Proceedings Paper:
Munive, Jonathan Huerta y, Struth, Georg and Foster, Simon David orcid.org/0000-0002-9889-9514 (2020) Differential Hoare Logics and Refinement Calculi for Hybrid Systems with Isabelle/HOL. In: 18th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2020). Lecture Notes in Computer Science . Springer

https://doi.org/10.1007/978-3-030-43520-2_11

Differential Hoare Logics and Refinement Calculi for Hybrid Systems with Isabelle/HOL

Simon Foster[1], Jonathan Julián Huerta y Munive[2], and Georg Struth[2]

[1] University of York, UK
[2] University of Sheffield, UK

Abstract. We present simple new Hoare logics and refinement calculi for hybrid systems in the style of differential dynamic logic. (Refinement) Kleene algebra with tests is used for reasoning about the program structure and generating verification conditions at this level. Lenses capture hybrid program stores in a generic algebraic way. The approach has been formalised with the Isabelle/HOL proof assistant. Several examples explain the workflow with the resulting verification components.

1 Introduction

Differential dynamic logic ($\mathsf{d}\mathcal{L}$) is a prominent deductive method for verifying hybrid systems [26]. It extends dynamic logic with specific inference rules for reasoning about the discrete control and continuous dynamics that characterise such systems. Continuous evolutions are modelled by $\mathsf{d}\mathcal{L}$'s evolution commands within a hybrid program syntax. These declare a vector field and a guard, which is meant to hold along the evolution. Reasoning with evolution commands in $\mathsf{d}\mathcal{L}$ requires either explicit solutions to differential equations represented by the vector field, or invariant sets [28] that describe these evolutions implicitly. Verification components inspired by $\mathsf{d}\mathcal{L}$ have already been formalised in the Isabelle proof assistant [16]. Yet the shallow embedding used in this work has shifted the focus from the original proof-theoretic approach to a semantic one, and ultimately to predicate transformer algebras supporting a different workflow.

Dynamic logics and predicate transformers are powerful tools. They support reasoning about program equivalences and transformations far beyond what standard program verification requires [4]. For the latter, much simpler Hoare logics generate precisely the verification conditions needed. Asking about the feasibility of a *differential Hoare logic* ($\mathsf{d}\mathcal{H}$) is therefore natural. As Hoare logic is strongly related to Morgan's refinement calculus [25], it is equally reasonable to ask whether and how a Morgan-style *differential refinement calculus* ($\mathsf{d}\mathcal{R}$) might allow constructing hybrid programs from specifications.

A prima facie answer to these questions seems positive: after all, the laws of Morgan's refinement calculus can be proved using the rules of Hoare logic, which in turn are derivable within dynamic logic. But the formalisms envisaged might not be expressive enough for hybrid program verification or less suitable than $\mathsf{d}\mathcal{L}$ in practice. Conceptually it is also not obvious what exactly it would take to extend a standard Hoare logic or refinement calculus to hybrid programs.

Our main contribution consists in evidence that $\mathsf{d}\mathcal{H}$ and $\mathsf{d}\mathcal{R}$ are as applicable for verifying simple hybrid programs as $\mathsf{d}\mathcal{L}$, and that developing these methods requires simply adding a single Hoare-style axiom and a single refinement rule for evolution commands to the standard formalisms.

This conceptual simplicity is reflected in the Isabelle verification components for $\mathsf{d}\mathcal{H}$ and $\mathsf{d}\mathcal{R}$. These reuse components for (refinement) Kleene algebra with tests [19,3,13] ((r)KAT) for the propositional Hoare logic and refinement calculi, ignoring assignment and evolution commands. The axioms and laws for these basic commands are derived in a concrete state transformer semantics for hybrid programs [15] over a generic hybrid store model based on lenses [10], reusing other Isabelle components [15,8,9]. Data-level verification conditions are discharged using Isabelle's impressive components for ordinary differential equations [17].

This simple modular development evidences the benefits of algebraic reasoning and shallow embeddings with proof assistants. Our verification components merely require formalising a state transformer semantics for KAT and rKAT along the lines of [16] and concrete store semantics for hybrid programs. Lenses [10] give us the flexibility to switch seamlessly between stores based on real vector spaces or executable Euclidean spaces. Beyond that it suffices to derive a few algebraic laws for invariants and the Hoare-axioms and refinement laws for evolution commands in the concrete semantics. Program verification is then performed at the concrete level, but this remains hidden, as tactics generate data-level verification conditions automatically and we have programmed boiler-plate syntax for programs and correctness specifications.

Our Isabelle components support the workflows of $\mathsf{d}\mathcal{L}$ in $\mathsf{d}\mathcal{H}$ and $\mathsf{d}\mathcal{R}$. We may reason explicitly with solutions to differential equations and implicitly with invariant sets. We have formalised a third method in which solutions, that is flows, are declared ab initio in correctness specifications and need not be certified.

Our program construction and verification components have so far been evaluated on a small set of simple examples. Further work is needed to evidence scalability or compare performance with the standard $\mathsf{d}\mathcal{L}$ tool chain. We present some examples to explain the work flows supported by $\mathsf{d}\mathcal{H}$ and $\mathsf{d}\mathcal{R}$. With Isabelle tactics for automated verification condition generation in place, we notice little difference relative to our predicate transformer components [16]. The entire Isabelle formalisation is available online[3].

2 Kleene Algebra with Tests

A *Kleene algebra with tests* [19] (KAT) is a structure $(K, B, +, \cdot, 0, 1, ^*, \neg)$ where $(B, +, \cdot, 0, 1, \neg)$ is a boolean algebra with join $+$, meet $\cdot$, complementation $\neg$, least element 0 and greatest element 1, $B \subseteq K$, and $(K, +, \cdot, 0, 1, ^*)$ is a Kleene algebra—a semiring with idempotent addition equipped with a star operation that satisfies the axioms $1 + \alpha \cdot \alpha^* \leq \alpha^*$ and $\gamma + \alpha \cdot \beta \leq \beta \rightarrow \alpha^* \cdot \gamma \leq \beta$, as well

[3] https://github.com/yonoteam/HybridKATpaper

as their opposities, with multiplication swapped. The ordering on K is defined by $\alpha \leq \beta \leftrightarrow \alpha + \beta = \beta$, as idempotent semirings are semilattices.

Elements of K represent programs; those of B tests, assertions or propositions. The operation $\cdot$ models the sequential composition of programs[4], $+$ their nondeterministic choice, $(-)^*$ their finite unbounded iteration. Program 0 aborts and 1 skips. Tests are embedded implicitly into programs. They are meant to hold in some states of a program and fail in others; $p\alpha$ (αp) restricts the execution of program α in its input (output) to those states where test p holds. The ordering $\leq$ is the opposite of the refinement ordering on programs (see Section 7).

Binary relations of type $\mathcal{P}\,(S \times S)$ form KATs [19] when $\cdot$ is interpreted as relational composition, $+$ as relational union, $(-)^*$ as reflexive-transitive closure and the elements of B as subidentities—relations below the relational unit. This grounds KAT within standard relational imperative program semantics. However, we prefer the isomorphic representation known as *state transformers* of type $S \to \mathcal{P}\,S$. Composition $\cdot$ is then interpreted as Kleisli composition

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

0 as $\lambda x.\ \emptyset$ and 1 as $\eta_S = \{-\}$. Stars $f^*\,s = \bigcup_{i\in\mathbb{N}} f^i\,s$ are defined with respect to Kleisli composition using $f^0 = \eta_S$ and $f^{n+1} = f \circ_K f^n$. The boolean algebra of tests has carrier set $B_S = \{f : S \to \mathcal{P}\,S \mid f \leq \eta_S\}$, where the order on functions has been extended pointwise, and complementation is given by

$$\overline{f}\,x = \begin{cases} \eta_S\,x, & \text{if } f\,x = \emptyset, \\ \emptyset, & \text{otherwise.} \end{cases}$$

We freely identify predicates, sets and state transformers below η_S, which are isomorphic: $P \cong \{s \mid P\,s\} \cong \lambda s.\ \{x \mid x = s \wedge P\,s\}$.

Proposition 2.1. $\mathsf{Sta}\,S = ((\mathcal{P}\,S)^S, B_S, \cup, \circ_K, \lambda x.\ \emptyset, \eta_S, (-)^*, \overline{(-)})$ *forms a* KAT*, the* full state transformer KAT *over the set* S*.*

A *state transformer* KAT over S is any subalgebra of $\mathsf{Sta}\,S$.

KAT has been formalised via type classes in Isabelle [2]. As these allow only one type parameter, we use an alternative to the standard two-sorted approach that expands a Kleene algebra K by an *antitest* function $n : K \to K$ from which a *test* function $t : K \to K$ is defined as $t = n \circ n$. Then $K_t = \{\alpha \mid t\,\alpha = \alpha\}$ forms a boolean algebra in which n yields test complementation. It can be used in place of B. The state transformer KAT has been formalised for this article.

3 Propositional Hoare Logic and Invariants

KAT provides a simple algebraic semantics for while programs with

$$\textbf{if } p \textbf{ then } \alpha \textbf{ else } \beta = p \cdot \alpha + \neg p \cdot \beta \qquad \text{and} \qquad \textbf{while } p \textbf{ do } \alpha = (p \cdot \alpha)^* \cdot \neg p.$$

[4] We therefore often write ; for this operation in later sections.

It captures validity of Hoare triples in a partial correctness semantics as

$$\{p\}\,\alpha\,\{q\} \leftrightarrow p \cdot \alpha \cdot \neg q = 0,$$

or equivalently by $p \cdot \alpha \leq \alpha \cdot q$ or $p \cdot \alpha = p \cdot \alpha \cdot q$. It also allows deriving the rules of *propositional Hoare logic* [20]—disregarding assignments—which are useful for verification condition generation:

$$\begin{aligned}
&\{p\}\,\mathit{skip}\,\{p\}, && \text{(h-skip)}\\
p \leq p' \wedge \{p'\}\,\alpha\,\{q'\} \wedge q' \leq q \;&\rightarrow \{p\}\,\alpha\,\{q\}, && \text{(h-cons)}\\
\{p\}\,\alpha\,\{r\} \wedge \{r\}\,\beta\,\{q\} \;&\rightarrow \{p\}\,\alpha \cdot \beta\,\{q\}, && \text{(h-seq)}\\
\{t \cdot p\}\,\alpha\,\{q\} \wedge \{\neg t \cdot p\}\,\beta\,\{q\} \;&\rightarrow \{p\}\,\mathbf{if}\ t\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta\,\{q\}, && \text{(h-cond)}\\
\{t \cdot p\}\,\alpha\,\{p\} \;&\rightarrow \{p\}\,\mathbf{while}\ t\ \mathbf{do}\ \alpha\,\{\neg t \cdot p\}. && \text{(h-while)}
\end{aligned}$$

Rules for commands with invariant assertions $\alpha\ \mathbf{inv}\ i$ are derivable in KAT, too (operationally, $\alpha\,\mathbf{inv}\,i = \alpha$). An *invariant* for $\alpha \in K$ is a test $i \in B$ satisfying $\{i\}\,\alpha\,\{i\}$. Then, with $\mathbf{loop}\,\alpha$ as syntactic sugar for α^*, we obtain

$$\begin{aligned}
p \leq i \wedge \{i\}\,\alpha\,\{i\} \wedge i \leq q \;&\rightarrow \{p\}\,\alpha\,\{q\}, && \text{(h-inv)}\\
\{i\}\,\alpha\,\{i\} \wedge \{j\}\,\alpha\,\{j\} &\rightarrow \{i \cdot j\}\,\alpha\,\{i \cdot j\}, && \text{(h-inv-mult)}\\
\{i\}\,\alpha\,\{i\} \wedge \{j\}\,\alpha\,\{j\} &\rightarrow \{i + j\}\,\alpha\,\{i + j\}, && \text{(h-inv-plus)}\\
p \leq i \wedge \{i \cdot t\}\,\alpha\,\{i\} \wedge \neg t \cdot i \leq q \;&\rightarrow \{p\}\,\mathbf{while}\ t\ \mathbf{inv}\ i\ \mathbf{do}\ \alpha\,\{q\}, && \text{(h-while-inv)}\\
p \leq i \wedge \{i\}\,\alpha\,\{i\} \wedge i \leq q \;&\rightarrow \{p\}\,\mathbf{loop}\,\alpha\,\mathbf{inv}\,i\,\{q\}. && \text{(h-loop-inv)}
\end{aligned}$$

We use (h-inv) for invariants for continuous evolutions of hybrid systems in Section 6-8. The rules (h-inv-mult) and (h-inv-plus) are part of a procedure, described in Section 6. Rule (h-while-inv) is standard for invariants for while loops; (h-loop-inv) is specific to loops of hybrid programs (see Section 4). The rules for propositional Hoare logic in Isabelle have already been derived for KAT [2,13], those for invariants have been formalised for this work. By Proposition 2.1, all of them hold in particular in the state transformer semantics. We have formalised this fact with Isabelle. At this stage, verification condition rules for the basic commands for assignments and evolution commands are still missing. These are formalised within the concrete state transformer semantics (see Section 5).

4 State Transformer Semantics for Hybrid Programs

Hybrid programs of differential dynamic logic (d$\mathcal{L}$) [26] are defined by the syntax

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

that adds *evolution commands* $x' = f\,\&\,G$ to the language of KAT—function $?(-)$ embeds tests explicitly into programs; in the tradition of KAT we leave this embedding implicit. Evolution commands introduce a time independent vector field $f : S \to S$ for an autonomous system of ordinary differential equations

(ODEs) [28] together with a guard G, a predicate modelling boundary conditions or similar restrictions that hold along temporal evolutions. Guards are also known as *evolution domain restrictions* [6].

Formally, we fix a state space S of the hybrid program such as $S \subseteq \mathbb{R}^n$ for $n \in \mathbb{N}$. We model continuous variables algebraically with lenses [10] to support different state space models generically. A lens $x : A \Longrightarrow S$ is a tuple $x = (A, S, \mathit{get}, \mathit{put})$, where A is a variable type. The functions $\mathit{get}_x : S \to A$ and $\mathit{put}_x : S \to A \to S$ query and update the value of x in a particular state. They are linked by three intuitive algebraic laws [10]. For all $s \in S$ and $v, v' \in A$,

$$\mathit{get}\ (\mathit{put}\ s\ v) = v, \qquad \mathit{put}\ (\mathit{put}\ s\ v')\ v = \mathit{put}\ s\ v, \qquad \mathit{put}\ s\ (\mathit{get}\ s) = s.$$

The predicate $x \bowtie y$ checks independence of lenses x and y, which holds when x and y refer to two different regions of S. As each program variable is a lens $x : \mathbb{R} \Longrightarrow S$, state spaces $S \subseteq \mathbb{R}^n$ require n independent lenses $x_1 \cdots x_n$. Yet more general state spaces such as vector spaces are supported as well.

Systems of ODEs are modelled using vector fields: functions of type $S \to S$ on some open set S. Geometrically, vector field f assigns vectors to each point of the state space S. A solution to the *initial value problem* (IVP) for the pair (f, s) and initial value $(0, s) \in T \times S$, where T is an open interval in $\mathbb{R}$ containing 0, is then a function $X : T \to S$ that satisfies $X'\, t = f\,(X\, t)$—an autonomous system of ODEs in vector form—and $X\, 0 = s$. Solution X is thus a curve in S through s, parametrised by T and tangential to f at any point in S; it is called the *trajectory* or *integral curve* of f at s whenever it is uniquely defined [28]. For IVP (f, s) with continuous vector field $f : S \to S$ and initial state $s \in S$ we define the set of solutions on T as

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

Each solution X is thus continuously differentiable and hence $f \circ X$ integrable in T. For $X \in \mathsf{Sols}\ f\, T\, s$ and guard $G : S \to \mathbb{B}$, we then define the *G-guarded orbit* of X along T in s [16] as a state transformer $\gamma_G^X : S \to \mathcal{P}\, S$ by

$$\gamma_G^X\ s = \{X\, t \mid t \in T \wedge \forall \tau \in {\downarrow} t.\ G\,(X\, \tau)\}\,,$$

where ${\downarrow} t = \{t' \in T \mid t' \leq t\}$. Intuitively, $\gamma_G^X\ s$ is the orbit at s defined along the longest interval in T that satisfies guard G. We also define the *G-guarded orbital* of f along T in s [16] via the state transformer $\gamma_G^f : S \to \mathcal{P}\, S$ as

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

In applications, ${\downarrow} t$ is usually an interval $[0, t] \subseteq T$. Expanding definitions,

$$\gamma_G^f\ s = \{X\, t \mid X \in \mathsf{Sols}\ f\, T\, s \wedge t \in T \wedge \forall \tau \in {\downarrow} t.\ G\,(X\, \tau)\}\,.$$

If $\top$ denotes the predicate that holds of all states in S (or the set S itself), we write γ^f instead of $\gamma_\top^f$. We define the semantics of the evolution command $x' = f\,\&\,G$ [16] for any continuous $f : S \to S$ and $G : S \to \mathbb{B}$ as

$$(x' = f\,\&\,G) = \gamma_G^f. \tag{st-evl}$$

In the evolution command $x' = f \,\&\, G$, x' is part of the traditional syntax used for specifying systems of ODEs, while de facto only a vector field f is specified. This explains why x does not appear in the right-hand side of (st-evl). Defining the state transformer semantics of assignments is standard [16], though we generalise using lenses. First, we use lenses to define state updates:

$$\sigma(x \mapsto e) = \lambda s.\ \mathit{put}_x\ (\sigma\ s)\ (e\ s)$$

for $x : A \Longrightarrow S$, $e : S \to A$, and $\sigma : S \to S$. Intuitively, this updates the value of variable x in $\sigma : S \to S$ to the value given by "expression" e in state s. For variables x and y, for example, the expression $x/(2 + y)$ is modelled by $\lambda s.\ \mathit{get}_x\ s\ /\ (2 + \mathit{get}_y\ s)$. We can also update n variables simultaneously:

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

where *id* is the identity function. State updates commute when assigning to independent lenses; they cancel one another out, when made to the same lens. We can then define a semantic analog of the substitution operator $e[f/x] = e \circ [x \mapsto f]$ that satisfies the standard laws [10]. Finally, we define the generalised simultaneous assignment operator

$$\langle \sigma \rangle = \lambda s.\ \{\sigma(s)\}. \tag{st-assgn}$$

that applies $\sigma : S \to S$ as an assignment. With our state update function, singleton assignment is a special case: $(x := e) = \langle x \mapsto e \rangle$. These concepts allow us to derive standard laws for assignments, as for instance in schematic KAT [1]:

$$\begin{aligned}
x := x &= \mathit{skip}, \\
x := e\ ;\ x := f &= x := f[e/x], \\
x := e\ ;\ y := f &= y := f\ ;\ x := e, && \text{if } x \bowtie y, x \sharp f, y \sharp e, \\
x := e\ ;\ \textbf{if } t \textbf{ then } \alpha \textbf{ else } \beta &= \textbf{if } t[e/x] \textbf{ then } x := e\ ;\ \alpha \textbf{ else } x := e\ ;\ \beta.
\end{aligned}$$

Here, $x \sharp e$ means that the semantic expression e does not depend in its valuation on lens x [10]. An assignment of x to itself is simply *skip*. Two assignments to x result in a single assignment, with a semantic substitution applied. Assignments to independent variables x and y commute provided that neither assigned expression depends on the corresponding variable. Assignment can be distributed through conditionals by a substitution to the condition. Such laws can be applied recursively for symbolic evaluation of hybrid programs.

Lenses support various store models, including records and functions [10]. We provide models for vector spaces, executable and infinite Euclidean spaces:

$$\begin{aligned}
\mathit{vec\text{-}lens}_k^n &= (\mathbb{R}, \mathbb{R}^n, \lambda s.\ \mathit{vec\text{-}nth}\ s\ k, \lambda s\ v.\ \mathit{vec\text{-}upd}\ k\ v\ s), && \text{if } k < n, \\
\mathit{eucl\text{-}lens}_k^n &= (\mathbb{R}, V, \lambda s.\ \mathit{eucl\text{-}nth}\ s\ k, \lambda s\ v.\ \mathit{eucl\text{-}upd}\ k\ v\ s), && \text{if } k < n, \\
\mathit{fun\text{-}lens}_i^{(A,B)} &= (B, A \to B, (\lambda f.fi), (\lambda f\ v.\ f(i := v))).
\end{aligned}$$

The vector lens selects the kth element of an n dimension vector using *vec-nth* and *vec-upd* from the HOL Analysis library [14], which provides an indexed type for the space $\mathbb{R}^n$. The Euclidean lens uses executable Euclidean spaces [18] that provide a list representation of the vectors in the n-dimensional V via an ordered basis and an inner product. The function lens selects range elements of a function associated with a domain element $i \in A$. It can be used in particular with infinite Euclidean spaces, $\mathbb{N} \to \mathbb{R}$. All three satisfy the lens axioms above.

The development in this section has been formalised with Isabelle [15,8,9], both for a state transformer and a relational semantics. An instance of the latter for particular vector fields with unique solutions forms the standard semantics of $\mathsf{d}\mathcal{L}$. By the direct connection to orbits or orbitals, the state transformer semantics is arguably conceptually simpler and more elegant.

5 Differential Hoare Logic for Flows

In the state transformer semantics of Hoare triples, the Kleisli composition in the left-hand side of $p \cdot \alpha \leq \alpha \cdot q$ ensures that p holds before executing α. The right-hand side guarantees that q holds after its execution. Specifically for evolution commands, and consistently with $\mathsf{d}\mathcal{L}$, q holds along the entire orbit of a solution for f. We now complete the derivation of inference rules of $\mathsf{d}\mathcal{H}$ by adding Hoare-style rules for assignments and evolution commands in the concrete state transformer semantics.

The assignment axiom of Hoare logic needs no explanation. Our concrete state transformer semantics allows us to derive it:

$$\{P[e/x]\}\ x := e\ \{P\}. \tag{h-assgn}$$

Hence all we need to add to Hoare logic is a rule for evolution commands. We restrict our attention to Lipschitz-continuous vector fields for which unique solutions to IVPs are guaranteed by Picard-Lindelöf's theorem [28]. These are *(local) flows* $\varphi : T \to S \to S$ and $X = \varphi_s = \lambda t.\ \varphi\, t\, s$ is the trajectory at s. Guarded orbitals γ^f_G then specialise to *guarded orbits*

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

where T is fixed by the Picard-Lindelöf theorem and $U \subseteq T$ is a time domain of interest, typically an interval $[0, t]$ for some $t \in T$ [16] where, by contrast to the previous section ${\downarrow} t = \{t' \in U \mid t' \leq t\}$ is relativised to U. This gives us the flexibility to consider dynamics over closed time intervals and it allows us to focus on time intervals and IVPs starting at $t = 0$. Accordingly, (st-evl) specialises to the following state transformer semantics for evolution commands.

$$(x' = f\,\&\,G) = \gamma^f_{G,U}. \tag{st-evl-flow}$$

The following Hoare-style rule for evolution commands is then derivable.

Lemma 5.1. *Let $f : S \to S$ be a Lipschitz continuous vector field on $S \subseteq \mathbb{R}^n$ and $\varphi : T \to S \to S$ its local flow with $0 \in T \subseteq \mathbb{R}$. Then, for $U \subseteq T$ with $0 \in U$ and $G, Q : S \to \mathbb{B}$,*

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

This finishes the derivation of rules for a Hoare logic $\mathsf{d}\mathcal{H}$ for hybrid programs—to our knowledge, the first Hoare logic of this kind. As usual, there is one rule per programming construct, so that the recursive application of the rules of propositional Hoare logic together with (h-assgn) and (h-evl) generates proof obligations that are entirely about data-level relationships—the discrete and continuous evolution of hybrid program stores.

The rule (h-evl) supports the following procedure for reasoning with an evolution command $x' = f\,\&\,G$ and set U in $\mathsf{d}\mathcal{H}$:

1. Check that f satisfies the conditions for Picard-Lindelöf's theorem (f is Lipschitz continuous and $S \subseteq \mathbb{R}^n$ is open).
2. Supply a (local) flow φ for f with open interval of existence T around 0.
3. Check that φ_s solves the IVP (f, s) for each $s \in S$; $(\varphi_s'\,t = f\,(\varphi_s\,t)$, $\varphi_s\,0 = s$, and $U \subseteq T)$.
4. If successful, apply rule (h-evl).

Example 5.2 (Thermostat verification via solutions). A thermostat regulates the temperature T of a room between bounds $T_l \leq T \leq T_h$. Variable T_0 stores an initial temperature; ϑ indicates whether the heater is switched on or off. Within time intervals of at most τ minutes, the thermostat resets time to 0, measures the temperature, and turns the heater on or off dependent on the value obtained. With $0 < T_l$, $T_h < T_u$, $0 < a$, $U = \{0..\tau\} = [0, \tau]$ we define f, for $c \in \{0, T_u\}$, as

abbreviation $f\ a\ c \equiv [T \mapsto_s -\,(a * (T - c)),\ T_0 \mapsto_s 0,\ \vartheta \mapsto_s 0,\ t \mapsto_s 1]$

The notation $x \mapsto_s f\,x$ indicates that vector field $f\,a\,c$ maps variable x to $f\,x$ for $x \in \{T, T_0, \vartheta, t\}$. Working with *vec-lens*^n_k or *eucl-lens*^n_k, we write ; instead of $\cdot$ and use guard G to restrict evolutions between T_l and T_h by setting

$$G\,T_l\,T_h\,a\,c = \left(t \leq -\frac{1}{a} \ln\left(\frac{c - \Delta_c}{c - T_0}\right)\right),$$

where $\Delta_c = T_l$ if $c = 0$, and $\Delta_c = T_h$ if $c = T_u$. The hybrid program *therm* below models the behaviour of the thermostat. To simplify notation, we separate into a loop invariant (I), discrete control (*ctrl*), and continuous dynamics (*dyn*).

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

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

abbreviation *dyn* $T_l\ T_h\ a\ T_u\ \tau \equiv$
IF $(\vartheta = 0)$ *THEN* $x' = f\ a\ 0\ \&\ G\ T_l\ T_h\ a\ 0$ *on* $\{0..\tau\}$ *UNIV* @ *0*
ELSE $x' = f\ a\ T_u\ \&\ G\ T_l\ T_h\ a\ T_u$ *on* $\{0..\tau\}$ *UNIV* @ *0*

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

The correctness specification and verification of the thermostat with $\mathsf{d}\mathcal{H}$ is then

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

The first line uses tactic *hyb-hoare* to blast away the structure of *therm* using $\mathsf{d}\mathcal{H}$. To apply *hyb-hoare*, the program must be an iteration of the composition of two programs—usually control and dynamics. The tactic requires lifting the store to an Isabelle/UTP expression [10], which is denoted by the $\mathbf{U}$ operator. Lemma *local-flow-therm*, whose proof captures the procedure described above, supplies the flow for $f\,a\,c$: $\varphi\,a\,c\,\tau = (-e^{-a\cdot\tau}(c - T) + c, \tau + t, T_0, \vartheta)^\top$, for all $\tau \in \mathbb{R}$. The remaining proof obligations are inequalities of transcendental functions. They are discharged automatically using auxiliary lemmas. □

6 Differential Hoare Logic for Invariants

Alternatively, $\mathsf{d}\mathcal{H}$ supports reasoning with invariants for evolution commands instead of supplying flows to (h-evl). The approach has been developed in [16]. Our invariants generalise the *differential invariants* of $\mathsf{d}\mathcal{L}$ [26] and the *invariant sets* of dynamical systems and (semi)group theory [28].

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

$$\bigcup \mathcal{P}\,\gamma_G^f\,I \subseteq I.$$

The operation $\bigcup \circ \mathcal{P}$ is the Kleisli extension $(-)^\dagger$ in the powerset monad. Hence we could simply write $(\gamma_G^f)^\dagger\,I \subseteq I$. The definition of invariance unfolds to

$$\forall s.\ I\,s \to (\forall X \in \mathsf{Sols}\,f\,T\,s. \forall t \in T.\ (\forall \tau \in {\downarrow}t.\ G\,(X\,\tau)) \to I\,(X\,t)).$$

For $G = \top$ we call I an *invariant* of f along T. Intuitively, invariants can be seen as sets of orbits. They are compatible with the invariants from Section 3.

Proposition 6.1. *Let* $f : S \to S$ *be continuous,* $G : S \to \mathbb{B}$ *and* $T \subseteq \mathbb{R}$. *Then* I *is an invariant for* f *and* G along T *if and only if* $\{I\}\,x' = f\,\&\,G\,\{I\}$.

Hence we can use a variant of (h-inv) for verification condition generation:

$$P \leq I \wedge \{I\}\, x' = f \,\&\, G\, \{I\} \wedge (I \cdot G) \leq Q \;\rightarrow \{P\}\, x' = f \,\&\, G\, \{Q\}. \quad \text{(h-invg)}$$

The following lemma leads to a procedure.

Lemma 6.2 ([16]). *Let $f : S \to S$ be a continuous vector field, $\mu, \nu : S \to \mathbb{R}$ differentiable and $T \subseteq \mathbb{R}$ an interval such that $0 \in T$.*

1. *If $(\mu \circ X)' = (\nu \circ X)'$ for all $X \in \mathsf{Sols}\, f\, T\, s$, then $\{\mu = \nu\}\, x' = f \,\&\, G\, \{\mu = \nu\}$,*
2. *if $(\mu \circ X)'\, t \leq (\nu \circ X)'\, t$ when $t > 0$, and $(\mu \circ X)'\, t \geq (\nu \circ X)'\, t$ when $t < 0$, for all $X \in \mathsf{Sols}\, f\, T\, s$, then $\{\mu < \nu\}\, x' = f \,\&\, G\, \{\mu < \nu\}$*
3. *if $\{\mu < \nu\}\, x' = f \,\&\, G\, \{\mu < \nu\}$ and $\{\mu > \nu\}\, x' = f \,\&\, G\, \{\mu > \nu\}$, then $\{\mu \neq \nu\}\, x' = f \,\&\, G\, \{\mu \neq \nu\}$ (and conversely if 0 is the least element in T),*
4. *$\{\mu \not\leq \nu\}\, x' = f \,\&\, G\, \{\mu \not\leq \nu\}$ if and only if $\{\mu > \nu\}\, x' = f \,\&\, G\, \{\mu > \nu\}$.*

Condition (1) follows from the well known fact that two continuously differentiable functions are equal if they intersect at some point and have the same derivatives. Rules (h-invg), (h-inv-mult), (h-inv-plus), Proposition 6.1 and Lemma 6.2 yield the following procedure for verifying $\{P\}\, x' = f \,\&\, G\, \{Q\}$:

1. Check whether candidate predicate I is an invariant for f along T:
   (a) transform I into negation normal form;
   (b) reduce complex I (with (h-inv-mult), (h-inv-plus) and Lemma 6.2 (3,4);
   (c) if I is atomic, apply Lemma 6.2 (1) and (2);
   (if successful, $\{I\}\, x' = f \,\&\, G\, \{I\}$ holds by Proposition 6.1),
2. if successful, prove $P \leq I$ and $(I \cdot G) \leq Q$ to apply rule (h-invg).

Example 6.3 (Water tank verification via invariants). A controller turns a water pump on and off to keep the water level h in a tank within bounds $h_l \leq h \leq h_h$. Variable h_0 stores an initial water level; π indicates whether the pump is on or off. The rate of change of the water-level is linear with slope $k \in \{-c_o, c_i - c_o\}$ (assuming $c_i > c_o$). The vector field f for this behaviour and its invariant dI are

abbreviation $f\ k \equiv [\pi \mapsto_s 0,\ h \mapsto_s k,\ h_0 \mapsto_s 0,\ t \mapsto_s 1]$

abbreviation $dI\ h_l\ h_h\ k \equiv$
$\mathbf{U}(h = k \cdot t + h_0 \wedge 0 \leq t \wedge h_l \leq h_0 \wedge h_0 \leq h_h \wedge (\pi = 0 \vee \pi = 1))$

Program *tank-dinv* for the controller is given by guard $G\ h_x\ k$ with $h_x \in \{h_l, h_h\}$ that restricts evolutions beyond h_x, loop invariant I, control and dynamics:

abbreviation $G\ h_x\ k \equiv \mathbf{U}(t \leq (h_x - h_0)/k)$

abbreviation $I\ h_l\ h_h \equiv \mathbf{U}(h_l \leq h \wedge h \leq h_h \wedge (\pi = 0 \vee \pi = 1))$

abbreviation $dyn\ c_i\ c_o\ h_l\ h_h\ \tau \equiv IF\ (\pi = 0)\ THEN$
$x' = f\ (c_i - c_o)\ \&\ G\ h_h\ (c_i - c_o)\ on\ \{0..\tau\}\ UNIV\ @\ 0\ DINV\ (dI\ h_l\ h_h\ (c_i - c_o))$
$ELSE\ x' = f\ (-c_o)\ \&\ G\ h_l\ (-c_o)\ on\ \{0..\tau\}\ UNIV\ @\ 0\ DINV\ (dI\ h_l\ h_h\ (-c_o))$

abbreviation *ctrl* $h_l\ h_h \equiv$
$(t ::= 0);(h_0 ::= h);$
$(IF\ (\pi = 0 \wedge h_0 \leq h_l + 1)\ THEN\ (\pi ::= 1)\ ELSE$
$(IF\ (\pi = 1 \wedge h_0 \geq h_h - 1)\ THEN\ (\pi ::= 0)\ ELSE\ skip))$

abbreviation *tank-dinv* $c_i\ c_o\ h_l\ h_h\ \tau \equiv$
$LOOP\ (ctrl\ h_l\ h_h;\ dyn\ c_i\ c_o\ h_l\ h_h\ \tau)\ INV\ (I\ h_l\ h_h)$

We distinguish DINV and INV to structure specifications. The correctness specification and verification of the water tank with $\mathsf{d}\mathcal{H}$ then proceeds as follows:

lemma *tank-diff-inv*: $0 \leq \tau \Longrightarrow$ *diff-invariant* $(dI\ h_l\ h_h\ k)\ (f\ k)\ \{0..\tau\}$ *UNIV 0 Guard*
$\langle proof \rangle$

lemma *tank-inv*:
assumes $0 \leq \tau$ **and** $0 < c_o$ **and** $c_o < c_i$
shows $\{I\ h_l\ h_h\}$ *tank-dinv* $c_i\ c_o\ h_l\ h_h\ \tau$ $\{I\ h_l\ h_h\}$
apply(*hyb-hoare* $\mathbf{U}(I\ h_l\ h_h \wedge t = 0 \wedge h_0 = h)$)
prefer *4* **prefer** *7* **using** *tank-diff-inv assms* **apply** *force+*
using *assms tank-inv-arith1 tank-inv-arith2* **by** *rel-auto′*

Tactic *hyb-hoare* blasts away the control structure. The second proof line uses Lemma *tank-diff-inv* to check that *dI* is an invariant for any guard (*Guard* is a universally quantified variable in Lemma *tank-diff-inv*), using the procedure outlined. Auxiliary lemmas discharge the remaining proof obligations. □

7 Differential Refinement Calculi

A *refinement Kleene algebra with tests* (rKAT) [3] is a KAT (K, B) expanded by an operation $[-,-] : B \times B \to K$ that satisfies, for all $\alpha \in K$ and $p, q \in B$,

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

The element $[p,q]$ of K corresponds to Morgan's *specification statement* [25]. It satisfies $\{p\}\,[p,q]\,\{q\}$ and $\{p\}\,\alpha\,\{q\} \to \alpha \leq [p,q]$, which makes $[p,q]$ the greatest element of K that satisfies the Hoare triple with precondition p and postcondition q. Indeed, in $\mathsf{Sta}\,S$ and for $S \subseteq \mathbb{R}^n$, $[P,Q] = \bigcup\{f : S \to \mathcal{P}\,S \mid \{P\}\,f\,\{Q\}\}$. Variants of Morgan's laws [25] of a *propositional refinement calculus*—once more ignoring assignments—are then derivable in rKAT [3].

$$1 \leq [p,p], \qquad \text{(r-skip)}$$
$$[p',q'] \leq [p,q], \quad \text{if } p \leq p' \text{ and } q' \leq q, \qquad \text{(r-cons)}$$
$$[p,r] \cdot [r,q] \leq [p,q], \qquad \text{(r-seq)}$$
$$\textbf{if } t \textbf{ then } [t \cdot p, q] \textbf{ else } [\neg t \cdot p, q] \leq [p,q], \qquad \text{(r-cond)}$$
$$\textbf{while } t \textbf{ do } [t \cdot p, p] \leq [p, \neg t \cdot p]. \qquad \text{(r-while)}$$

We have also derived $\alpha \leq [0,1]$ and $[1,0] \leq \alpha$, but do not use them in proofs.

For invariants and loops, we obtain the additional refinement laws

$$[i,i] \leq [p,q], \qquad \text{if } p \leq i \leq q, \tag{r-inv}$$
$$\mathbf{loop}\,[i,i] \leq [i,i]. \tag{r-loop}$$

In $\mathsf{Sta}\,S$, moreover, the following assignments laws are derivable [3].

$$(x := e) \leq [Q[e/x], Q]\,, \tag{r-assgn}$$
$$(x := e) \cdot [Q,Q] \leq [Q[e/x], Q], \tag{r-assgnl}$$
$$[Q, Q[e/x]] \cdot (x := e) \leq [Q,Q]. \tag{r-assgnf}$$

The second and third law are known as *leading* and *following* law. They introduce an assignment before and after a block of code.

Finally, we obtain the following refinement laws for evolution commands.

Lemma 7.1. *Let $f : S \to S$ be a Lipschitz continuous vector field on $S \subseteq \mathbb{R}^n$ and $\varphi : T \to S \to S$ its local flow with $0 \in T \subseteq \mathbb{R}$. Then, for $U \subseteq T$ with $0 \in U$ and $G, Q : S \to \mathbb{B}$,*

$$(x' = f\,\&\,G) \leq [\lambda s.\forall t \in U.\ (\forall \tau \in \downarrow t.\ G\,(\varphi_s\,\tau)) \to Q\,(\varphi_s\,t), Q], \tag{r-evl}$$
$$(x' = f\,\&\,G) \cdot [Q,Q] \leq [\lambda s.\forall t \in U.\ (\forall \tau \in \downarrow t.\ G\,(\varphi_s\,\tau)) \to Q\,(\varphi_s\,t), Q], \tag{r-evll}$$
$$[Q, \lambda s.\forall t \in U.\ (\forall \tau \in \downarrow t.\ G\,(\varphi_s\,\tau)) \to Q\,(\varphi_s\,t)] \cdot (x' = f\,\&\,G) \leq [Q,Q]. \tag{r-evlr}$$

The laws in this section form the differential refinement calculus $\mathsf{d}\mathcal{R}$. They suffice for constructing hybrid programs from initial specification statements by step-wise refinement incrementally and compositionally. A more powerful variant based on predicate transformers à la Back and von Wright [4] has been developed in [16]; yet applications remain to be explored. A previous approach to refinement in $\mathsf{d}\mathcal{L}$ [23] is quite different to the two standard calculi mentioned (see Conclusion).

Example 7.2 (Thermostat refinement via solutions). We now construct program *therm* from Example 5.2 by step-wise refinement using the rules of $\mathsf{d}\mathcal{R}$.

lemma *R-therm-down*:
assumes $a > 0$ **and** $0 \leq \tau$ **and** $0 < T_l$ **and** $T_h < T_u$
shows $[\vartheta = 0 \wedge I\ T_l\ T_h \wedge t = 0 \wedge T_0 = T,\ I\ T_l\ T_h] \geq$
$(x' = f\ a\ 0\ \&\ G\ T_l\ T_h\ a\ 0\ on\ \{0..\tau\}\ UNIV\ @\ 0)$
apply(*rule local-flow.R-g-ode-ivl*[*OF local-flow-therm*])
using *therm-dyn-down*[*OF assms*(*1*,*3*), *of - T_h*] *assms* **by** *rel-auto'*

lemma *R-therm-up*:
assumes $a > 0$ **and** $0 \leq \tau$ **and** $0 < T_l$ **and** $T_h < T_u$
shows $[\neg\ \vartheta = 0 \wedge I\ T_l\ T_h \wedge t = 0 \wedge T_0 = T,\ I\ T_l\ T_h] \geq$
$(x' = f\ a\ T_u\ \&\ G\ T_l\ T_h\ a\ T_u\ on\ \{0..\tau\}\ UNIV\ @\ 0)$
apply(*rule local-flow.R-g-ode-ivl*[*OF local-flow-therm*])
using *therm-dyn-up*[*OF assms*(*1*) *- - assms*(*4*), *of T_l*] *assms* **by** *rel-auto'*

lemma *R-therm-time*: $[I\ T_l\ T_h,\ I\ T_l\ T_h \wedge t = 0] \geq (t ::= 0)$
by (*rule R-assign-law*, *pred-simp*)

lemma *R-therm-temp*: $[I\ T_l\ T_h \wedge t = 0,\ I\ T_l\ T_h \wedge t = 0 \wedge T_0 = T] \geq (T_0 ::= T)$
by (*rule R-assign-law*, *pred-simp*)

lemma *R-thermostat-flow*:
assumes $a > 0$ **and** $0 \leq \tau$ **and** $0 < T_l$ **and** $T_h < T_u$
shows $[I\ T_l\ T_h,\ I\ T_l\ T_h] \geq therm\ T_l\ T_h\ a\ T_u\ \tau$
by (*refinement*;(*rule R-therm-time*)*?*,(*rule R-therm-temp*)*?*,(*rule R-assign-law*)*?*,
(*rule R-therm-up*[*OF assms*])*?*, (*rule R-therm-down*[*OF assms*])*?*) *rel-auto′*

The *refinement* tactic pushes the refinement specification through the program structure until the only remaining proof obligations are atomic refinements. We only refine the atomic programs needed to complete proofs automatically; those for the first two assignment and the evolution commands. □

Example 7.3 (Water tank refinement via invariants). Alternatively we may use differential invariants with $\mathsf{d}\mathcal{R}$ to refine *tank-dinv* from Example 6.3. This time we supply a single structured proof to show another style of refinement. We abbreviate long expressions with schematic variables.

lemma *R-tank-inv*:
assumes $0 \leq \tau$ **and** $0 < c_o$ **and** $c_o < c_i$
shows $[I\ h_l\ h_h,\ I\ h_l\ h_h] \geq$ *tank-dinv* $c_i\ c_o\ h_l\ h_h\ \tau$
proof−
have $[I\ h_l\ h_h,\ I\ h_l\ h_h] \geq$
LOOP $((t ::= 0);[I\ h_l\ h_h \wedge t = 0,\ I\ h_l\ h_h])$ *INV I* $h_l\ h_h$ (**is** - $\geq$ *?R1*)
by (*refinement*, *rel-auto′*)
moreover have *?R1* $\geq$ *LOOP*
$((t ::= 0);(h_0 ::= h);[I\ h_l\ h_h \wedge t = 0 \wedge h_0 = h,\ I\ h_l\ h_h])$ *INV I* $h_l\ h_h$ (**is** - $\geq$ *?R2*)
by (*refinement*, *rel-auto′*)
moreover have *?R2* $\geq$
LOOP $(ctrl\ h_l\ h_h;[I\ h_l\ h_h \wedge t = 0 \wedge h_0 = h,\ I\ h_l\ h_h])$ *INV I* $h_l\ h_h$ (**is** - $\geq$ *?R3*)
by (*simp only*: *mult.assoc*, *refinement*; (*force*)*?*, (*rule R-assign-law*)*?*) *rel-auto′*
moreover have *?R3* $\geq$ *LOOP* $(ctrl\ h_l\ h_h;\ dyn\ c_i\ c_o\ h_l\ h_h\ \tau)$ *INV I* $h_l\ h_h$
apply(*simp only*: *mult.assoc*, *refinement*; (*simp*)*?*)
prefer *4* **using** *tank-diff-inv assms* **apply** *force+*
using *tank-inv-arith1 tank-inv-arith2 assms* **by** *rel-auto′*
ultimately show $[I\ h_l\ h_h,\ I\ h_l\ h_h] \geq$ *tank-dinv* $c_i\ c_o\ h_l\ h_h\ \tau$
by *auto*
qed

The proof incrementally refines the specification of *tank-dinv* using the laws of $\mathsf{d}\mathcal{R}$. As in Example 7.2, after refining the first two assignments, tactic *refinement* completes the construction of *ctrl*. After that, the invariant is supplied via lemma *tank-diff-inv* from Example 6.3 to construct *dyn*. The final program is then constructed by transitivity of $\leq$. A more detailed derivation is also possible. □

8 Evolution Commands for Flows

Finally, we present variants of $\mathsf{d}\mathcal{H}$ and $\mathsf{d}\mathcal{R}$ that start directly from flows $\varphi : T \to S \to S$ instead of vector fields. This avoids checking the conditions of the Picard-Lindelöf theorem and simplifies verification proofs considerably. Instead of $x' = f \,\&\, G$, we now use the command $\mathbf{evol}\,\varphi\,G$ in hybrid programs and define

$$(\mathbf{evol}\,\varphi\,G) = \lambda s.\ \gamma_G^{\varphi_s}\ s$$

with respect to the guarded orbit of φ_s along T in s. It then remains to derive a Hoare-style axiom and a refinement law for such evolution commands.

Lemma 8.1. *Let $\varphi : T \to S \to S$, where S is a set and T a preorder. Then, for $G, P, Q : S \to \mathbb{B}$,*

$$\{\lambda s \in S. \forall t \in T.\ (\forall \tau \in {\downarrow} t.\ G\,(\varphi_s\,\tau)) \to P\,(\varphi_s\,t)\}\,\mathbf{evol}\,\varphi\,G\,\{P\}, \qquad \text{(h-evlfl)}$$

$$\mathbf{evol}\,\varphi\,G \leq [\lambda s. \forall t \in T.\ (\forall \tau \in {\downarrow} t.\ G\,(\varphi_s\,\tau)) \to Q\,(\varphi_s\,t), Q], \qquad \text{(r-evlf)}$$

$$(\mathbf{evol}\,\varphi\,G) \cdot [Q, Q] \leq [\lambda s. \forall t \in T.\ (\forall \tau \in {\downarrow} t.\ G\,(\varphi_s\,\tau)) \to Q\,(\varphi_s\,t), Q], \qquad \text{(r-evlfl)}$$

$$[Q, \lambda s. \forall t \in T.\ (\forall \tau \in {\downarrow} t.\ G\,(\varphi_s\,\tau)) \to Q\,(\varphi_s\,t)] \cdot (\mathbf{evol}\,\varphi\,G) \leq [Q, Q]. \qquad \text{(r-evlfr)}$$

Example 8.2 (Bouncing ball via Hoare logic and refinement). A ball falls down from height $h \geq 0$, with x denoting its position, v its velocity and g its acceleration. Its kinematics is modelled by the flow

abbreviation $\varphi\ g\ \tau \equiv [x \mapsto_s g \cdot \tau\ \hat{}\ 2/2 + v \cdot \tau + x,\ \ v \mapsto_s g \cdot \tau + v]$

The ball bounces back elastically from the ground. This is modelled by a discrete control that checks for $x = 0$ and then flips the velocity. Guard $G = (x \geq 0)$ excludes any motion below the ground. This is modelled by the hybrid program [26]

abbreviation *bb-evol g h T* ≡
LOOP (*EVOL* (φ *g*) ($x \geq 0$) *T*; (*IF* ($x = 0$) *THEN* ($v ::= -v$) *ELSE skip*))
INV ($0 \leq x \wedge 2 \cdot g \cdot x = 2 \cdot g \cdot h + v \cdot v$)

Its loop invariant conjoins the guard G with a variant of energy conservation. The correctness specification and proof with $\mathsf{d}\mathcal{H}$ and $\mathsf{d}\mathcal{R}$ are then straightforward.

lemma *bouncing-ball-dyn*:
assumes $g < 0$ **and** $h \geq 0$
shows $\{x = h \wedge v = 0\}$ *bb-evol g h T* $\{0 \leq x \wedge x \leq h\}$
apply(*hyb-hoare* $\mathbf{U}(0 \leq x \wedge 2 \cdot g \cdot x = 2 \cdot g \cdot h + v \cdot v)$)
using *assms* **by** (*rel-auto′ simp*: *bb-real-arith*)

lemma *R-bouncing-ball-dyn*:
assumes $g < 0$ **and** $h \geq 0$
shows $[x = h \wedge v = 0,\ 0 \leq x \wedge x \leq h] \geq$ *bb-evol g h T*
apply(*refinement*; (*rule R-bb-assign*[*OF assms*])*?*)
using *assms* **by** (*rel-auto′ simp*: *bb-real-arith*)

In the refinement proof, the tactic leaves only the refinement for the assignment $v ::= -v$. This is supplied via lemma *R-bb-assign* and the remaining obligations are discharged with the same arithmetical facts. □

9 Conclusion

We have contributed new methods and Isabelle components to an open modular semantic framework for verifying hybrid systems that so far focussed on predicate transformer semantics [16]; more specifically a Hoare logic $\mathsf{d}\mathcal{H}$ and a Morgan-style refinement calculus $\mathsf{d}\mathcal{R}$ for hybrid programs, more generic state spaces modelled by lenses, improved Isabelle syntax for correctness specifications and hybrid programs, and increased proof automation via the tactics *hyb-hoare* and *refinement*. These components support three workflows based on certifying solutions to Lipschitz-continuous vector fields, reasoning with invariant sets for continuous vector fields, and working directly with flows without certification.

Compared to the standard $\mathsf{d}\mathcal{L}$ toolchain, $\mathsf{d}\mathcal{H}$ and $\mathsf{d}\mathcal{R}$ emphasise a natural mathematical style of semantic reasoning about dynamical systems, with minimal conceptual overhead relative to standard Hoare logics and refinement calculi. $\mathsf{d}\mathcal{H}$, in particular, is only used for automated verification condition generation. The modular approach with algebras and a shallow embedding has simplified the construction of these verification components and made it incremental relative to extant ones. Our framework is not only open to use any proof method and mathematical approach supported by Isabelle, it should also allow adding new methods, for instance based on discrete dynamical systems, hybrid automata or duration calculi [22,7], or integrate CAS's for finding solutions.

The relevance of $\mathsf{d}\mathcal{H}$ and $\mathsf{d}\mathcal{R}$ to hybrid systems verification is further evidenced by the fact that such approaches are not new: A hybrid Hoare logic has been proposed by Liu et al. [22] for a duration calculus based on hybrid CSP. It is conceptually very different from $\mathsf{d}\mathcal{H}$ and $\mathsf{d}\mathcal{L}$. A differential refinement logic based on $\mathsf{d}\mathcal{L}$ has been developed as part of Loos' PhD work [23]. It uses a proof system with inference rules for reasoning about inequalities between KAT expressions, which are interpreted as refinements between hybrid programs. According to the authors, it differs substantially from the standard approaches [4,25] in that local instead of global refinement relations can be used. Nevertheless their refinement logic has the same expressivity as $\mathsf{d}\mathcal{L}$ [23], which is essentially a predicate transformer calculus for hybrid programs [16] and thus a refinement calculus à la Back and von Wright. Ultimately, this suggests that Loos' logic is more expressive than our Morgan-style calculus, but the relative merits of the two approaches remain to be explored. The proof theory of $\mathsf{d}\mathcal{L}$ has already been deeply embedded in proof assistants [5], yet with a focus on soundness proofs for its inference rules and a mechanisation of its idiosyncratic substitution calculus, but not as prima facie verification components.

The expressivity and complexity gap between Hoare logic and predicate transformer semantics is apparent within algebra. The weakest liberal precondi-

tion operator cannot be expressed in KAT [27]. The equational theory of KAT, which captures propositional Hoare logic, is PSPACE complete [21], that of modal Kleene algebra, which yields predicate transformers, in EXPTIME [24].

Finally, while KAT and rKAT are convenient starting points for building program construction and verification components for hybrid programs, the simple and more general setting of Hoare semigroups [27] would support developing hybrid Hoare logics for total program correctness—where balls may bounce forever—or even for multirelational semantics [12,11], which are relevant needed for differential game logic [26]. This is left for future work.

Acknowledgements. The authors wish to thank the reviewers for their very thorough and insightful comments, and to Sergey Goncharov and André Platzer for discussions on a preliminary version. The second author is sponsored by CONACYT's scholarship no. 440404; the first author is supported by the EPSRC project CyPhyAssure[5] (Grant EP/S001190/1).

References

1. A. Angus and D. Kozen. Kleene algebra with tests and program schematology. Rechnical Report 2001-1844, Computer Science Department, Cornell University, 2001.
2. A. Armstrong, V. B. F. Gomes, and G. Struth. Kleene algebra with tests and demonic refinement algebras. *Archive of Formal Proofs*, 2014.
3. A. Armstrong, V. B. F. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2016.
4. R. Back and J. von Wright. *Refinement Calculus—A Systematic Introduction*. Springer, 1998.
5. B. Bohrer, V. Rahli, I. Vukotic, M. Völp, and A. Platzer. Formally verified differential dynamic logic. In *CPP 2017*, pages 208–221. ACM, 2017.
6. L. Doyen, G. Frehse, G. J. Pappas, and A. Platzer. Verification of hybrid systems. In *Handbook of Model Checking*, pages 1047–1110. Springer, 2018.
7. S. Foster. Hybrid relations in Isabelle/UTP. In *7th Intl. Symp. on Unifying Theories of Programming (UTP)*, volume 11885 of *LNCS*, pages 130–153. Springer, 2019.
8. S. Foster and F. Zeyda. Optics in Isabelle/HOL. *Archive of Formal Proofs*, 2018.
9. S. Foster, F. Zeyda, Y. Nemouchi, P. Ribeiro, and B. Wolff. Isabelle/UTP: Mechanised Theory Engineering for Unifying Theories of Programming. *Archive of Formal Proofs*, 2019.
10. S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *ICTAC 2016*, volume 9965 of *LNCS*, pages 295–314, 2016.
11. H. Furusawa and G. Struth. Binary multirelations. *Archive of Formal Proofs*, 2015.
12. H. Furusawa and G. Struth. Taming multirelations. *ACM TOCL*, 17(4):28:1–28:34, 2016.
13. V. B. F. Gomes and G. Struth. Program construction and verification components based on Kleene algebra. *Archive of Formal Proofs*, 2016.

[5] https://www.cs.york.ac.uk/circus/CyPhyAssure/

14. J. Hölzl, F. Immler, and B. Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In *ITP 2013*, volume 7998 of *LNCS*, pages 279–294. Springer, 2013.
15. J. J. Huerta y Munive. Verification components for hybrid systems. *Archive of Formal Proofs*, 2019.
16. J. J. Huerta y Munive and G. Struth. Predicate transformer semantics for hybrid systems: Verification components for Isabelle/HOL. arXiv:1909.05618 [cs.LO], 2019.
17. F. Immler and J. Hölzl. Ordinary differential equations. *Archive of Formal Proofs*, 2012.
18. F. Immler and C. Traut. The flow of ODEs: Formalization of variational equation and Poincaré map. *J. Automated Reasoning*, 62(2):215–236, 2019.
19. D. Kozen. Kleene algebra with tests. *ACM TOPLAS*, 19(3):427–443, 1997.
20. D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM TOCL*, 1(1):60–76, 2000.
21. D. Kozen, E. Cohen, and F. Smith. The complexity of Kleene algebra with tests. Technical Report TR96-1598, Computer Science Department, Cornell University, 1996.
22. J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou. A calculus for hybrid CSP. In *APLAS 2010*, volume 6461 of *LNCS*, pages 1–15. Springer, 2010.
23. S. M. Loos and A. Platzer. Differential refinement logic. In *LICS 2016*, pages 505–514. ACM, 2016.
24. B. Möller and G. Struth. Algebras of modal operators and partial correctness. *Theoretical Computer Science*, 351(2):221–239, 2006.
25. C. Morgan. *Programming from specifications, 2nd Edition*. Prentice Hall, 1994.
26. A. Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.
27. G. Struth. Hoare semigroups. *Mathematical Structures in Computer Science*, 28(6):775–799, 2018.
28. G. Teschl. *Ordinary Differential Equations and Dynamical Systems*. AMS, 2012.