle [31] might allow us to extend our framework to discrete dynamical systems and more computational approaches to hybrid systems.

Moreover, differential-algebraic systems of equations [20], which mix differential equations and algebraic equations, and partial differential equations [34] are important for many applications in control engineering and physics. Extending our approach is likely to require

significant background work on mathematical components with Isabelle. While, in both settings, some simple cases can be reduced to systems of ODEs, numerical methods are usually needed for working with such systems. Whether the workflow of mathematicians, physicists and engineers with such more computational approaches can be approximated easily with Isabelle remains to be seen.

Finally, much work is needed to transform our framework into an applicable verification tool for hybrid systems. First steps have meanwhile been taken [12] with respect to more refined hybrid stores and a more user-friendly specification language for hybrid programs and their correctness properties, as already mentioned. More important, however, seems the integration of external solvers and decision procedures, to which much work in the hybrid systems community has already been devoted [13,39,54,58,60]. Such procedures already increase the proof automation of KeYmaera X, and we foresee no reason why similar integrations should not lead to similar benefits within our own framework.

A Cross-References to Isabelle Lemmas

Result in article	Formalisation in Isabelle [25]
Proposition 2.1	Implied by **interpretation** *rel-aka* in Sect. 15
Proposition 2.2	Implied by **instantiation** *nd-fun* in Sect. 15
Proposition 3.1	**interpretation** *rel-aka* in Sect. 15
Proposition 3.2	**instantiation** *nd-fun* in Sect. 15
wlp for sequential composition (wlp-seq)	*ffb-kcomp* in Sect. 15
wlp rule for if-then-else (wlp-cond)	*ffb-if-then-else* in Sect. 15
wlp rule for finite iterations (wlp-while)	In the proof of *ffb-loopI* in Sect. 15
Lemma 4.1	Part of **named-theorems** *diff-invariant-rules* of Sect. 15
Update functions from Sect. 7	**definition** *vec-upd* in Sect. 15
Semantics for assignments from Sect. 7	**definition** *assign* in Sect. 15
wlp for assignments from Sects. 7 (wlp-asgn)	*ffb-assign* in Sect. 15
Orbits in Sect. 8	**definition** *orbit* in Sect. 15
Picard–Lindelöf Theorem 8.2	*picard-lindeloef-closed-ivl.unique-solution* in Sect. 14
Monoid action identities for flows from Sect. 8	*local-flow.is-monoid-action* in Sect. 14

Result in article	Formalisation in Isabelle [25]
G-guarded orbit (γ_G^φ) in Sect. 9	instance of **definition** *g-orbit* of Sect. 14
Lemma 9.1	*g-orbital-collapses* in Sect. 14
Semantics for evolution commands in Sect. 9	**notation** *g-orbital* of Sect. 15
Proposition 9.2	implied by *ffb-g-ode* in Sect. 15
Lemma 9.3 (wlp-evl)	*ffb-g-ode* in Sect. 15
Example 9.4	Example 17.1
Sols $f\, t_0\, s$ in Sect. 10	**definition** *ivp-sols* in Sect. 14
G-guarded orbit γ_G^X of X along T in Sect. 10	**definition** *g-orbit* of Sect. 14
G-guarded orbital γ_G^f of f along T in Sect. 10	**definition** *g-orbital* of Sect. 14
Lemma 10.1	*ffb-g-orbital-eq* in Sect. 14
Semantics for evolution commands in Sect. 10	**notation** *g-orbital* of Sect. 15
Proposition 10.2	*ffb-g-orbital* of Sect. 15
Invariant of IVP (f, s)	**definition** *diff-invariant* in Sect. 15
Proposition 11.1	*ffb-diff-inv* in Sect. 15
Proposition 11.3	**named-theorems** *diff-invariant-rules* of Sect. 15
Example 11.5	Example 17.2
d$\mathcal{L}$-axiom (DS) in Sect. 12	*DS* in Sect. 16
Proposition 9.2 (d$\mathcal{L}$-rule (dSolve))	*solve* in Sect. 16
Lemma 12.2 (DC)	*DC* in Sect. 16
Lemma 12.2 (dC)	*dC* in Sect. 16
Lemma 12.2 (DW)	*DW* in Sect. 16
Lemma 12.2 (dW)	*dW* in Sect. 16
Lemma 12.2 (dI)	*dI* in Sect. Sect. 16

References

1. Ábrahám-Mumm, E., Steffen, M., Hannemann, U.: Verification of hybrid systems: Formalization and proof rules in PVS. In: ICECCS 2001, pp. 48–57. IEEE Computer Society (2001)
2. Armstrong, A., Gomes, V.B.F., Struth, G.: Building program construction and verification tools from algebraic principles. Formal Aspects Comput. **28**(2), 265–293 (2016)
3. Armstrong, A., Struth, G., Weber, T.: Kleene algebra. Archive of Formal Proofs (2013). https://isa-afp.org/entries/Kleene_Algebra.html
4. Arnol'd, V.I.: Ordinary Differential Equations. Springer, New York (1992)
5. Back, R., von Wright, J.: Refinement Calculus—A Systematic Introduction. Springer, New York (1998)
6. Bohrer, B., Rahli, V., Vukotic, I., Völp, M., Platzer, A.: Formally verified differential dynamic logic. In: CPP 2017, pp. 208–221. ACM (2017)
7. Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: a user-friendly library of real analysis for Coq. MCS **9**(1), 41–62 (2015)
8. Cruz-Filipe, L., Geuvers, H., Wiedijk, F.: C-corn, the constructive Coq repository at Nijmegen. In: MKM 2004, volume 3119 of LNCS, pp. 88–103. Springer (2004)
9. Desharnais, J., Struth, G.: Internal axioms for domain semirings. Sci. Comput. Program. **76**(3), 181–203 (2011)
10. Doyen, L., Frehse, G., Pappas, G.J., Platzer, A.: Verification of hybrid systems. In: Handbook of Model Checking., pp. 1047–1110. Springer (2018)
11. Foster, S., Huerta y Munive, J.J., Struth, G.: Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In: RAMiCS 2020, pp. 169–186 (2020)
12. Foster, S., y Munive, J.J.H., Gleirscher, M., Struth, G.: Hybrid systems verification with isabelle/HOL: simpler syntax, better models, faster proofs. *CoRR*, abs/2106.05987 (2021)
13. Fulton, N., Mitsch, S., Bohrer, B., Platzer, A.: Bellerophon: Tactical theorem proving for hybrid systems. In: ITP 2017, LNCS, pp. 207–224. Springer (2017)
14. Fulton, N., Mitsch, S., Quesel, J., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: CADE-25, volume 9195 of *LNCS*, pp. 527–538. Springer (2015)
15. Gierz, G., Hofmann, K.H., Lawson, J.D., Mislove, M., Scott, D.S.: A Compendium of Continuous Lattices. Springer, New York (1980)

16. Gomes, V.B.F., Guttmann, W., Höfner, P., Struth, G., Weber, T.: Kleene algebras with domain. Archive of Formal Proofs (2016). https://isa-afp.org/entries/KAD.html
17. Gomes, V.B.F., Struth, G.: Modal Kleene algebra applied to program correctness. In: FM 2016, volume 9995 of LNCS, pp. 310–325 (2016)
18. Gomes, V.B.F., Struth, G.: Program construction and verification components based on Kleene algebra. In: Archive of Formal Proofs (2016)
19. Gordon, M.J.C.: Mechanizing Programming Logics in Higher Order Logic, pp. 387–439. Springer, New York (1989)
20. Hairer, E., Wanner, G.: Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems. Springer, New York (1996)
21. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
22. Hirsch, M.W., Smale, S., Devaney, R.L.: Differential Equations, Dynamical Systems, and Linear Algebra. Academic Press, London (1974)
23. Hölzl, J., Immler, F., Huffman, B.: Type classes and filters for mathematical analysis in Isabelle/HOL. In: ITP 2013, volume 7998 of LNCS, pp. 279–294. Springer (2013)
24. Hubbard, J.H., West, B.H.: Differential Equations: A Dynamical Systems Approach. Springer, New York (1991)
25. Huerta y Munive, J.J.: Verification components for hybrid systems. Archive of Formal Proofs (2019). https://isa-afp.org/entries/Hybrid_Systems_VCs.html
26. Huerta y Munive, J.J.: Affine systems of ODEs in Isabelle/HOL for hybrid-program verification. In: SEFM 2020, volume 12310 of LNCS, pp. 77–92. Springer (2020)
27. Huerta y Munive, J.J.: *Algebraic verification of hybrid systems in Isabelle/HOL.* PhD thesis, University of Sheffield (2021)
28. Huerta y Munive, J.J., Struth, G.: Verifying hybrid systems with modal Kleene algebra. In: Desharnais, J., Guttmann, W., Joosten, S. (eds.) Relational and Algebraic Methods in Computer Science, pp. 225–243. Springer (2018)
29. Immler, F., Hölzl, J.: Numerical analysis of ordinary differential equations in Isabelle/HOL. In: ITP 2012, volume 7406 of LNCS, pp. 377–392. Springer (2012)
30. Immler, F., Hölzl, J.: Ordinary differential equations. Archive of Formal Proofs (2012). https://isa-afp.org/entries/Ordinary_Differential_Equations.html
31. Immler, F., Traut, C.: The flow of ODEs: formalization of variational equation and Poincaré map. J. Autom. Reason. **62**(2), 215–236 (2019)
32. Jacobs, B.: A recipe for state-and-effect triangles. In: Logical Methods in Computer Science, vol. 13, no. 2 (2017)
33. Jeannin, J., Ghorbal, K., Kouskoulas, Y., Schmidt, A., Gardner, R., Mitsch, S., Platzer, A.: A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. STTT **19**(6), 717–741 (2017)
34. John, F.: Partial Differential Equations. Springer, New York (1986)
35. Jónsson, B., Tarski, A.: Boolean algebras with operators, Part I. Americal. J. Math. **73**(4), 207–215 (1951)
36. Kneser, H.: Über die Lösungen eines Systems gewöhnlicher Differentialgleichungen, das der Lipschitzschen Bedingung nicht genügt. *Sitz.ber. Preuß*, pp. 58–61 (1923)
37. Kozen, D.: Kleene algebra with tests. ACM TOPLAS **19**(3), 427–443 (1997)
38. Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., Zou, L.: A calculus for hybrid CSP. In: APLAS 2010, volume 6461 of LNCS, pp. 1–15. Springer (2010)
39. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: Chakraborty, S., Jerraya, A., Baruah, S.K., Fischmeister, S. (eds) EMSOFT 2011, pp. 97–106. ACM (2011)
40. Loos, S.M., Platzer, A., Nistor, L.: Adaptive cruise control: Hybrid, distributed, and now formally verified. In: FM 2011, volume 6664 of LNCS, pp. 42–56. Springer (2011)
41. MacLane, S.: Categories for the Working Mathematician. Springer, New York (1971)
42. Makarov, E., Spitters, B.: The Picard algorithm for ordinary differential equations in Coq. In: ITP 2013, volume 7998 of LNCS, pp. 463–468. Springer (2013)
43. Manes, E.G.: Predicate Transformer Semantics. Cambridge University Press, Cambridge (1992)
44. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Functional Programming Languages and Computer Architecture 1991, volume 523 of LNCS. Springer (1991)
45. Mitsch, S., Huerta y Munive, J.J., Jin, X., Zhan, B., Wang, S., Zhan, N.: ARCH-COMP20 category report: Hybrid systems theorem proving. In: ARCH20, pp. 141–161 (2019)
46. Platzer, A.: The structure of differential invariants and differential cut elimination. In: LMCS, vol. 8, no. 4 (2008)

47. Platzer, A.: Logical Analysis of Hybrid Systems. Springer, New York (2010)
48. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. J. Autom. Reason. **59**(2), 219–265 (2017)
49. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer, New York (2018)
50. Platzer, A.: Differential game logic. Archive of Formal Proofs (2019). https://isa-afp.org/entries/Differential_Game_Logic.html
51. Platzer, A., Tan, Y.K.: Differential equation axiomatization: the impressive power of differential ghosts. In: LICS, pp. 819–828. ACM (2018)
52. Preoteasa, V.: Algebra of monotonic Boolean transformers. Archive of Formal Proofs (2011). https://isa-afp.org/entries/MonoBoolTranAlgebra.html
53. Preoteasa, V.: Algebra of monotonic boolean transformers. In: Archive of Formal Proofs (2011)
54. Rebiha, R., Moura, A.V., Matringe, N.: Generating invariants for non-linear hybrid systems. Theor. Comput. Sci. **594**, 180–200 (2015)
55. Rosenthal, K.I.: The Theory of Quantaloids. Chapman and Hall/CRC, London (1996)
56. Rosenthal, K.L.: Quantales and Their Applications. Longman Scientific & Technical, Essex (1990)
57. Rouhling, D.: A formal proof in Coq of a control function for the inverted pendulum. In: CPP 2018, pp. 28–41. ACM (2018)
58. Sassi, M. A. B., Girard, A., Sankaranarayanan, S.: Iterative computation of polyhedral invariants sets for polynomial dynamical systems. In: CDC 2014, pp. 6348–6353. IEEE (2014)
59. Slagel, J.T., White, L., Dutle, A.: Formal verification of semi-algebraic sets and real analytic functions. In: CPP 21, pp. 278–290. ACM (2021)
60. Sogokon, A., Mitsch, S., Tan, Y.K., Cordwell, K., Platzer, A.: Pegasus: A framework for sound continuous invariant generation. In: FM 2019, LNCS, pp. 138–157. Springer (2019)
61. Struth, G.: On the expressive power of Kleene algebra with domain. Inf. Process. Lett. **116**(4), 284–288 (2016)
62. Struth, G.: Hoare semigroups. Math. Struct. Comput. Sci. **28**(6), 775–799 (2018)
63. Struth, G.: Properties of orderings and lattices. In: Archive of Formal Proofs (2018)
64. Struth, G.: Quantales. In: Archive of Formal Proofs (2018)
65. Struth, G.: Transformer semantics. Archive of Formal Proofs (2018). https://isa-afp.org/entries/Transformer_Semantics.html
66. Teschl, G.: Ordinary Differential Equations and Dynamical Systems. AMS (2012)
67. Wang, S., Zhan, N., Zou, L.: An improved HHL prover: an interactive theorem prover for hybrid systems. In: ICFEM 2015, pp. 382–399 (2015)