Lince: Lightweight Prototyping of Hybrid Programs

Sergey Goncharov[1], Renato Neves[2], and José Proença[3]

[1] Friedrich-Alexander-Universität, Erlangen-Nürnberg, Germany,
sergey.goncharov@fau.de
[2] University of Minho & INESC-TEC, Braga, Portugal
{lsb,nevrenato}@di.uminho.pt
[3] CISTER-ISEP & INESC-TEC, Porto, Portugal,
pro@isep.ipp.pt

Abstract. Hybrid programs combine classical program constructs with differential equations, and thus naturally appear in a wide range of application domains, from biology and control theory to software engineering. This ability to entangle discrete and continuous behaviour, however, yields aspects unusual to computer science and renders the formal design of hybrid programs a difficult task, not properly handled by the current programming theory and practices.

As a stepping stone for closing this gap, here we develop the theoretical foundations for an interpreter of hybrid programs and present a corresponding implementation – Lince. These results serve not only as basis for the implementation of typical tools of programming (*e.g.* debuggers and refactoring systems) but also tools specific to the hybrid domain, such as the detection of chaotic or Zeno behaviour. We also summarise Lince's most distinctive features and illustrate its relevance for detecting design errors in hybrid programs at early development phases.

Keywords: Hybrid System, Program Semantics, Elgot Iteration

1 Introduction

1.1 Hybrid programming

Hybrid programming [32,30,38] is a rapidly emerging computational paradigm that aims at using principles and techniques from programming theory (*e.g.* compositionality [14,30] and Hoare calculus [32,38]) to engineer computational objects that closely interact with physical processes. Cruise controllers are a typical example of this pattern, illustrated by by the hybrid program below.

$$\texttt{while}\ \texttt{true}\ \texttt{do}\ \{\ \texttt{if}\ \texttt{v} \leq 10\ \texttt{then}\ (\texttt{v}' = 1\ \texttt{for}\ 1)\ \texttt{else}\ (\texttt{v}' = -1\ \texttt{for}\ 1)\ \}$$

In a nutshell, the program specifies a digital controller that periodically measures and regulates a vehicle's velocity (v): if the latter is less or equal than 10 the controller accelerates during 1 time unit, as dictated by the program statement $\texttt{v}' = 1\ \texttt{for}\ 1$ ($\texttt{v}' = 1$ is a differential equation representing the velocity's rate of

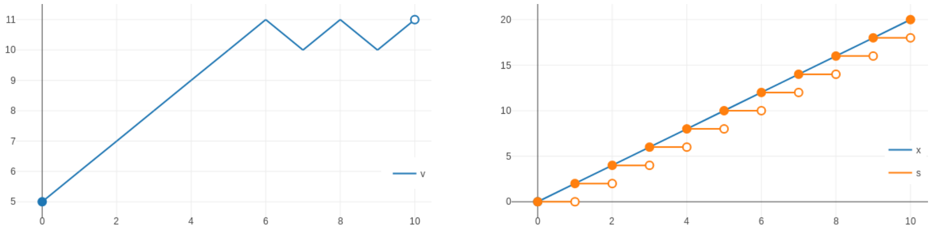


Fig. 1. Velocity's vehicle over time (on the left); Signal and its sample (on the right).

change over time). Otherwise, it decelerates during the same amount of time ($\mathtt{v}' = -1$ **for** 1). Figure 1 (left image) shows the output respective to this hybrid program for an initial velocity of 5.

In contrast to standard programming, the previous cruise controller involves not only classical constructs (namely while-loops and conditional statements) but also *differential* constructs, used for describing physical processes such as velocity, movement, energy, and time. This cross-disciplinary combination is the core feature of hybrid programming and has a notably wide range of application domains (see *e.g.* [32,33]; we will also see a sample of these throughout the paper). However, it also brings new challenges to the area of programming: issues range from the lack of suitable semantics to the lack of computational tools for the rigorous design and analysis of hybrid programs.

Lince. As a stepping stone for closing this gap, in this paper we develop the theoretical foundations and corresponding implementation of an *interpreter*—Lince–for a classical while-language extended with a notion of differential equation. Lince serves as a practical framework to *formally* specify and analyse hybrid programs, such as the cruise controller presented before and relevant properties regarding its behaviour: *e.g. what is the maximum velocity that the vehicle will attain? Does it ever reach an unsafe velocity? And if so for how long?* Moreover, Lince and its foundations are a solid basis for the implementation of programming tools oriented to the hybrid domain, such as refactoring systems, equivalence checking, and program verification tools.

1.2 Related work

We emphasise the distinction between a *hybrid system* and a *hybrid program*: the former are those systems whose behaviour has both discrete and continuous components [42,17]; whilst the latter are specifications of hybrid systems written in a programming-oriented style [32,30,38]. The cruise controller described before is a typical example of a hybrid program; *sampling algorithms* (used in signal processing) form another interesting class of examples. The following hybrid program describes a simplistic sampling algorithm.

$$\mathtt{x} \coloneqq 0 \,;\, \mathtt{s} \coloneqq 0 \,;\, \mathtt{while}\ \mathtt{true}\ \mathtt{do}\ \{\, (\mathtt{x}' = 2\ \mathtt{for}\ 1) \,;\, \mathtt{s} \coloneqq \mathtt{x} \,\}$$

The algorithm uses the variable s to sample, with a frequency of 1 per time unit, the signal represented by the differential equation $\mathtt{x}' = \mathtt{2}$. The trajectory produced by the sampling algorithm, depicted in Figure 1, was calculated by our tool. Other examples of hybrid systems can be typically found in the areas of control theory, impact-based mechanics, embedded software, and biology (including *e.g.* disease propagation [25] and personalised treatments against cancer [24]).

There are several formalisms of hybrid systems. Currently, the *de facto* one is *hybrid automata* [17]: briefly, an extension of classic automata with state variables that evolve continuously whilst in a state and change discretely at state transitions. Hybrid automata possess a rich theory and powerful tools, both typically focused on *reachability analysis* [10,34,18,3]. Closer to a programming-oriented style, and thus to our work, *differential dynamic logic* [32] is another well-known formalism. Its underlying action algebra is a *relational* Kleene algebra that interprets differential equations as the *set of points* that the respective solutions intersect. The logic possesses a semi-automated theorem prover—KeyMaera—and, as hybrid automata, focuses mainly on reachability analysis.

The tools Simulink [19] and Modelica [11] are the industrial standards for the design and analysis of hybrid systems, but lack a well-established, formal semantics. Following traditions of control theory, Simulink consists of a circuit-like, *block-based* language for describing hybrid systems, and moreover supports their simulation via numerical approximation methods. Modelica, on the other hand, is an *acausal*, specification language, and thus particularly useful for modelling purely physical systems such as electric circuits and the like which are traditionally modelled by systems of equations. There have been efforts on providing formal semantics to subsets of Simulink and Modelica *e.g.* [4,9], and avoiding simulation errors caused by numerical approximation methods *e.g.* [7,5].

1.3 Overview of Lince and contributions

Lince is an interpreter of hybrid programs. More specifically, a tool that, given a hybrid program and a time instant t, returns the value to which the program evaluates to at t. We regard this as a basic block not only for developing the usual tools of programming oriented to the hybrid setting but also for simulating hybrid systems' executions, in the spirit of Simulink and Modelica. To the best of our knowledge Lince is the first tool of its kind:

1. Its underlying language is the classical while-language extended with the notion of differential equation, thus promoting a compositional view and keeping in touch with classical programming theory. This contrasts with hybrid automata which are not amenable to a compositional approach [32, page 16] nor are they as close to a programming-oriented style.
2. Lince differs from the theorem prover KeyMaera in that it provides more lightweight verification methods at the cost of not being able to ensure that a program behaves as expected under all possible scenarios.[4]

[4] In principle, both tools could form a tool-chain: program properties would first be tested on Lince and only then validated in heavier, costlier tools, such as KeyMaera (a common practice in software verification).

Moreover, recall that the underlying semantics of KeyMaera interprets a differential equation as the set of points that the corresponding solution intersects *in lieu* of the solution (*i.e.* the trajectory) itself. This provides a more incisive focus on reachability properties, but renders more difficult to analyse certain aspects of trajectories: *e.g.* the total time during which a property holds or the velocity of approximation to a desired value [6,8]. Since we want our tool to be applicable to different domains we do interpret differential equations as the corresponding solutions.

3. Lince has simulation capabilities in the spirit of Simulink and Modelica, but unlike the latter it also has an established semantics. Actually, we will see below that one of the major challenges in our work was to ensure that the interpretation of hybrid programs at the implementation level coincided precisely with the intended semantics.

Semantics. In order to formally interpret hybrid programs, we furnish Lince's language with a small-step operational semantics that underlies Lince's interpretation process: given a hybrid program and a time instant the semantics tells how to convert, via a sequence of reduction steps, the given pair into a value—the latter representing the output of the program at the given time. Furthermore, we provide a compositional, *denotational* counterpart and a soundness/adequacy result linking both styles of semantics. The denotational semantics is based on a monad [28,29], which facilitates the extension of our language with new features such as non-deterministic, state-based, or probabilistic effects.

The challenge of finite precision. The fact that computers only support *finite-precision* numerical systems was a main challenge in our work:

1. In our operational semantics we could not treat solutions of differential equations as maps of the type $\phi\colon \mathbb{R}^n \times [0,\infty) \to \mathbb{R}^n$, due to the presence of real numbers which require infinite precision. Instead, the semantics uses *symbolic* representations of these solutions. Lince obtains such representations by invoking the free computer algebra tool SageMath [37]. Note that Lince is web-based, thus it does not require any installation from the user.[5]
2. In the interpretation of hybrid programs, if-then-else statements require testing whether certain conditions are true. However, here we could not test such conditions by merely invoking predicate functions – such as equality $(==)\colon \mathtt{Eq\,a} \Rightarrow \mathtt{a} \rightarrow \mathtt{a} \rightarrow \mathtt{Bool}$ – from a common programming language. Consider for example the equation $\mathbf{e^{-10^3} * e^{10^3} = 1}$, where e is Euler's number. Scala and JavaScript (which were used to implement Lince) wrongly evaluate this equation to *false*. Other programming languages suffer from the same problem, including Haskell and OCaml. This discrepancy causes highly spurious evaluations. Consider, for example, the program below.

$$\mathtt{x :=1\,;(x' = -x\ for\ 10^3)\,;(x' = x\ for\ 10^3)\,;if\ x = 1\ then\ p\ else\ q}$$

[5] https://github.com/arcalab/lince – it can also be downloaded, compiled, and executed locally (instructions in the website).

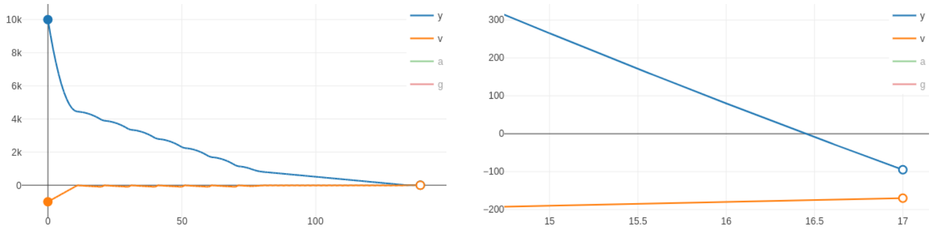


Fig. 2. Velocity and altitude of the spacecraft over time with an initial velocity of 3600km/h (plot on the left); spacecraft crashing with a velocity around 720km/h as a consequence of increasing the initial velocity to 4680km/h (plot on the right).

As explained later in the paper, $e^{-10^3} * e^{10^3} = 1$ will be tested in the evaluation of the conditional statement above. If the test wrongly yields false then the program `q` will execute instead of `p`. Note that increasing the precision of the underlying numerical system does not suffice, since increasing the durations of the differential statements would wrongly lead to the execution of program `q` instead of `p` again. In order to overcome this obstacle Lince, tests such conditions via SageMath which gives preference to algebraic laws rather than numerical approximations.

Illustration: Spacecraft landing. Let us illustrate Lince as lightweight tool for engineering hybrid programs, in particular to test early design choices and ideas. Consider the following scenario (described in [43]): a spacecraft is descending with high velocity into a planet, and we need to make it land smoothly by means of a vertical thrust vector. The spacecraft is inhabited by humans, so we need to restrict the maximum acceleration of the thrust vector to $100m/s^2$ (roughly, the acceleration force that a jet fighter pilot is subjected to). We assume that our task starts at $10km$ of altitude with an initial velocity of $3600km/h$ (*i.e.* $1000m/s$). For simplicity purposes, we consider a *constant* gravitation pull `g` and assume that the planet has no atmosphere (no frictions forces are involved). We then divide our task into three smaller ones in the form of while-loops (code available the tool's website). Each of these tasks corresponds to a different phase of the descent process, marked by the altitude of the spacecraft: the first task is active whilst at an altitude equal or above $1km$; the goal is to rapidly decrease the spacecraft's velocity by applying the maximum thrust force. The third task is active whilst at an altitude less or equal than $25m$; here the computer controls the descent much more precisely and uses much less thrust force. The second task serves as a transition between the two previous tasks, allowing to correct possible deviations from the starting altitude and velocity. The corresponding trajectory is presented in Figure 2 (on the left). A natural question to ask is whether our algorithm is robust against higher starting velocities. We can test this in Lince: changing the starting velocity to *e.g.* $4500km/h$, we can observe that the crew is still going to survive the descent. However, if we slightly increase the velocity up to *e.g.* $4680km/h$ then the spacecraft will crash at high speed.

Document structure. Section 2 introduces LINCE's while-language and its operational semantics. As mentioned before, this semantics is the core engine of hybrid program evaluation. Section 3 overviews LINCE's architecture, in particular its components and relations with one another. Then, Section 4 provides the aforementioned denotational counterpart to our operational semantics. As usual, the denotational semantics is compositional and abstracts from *how* the program is evaluated to focus on *what* values it outputs. Finally, Section 5 briefly discusses research lines opened up by our work.

2 A Hybrid While-Language

2.1 Syntax and semantics

We now introduce the syntax of LINCE's while-language. We fix a finite stock of variables $\mathcal{X}$, over which we build *atomic* programs according to the grammar:

$$\begin{aligned} \mathtt{At}(\mathcal{X}) \ni \mathtt{x} \coloneqq \mathtt{t} \mid \mathtt{x}_1' = \mathtt{t}, \dots, \mathtt{x}_n' = \mathtt{t} \;\mathtt{for}\; \mathtt{d} \\ \mathtt{t}, \mathtt{s} \ni \mathtt{r} \mid \mathtt{r} \cdot \mathtt{x} \mid \mathtt{t} + \mathtt{s} \end{aligned}$$

where $\mathtt{x}, \mathtt{x}_i \in \mathcal{X}$, $\mathtt{r} \in \mathbb{Q}$, and $\mathtt{d} \in [0, \infty) \cap \mathbb{Q}$. An atomic program is thus either a classical assignment $\mathtt{x} \coloneqq \mathtt{t}$ or a differential equation $\mathtt{x}_1' = \mathtt{t}, \dots, \mathtt{x}_n' = \mathtt{t} \;\mathtt{for}\; \mathtt{d}$; the latter reads as "*run the system of differential equations* $\mathtt{x}_1' = \mathtt{t}, \dots, \mathtt{x}_n' = \mathtt{t}$ *for* $\mathtt{d}$ *time units*". Next, we build our while-language via the grammar,

$$\mathtt{p}, \mathtt{q} \ni \mathtt{a} \mid \mathtt{p} \,;\, \mathtt{q} \mid \mathtt{if}\; \mathtt{b} \;\mathtt{then}\; \mathtt{p} \;\mathtt{else}\; \mathtt{q} \mid \mathtt{while}\; \mathtt{b} \;\mathtt{do}\; \{\, \mathtt{p} \,\}$$

where $\mathtt{a} \in \mathtt{At}(\mathcal{X})$ and $\mathtt{b}$ is an element of the free Boolean algebra generated by the terms $\mathtt{t} \leq \mathtt{s}$ and $\mathtt{t} \geq \mathtt{s}$.

Remark 1. *The systems of differential equations that the language allows are always linear. We could easily extend the language with more general systems, because its semantics (presented below) only requires that they have a solution. We refrain from doing this, however, because the usual examples of a hybrid program only require linear systems and deciding which classes of differential equation to allow would distract us from building the core features of* LINCE.

Notation. We abbreviate differential statements $\mathtt{x}_1' = \mathtt{t}_1, \dots, \mathtt{x}_n' = \mathtt{t}_n \;\mathtt{for}\; \mathtt{d}$ to the expression $\overline{\mathtt{x}}' = \overline{\mathtt{t}} \;\mathtt{for}\; \mathtt{d}$, where $\overline{\mathtt{x}}'$ and $\overline{\mathtt{t}}$ abbreviate the corresponding vectors of variables $\mathtt{x}_1{}' \dots \mathtt{x}_n{}'$ and terms $\mathtt{t}_1 \dots \mathtt{t}_n$.

The cruise controller, sampling algorithm, and spacecraft that were presented before are examples of programs written in this while-language. Let us briefly analyse another class of examples.

Example 2 (Real-Time Computation). Hybrid programs are closely related to real-time computation [21]; *e.g.*, one can introduce *wait calls* via the program:

$$\mathtt{x}_1' = 0, \dots, \mathtt{x}_n' = 0 \;\mathtt{for}\; \mathtt{d}$$

meaning that the computer's execution will halt for $\mathtt{d}$ time units (we abbreviate this instruction by the expression $\mathtt{wait}\; \mathtt{d}$). This allows to consider *oscillators* which are present in *e.g.* musical synthesisers and traffic lights [12]. The hybrid program below is a simple example of an oscillator.

$$\mathtt{a} \coloneqq 1\,;\ \mathtt{while}\ \mathtt{true}\ \mathtt{do}\ \{\ \mathtt{a} \coloneqq -\mathtt{a}\,;(\mathtt{wait}\ 1)\ \}$$

We will now introduce the aforementioned small-step, operational semantics for our while-language. Intuitively, the semantics establishes a set of rules for reducing triples ⟨program statement, variables' values, time instant⟩ to values, via a sequence of reduction steps. This semantics is inspired by the *big-step*, operational semantics presented in [15], but with the following crucial differences:

1. Our semantics does not involve rules with infinitely many premises, which would render a computational implementation impossible to achieve.
2. Moreover, our semantics does not require calculating the total duration of programs: this would yield infinite reduction sequences in the presence of infinite while-loops and thus would hinder the computational evaluation of such programs.
3. In order to be able to *exactly* evaluate hybrid programs, our semantics does not interpret solutions of systems of differential equations paired with initial values σ as functions of the type $\phi_\sigma \colon [0,\infty) \to \mathbb{R}^n$. Instead, it interprets solutions as n-tuples $\bar{\mathtt{u}}$ of terms built from the grammar,

$$\mathtt{u},\mathtt{v} \ni \mathtt{r} \mid t \mid \mathit{sin}\,(\mathtt{u}) \mid \mathit{cos}\,(\mathtt{u}) \mid \mathtt{u}\cdot\mathtt{v} \mid \mathtt{u}+\mathtt{v} \mid e^{\mathtt{u}} \mid -\mathtt{u} \mid \sqrt{\mathtt{u}}$$

where $\mathtt{r} \in \mathbb{Q}$ and t is what we call a time variable. For example, the solution of $\mathtt{x}' = \mathtt{x}$ with $\mathtt{x} = 2$ as the initial condition is $\phi_{\mathtt{x}\mapsto 2} = 2 \cdot e^t$.

Remark 3. *Suenaga and Hasuo [38] also developed a while-language for hybrid systems, but differently from us their approach is based on a constant that represents an* infinitesimal. *In particular, their semantics is defined via* nonstandard analysis *and is therefore not amenable to implementation.*

Next, we denote the set of closed terms $\mathtt{u}$ (*i.e.* with no variable t) by $\mathtt{CTerm}$ and call functions of type $\sigma\colon \mathcal{X} \to \mathtt{CTerm}$ *environments*; these functions map program variables to their respective closed terms, the latter being symbolic representations of real numbers. For every linear term $\mathtt{t}$ (defined in the beginning of the section), we use the notation $\mathtt{t}\sigma$ to refer to the closed term obtained by substituting the free variables in $\mathtt{t}$ by the respective closed terms according to σ. We use the notation $\sigma\nabla[\bar{\mathtt{u}}/\bar{\mathtt{x}}]$ to denote the environment that maps each $\mathtt{x_i}$ in $\bar{\mathtt{x}}$ to $\mathtt{u_i}$ and the rest of variables in the same way as σ. Finally, we denote by ϕ_σ the solution of a system of differential equations with σ as the initial condition, and subsequently denote by $\phi_\sigma(r)$ the expression that results from substituting t by r in ϕ_σ where r is a non-negative rational number.

The small-step rules are presented in Figure 3. Contrary to standard operational semantics, they are not defined on program statements and environments but on triples comprised of a program statement, an environment, and a time instant. The terminal configuration skip, σ, t represents a successful end of a computation, which can then be fed into another computation (via rule (**seq-skip**$^{\to}$)). On the other hand, stop, σ, t represents an *abnormally* terminating computation, which cannot be composed with a following one. The latter case is reflected in rules **(diff-stop$^{\rightarrow}$)** and **(seq-stop$^{\rightarrow}$)** which tell that, depending on the chosen time instant, we do not need to evaluate the whole program, but merely a part of it – consequently, infinite while-loops do not necessarily yield infinite reduction sequences (as explained in Remark 4). The rules **(seq)** and **(seq-skip$^{\rightarrow}$)** correspond to the the standard rules of operational semantics for while languages over an imperative store [41].

Let $\rightarrow^{\star}$ be the transitive closure of the reduction relation $\rightarrow$ that was previously presented. Then we say that a triple $\mathtt{p}\,,t\,,\sigma$ evaluates into a value σ' iff $\mathtt{p}\,,t\,,\sigma \;\rightarrow^{\star}\; s\,,\sigma'\,,0$ where s is either *skip* or *stop*.

Remark 4. *Infinite while-loops do not necessarily yield infinite reduction steps: take for example the while-loop*

$$\mathtt{x} \coloneqq \mathtt{0}\,;\mathtt{while}\;\mathtt{true}\;\mathtt{do}\;\{\;\mathtt{x} \coloneqq \mathtt{x} + \mathtt{1}\;;\;\mathtt{wait}\;\mathtt{1}\;\} \tag{1}$$

whose iterations always have duration 1*. It yields a finite reduction sequence for the time instant* $\frac{1}{2}$*, as shown by the following calculation:*

$$\mathtt{x} \coloneqq \mathtt{0}\,;\mathtt{while}\;\mathtt{true}\;\mathtt{do}\;\{\;\mathtt{x} \coloneqq \mathtt{x} + \mathtt{1}\;;\;\mathtt{wait}\;\mathtt{1}\;\}\,,\sigma\,,\tfrac{1}{2} \rightarrow$$

$\{$*by the rules* ***(asg$^{\rightarrow}$)*** *and* ***(seq-skip$^{\rightarrow}$)***$\}$

$$\mathtt{while}\;\mathtt{true}\;\mathtt{do}\;\{\;\mathtt{x} \coloneqq \mathtt{x} + \mathtt{1}\;;\;\mathtt{wait}\;\mathtt{1}\;\}\,,\sigma\nabla[\mathtt{0}/\mathtt{x}]\,,\tfrac{1}{2} \rightarrow$$

$\{$*by the rule* ***(wh-true$^{\rightarrow}$)***$\}$

$$\mathtt{x} \coloneqq \mathtt{x} + \mathtt{1}\;;\;\mathtt{wait}\;\mathtt{1}\,;\mathtt{while}\;\mathtt{true}\;\mathtt{do}\;\{\;\mathtt{x} \coloneqq \mathtt{x} + \mathtt{1}\;;\;\mathtt{wait}\;\mathtt{1}\;\}\,,\sigma\nabla[\mathtt{0}/\mathtt{x}]\,,\tfrac{1}{2} \rightarrow$$

$\{$*by the rules* ***(asg$^{\rightarrow}$)*** *and* ***(seq-skip$^{\rightarrow}$)***$\}$

$$\mathtt{wait}\;\mathtt{1}\,;\mathtt{while}\;\mathtt{true}\;\mathtt{do}\;\{\;\mathtt{x} \coloneqq \mathtt{x} + \mathtt{1}\;;\;\mathtt{wait}\;\mathtt{1}\;\}\,,\sigma\nabla[\mathtt{0} + \mathtt{1}/\mathtt{x}]\,,\tfrac{1}{2} \rightarrow$$

$\{$*by the rules* ***(diff-stop$^{\rightarrow}$)*** *and* ***(seq-stop$^{\rightarrow}$)***$\}$

$$\mathit{stop}\,,\sigma\nabla[\mathtt{0} + \mathtt{1}/\mathtt{x}]\,,0$$

The gist here is that to evaluate the program (1) *at time instant* $\frac{1}{2}$ *we only need to unfold the infinite while-loop once, because the time instant* $\frac{1}{2}$ *occurs* before *the end of the first iteration. On the other hand, if the wait statement is removed from the program then the reduction of the resulting program would not terminate, intuitively because all iterations would be instantaneous and thus no finite number of unfoldings of the loop would have duration greater than* $\frac{1}{2}$*.*

The following theorem entails that our semantics is deterministic; moreover, the corresponding proof details the algorithm behind Lince for deciding which rule to apply in a reduction step.

Theorem 5. *For every program* $\mathtt{p}$*, environment* σ*, and time instant* t *there is* at most one *applicable reduction rule.*

Proof. The proof follows by inspecting the structure of program terms: first, for atomic programs the proof follows directly, because the corresponding premises are mutually exclusive. For conditionals, the proof also follows directly due to the same reason. For sequential composition $\mathtt{p} \mathbin{;} \mathtt{q}$, we need to proceed by case distinction: if $\mathtt{p}$ is atomic then the only applicable rules are **(seq-stop$^{\rightarrow}$)** and **(seq-skip$^{\rightarrow}$)** but then it is easy to see that the corresponding premises are mutually exclusive. If $\mathtt{p}$ is non-atomic then the only applicable rules are **(seq$^{\rightarrow}$)** and **(seq-skip$^{\rightarrow}$)**. But in this context, the application of **(seq-skip$^{\rightarrow}$)** requires that $\mathtt{p}$ is a while-loop with $\mathtt{b}\sigma = \bot$ which forbids the application of **(seq$^{\rightarrow}$)**. Conversely, the application of **(seq$^{\rightarrow}$)** requires that $\mathtt{p}$ is not a while-loop with $\mathtt{b}\sigma = \bot$ and thus we cannot apply **(seq-skip$^{\rightarrow}$)**. The proof for while-loops is direct because the relevant premises are mutually disjoint. □

Corollary 6. *For every program term* $\mathtt{p}$*, environments* σ*,* σ'*,* σ''*, time instants* t*,* t'*,* t''*, and termination flags* $s, s' \in \{\mathit{skip}, \mathit{stop}\}$*, if* $\mathtt{p}, \sigma, t \;\rightarrow^{\star}\; s, \sigma', t'$ *and* $\mathtt{p}$*,* $\sigma, t \;\rightarrow^{\star}\; s', \sigma'', t''$*, then the equations* $s = s'$*,* $\sigma' = \sigma''$ *and* $t' = t''$ *must hold.*

Proof. Follows by induction on the number of reduction steps and Theorem 5. □

Note that operational semantics treats time as a resource formalised below.

Proposition 7. *For all program terms* $\mathtt{p}$ *and* $\mathtt{q}$*, environments* σ *and* σ'*, and time instants* t*,* t' *and* s*, if* $\mathtt{p}, \sigma, t \;\rightarrow\; \mathtt{q}, \sigma', t'$ *then* $\mathtt{p}, \sigma, t + s \;\rightarrow\; \mathtt{q}, \sigma'$*,* $t' + s$*; and if* $\mathtt{p}, \sigma, t \;\rightarrow\; \mathit{skip}, \sigma', t'$ *then* $\mathtt{p}, \sigma, t + s \;\rightarrow\; \mathit{skip}, \sigma', t' + s$*.*

Note that Proposition 7 does not apply to *stop*, however, it is easy to see that $\mathtt{p}$, $\sigma, t \;\rightarrow\; \mathit{stop}, \sigma', 0$ implies $\mathtt{p}, \sigma, s \;\rightarrow\; \mathit{stop}, \sigma', 0$ for any $s < t$.

2.2 Event-triggered programs

The differential statements $\mathtt{x}_1' = \mathtt{t}, \ldots, \mathtt{x}_n' = \mathtt{t}\;\mathtt{for}\;\mathtt{d}$ are time-triggered: *i.e.* they terminate precisely when the instant of time $\mathtt{d}$ is achieved. In the area of hybrid systems, it is also usual to consider *event-triggered* programs: *i.e.* programs that terminate *as soon as* a specified condition ψ becomes true [42,7,13]. We thus next consider atomic programs of this type,

$$\mathtt{x}_1' = \mathtt{t}, \ldots, \mathtt{x}_n' = \mathtt{t}\;\mathtt{until}\;\psi$$

where ψ is an element of the free Boolean algebra generated by $\mathtt{t} \leq \mathtt{s}$ and $\mathtt{t} \geq \mathtt{s}$. We will show that it is generally very hard to provide a semantics for these programs such that it can be *precisely* implemented; take the program,

$$\mathtt{x} \coloneqq \mathtt{0} \mathbin{;} \mathtt{y} \coloneqq \mathtt{1} \mathbin{;} \mathtt{z} \coloneqq \mathtt{0} \mathbin{;} (\mathtt{x}' = \mathtt{y}, \mathtt{y}' = -\mathtt{x}, \mathtt{z}' = \mathtt{1} + \mathtt{y}\;\mathtt{until}\;\mathtt{z} = \mathtt{1})$$

Consider the system $\mathtt{x}' = \mathtt{y}, \mathtt{y}' = -\mathtt{x}$ with $\mathtt{x} = \mathtt{0}, \mathtt{y} = \mathtt{1}$ as the initial condition. Then, the solution for the variable $\mathtt{y}$ is the function $t \mapsto \sin(t)$. Hence, by linearity, the solution of the differential system $\mathtt{x}' = \mathtt{y}, \mathtt{y}' = -\mathtt{x}, \mathtt{z}' = \mathtt{1} + \mathtt{y}$ and initial condition $\mathtt{x} = \mathtt{0}, \mathtt{y} = \mathtt{1}, \mathtt{z} = \mathtt{0}$ for the variable $\mathtt{z}$ is $t \mapsto t + \sin(t)$. Consequently, to determine the program's total duration we need to solve the equation $t + \sin(t) = 1$, but there is no known method in the literature that provides the exact solution for this.

$$\textbf{(asg}^{\rightarrow}\textbf{)} \qquad \mathtt{x} \coloneqq \mathtt{t}\,, \sigma\,, t \;\rightarrow\; \mathit{skip}\,, \sigma\nabla[\mathtt{t}\sigma/\mathtt{x}]\,, t$$

$$\textbf{(diff-stop}^{\rightarrow}\textbf{)} \qquad \bar{\mathtt{x}}' = \bar{\mathtt{u}}\;\mathtt{for}\;\mathtt{d}\,, \sigma\,, t \;\rightarrow\; \mathit{stop}\,, \sigma\nabla[\phi_\sigma(t)/\bar{\mathtt{x}}]\,, 0 \qquad (\mathit{if}\;\; t < \mathtt{d})$$

$$\textbf{(diff-skip}^{\rightarrow}\textbf{)} \qquad \bar{\mathtt{x}}' = \bar{\mathtt{u}}\;\mathtt{for}\;\mathtt{d}\,, \sigma\,, t \;\rightarrow\; \mathit{skip}\,, \sigma\nabla[\phi_\sigma(\mathtt{d})/\bar{\mathtt{x}}]\,, t - \mathtt{d} \qquad (\mathit{if}\;\; t \geq \mathtt{d})$$

$$\textbf{(if-true}^{\rightarrow}\textbf{)} \qquad \mathtt{if}\;\mathtt{b}\;\mathtt{then}\;\mathtt{p}\;\mathtt{else}\;\mathtt{q}\,, \sigma\,, t \;\rightarrow\; \mathtt{p}\,, \sigma\,, t \qquad (\mathit{if}\;\; \mathtt{b}\sigma = \top)$$

$$\textbf{(if-false}^{\rightarrow}\textbf{)} \qquad \mathtt{if}\;\mathtt{b}\;\mathtt{then}\;\mathtt{p}\;\mathtt{else}\;\mathtt{q}\,, \sigma\,, t \;\rightarrow\; \mathtt{q}\,, \sigma\,, t \qquad (\mathit{if}\;\; \mathtt{b}\sigma = \bot)$$

$$\textbf{(wh-true}^{\rightarrow}\textbf{)} \qquad \mathtt{while}\;\mathtt{b}\;\mathtt{do}\;\{\,\mathtt{p}\,\}\,, \sigma\,, t \;\rightarrow\; \mathtt{p}\,\mathtt{;}\,\mathtt{while}\;\mathtt{b}\;\mathtt{do}\;\{\,\mathtt{p}\,\}\,, \sigma\,, t \qquad (\mathit{if}\;\; \mathtt{b}\sigma = \top)$$

$$\textbf{(wh-false}^{\rightarrow}\textbf{)} \qquad \mathtt{while}\;\mathtt{b}\;\mathtt{do}\;\{\,\mathtt{p}\,\}\,, \sigma\,, t \;\rightarrow\; \mathit{skip}\,, \sigma\,, t \qquad (\mathit{if}\;\; \mathtt{b}\sigma = \bot)$$

$$\textbf{(seq-stop}^{\rightarrow}\textbf{)} \quad \frac{\mathtt{p}\,, \sigma\,, t \;\rightarrow\; \mathit{stop}\,, \sigma'\,, t'}{\mathtt{p}\,\mathtt{;}\,\mathtt{q}\,, \sigma\,, t \;\rightarrow\; \mathit{stop}\,, \sigma'\,, t'} \qquad\qquad \textbf{(seq-skip}^{\rightarrow}\textbf{)} \quad \frac{\mathtt{p}\,, \sigma\,, t \;\rightarrow\; \mathit{skip}\,, \sigma'\,, t'}{\mathtt{p}\,\mathtt{;}\,\mathtt{q}\,, \sigma\,, t \;\rightarrow\; \mathtt{q}\,, \sigma'\,, t'}$$

$$\textbf{(seq}^{\rightarrow}\textbf{)} \quad \frac{\mathtt{p}\,, \sigma\,, t \;\rightarrow\; \mathtt{p}'\,, \sigma'\,, t'}{\mathtt{p}\,\mathtt{;}\,\mathtt{q}\,, \sigma\,, t \;\rightarrow\; \mathtt{p}'\mathtt{;}\,\mathtt{q}\,, \sigma'\,, t'}$$

Fig. 3. Small-step Operational Semantics

The simulation engines behind Simulink and Modelica tackle this problem by checking the condition ψ periodically, which essentially reduces event-triggered programs into time-triggered ones. The cost is that the simulation of a specification might greatly diverge from the nominal behaviour, as discussed for instance in documents [5,7]. Here we also follow the strategy of periodically checking the condition ψ, but avoid the aforementioned cost in the following way: rather than simply allowing event-triggered programs, we allow programs of the form,

$$\mathtt{x}_1' = \mathtt{t}, \ldots, \mathtt{x}_\mathtt{n}' = \mathtt{t}\;\mathtt{until}_\epsilon\;\psi$$

where ϵ is a non-negative rational number. In words, we explicitly write the period for checking the condition ψ in the specification rather than hiding it in the evaluation engine of our tool. A program of this form then abbreviates the while-loop,

$$\mathtt{while}\;\neg\psi\;\mathtt{do}\;\{\,\mathtt{x}_1' = \mathtt{t}, \ldots, \mathtt{x}_\mathtt{n}' = \mathtt{t}\;\mathtt{for}\;\epsilon\,\}$$

which, as discussed before, Lince can *precisely* evaluate.

Remark 8. *A limitation of our approach is that explicitly writing the period* ϵ *for checking* ψ *might distract the programmer from what he or she wishes to*

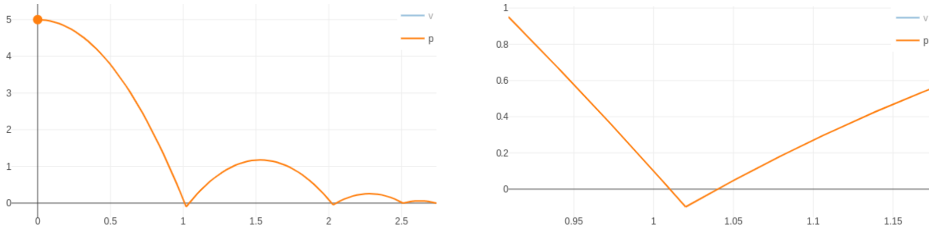


Fig. 4. Position of the bouncing ball over time (plot on the left); zoomed in position of the bouncing ball at the first bounce (plot on the right).

analyse. One interesting way of overcoming this issue is to study methods for automatically *calculating suitable values for ϵ.*

Example 9 (Bouncing Ball). In order to illustrate our approach to event-triggered programs, consider a bouncing ball dropped at a positive height `p` and with no initial velocity `v`. Due to the gravitational acceleration `g`, it falls into the ground and then bounces back up, losing part of its kinetic energy in the process. This can be approximated by the following hybrid program.

$$(\mathtt{p}' = \mathtt{v}, \mathtt{v}' = \mathtt{g}\ \mathtt{until}_{0.01}\ \mathtt{p} \leq 0 \land \mathtt{v} \leq 0)\,;(\mathtt{v} := \mathtt{v} \times -0.5)$$

where 0.5 is the dampening factor of the ball. We now want to drop the ball from a specific height (*e.g.* 5 meters) and let it bounce until it stops. Abbreviating the previous program into `b`, this behaviour can be approximated by,

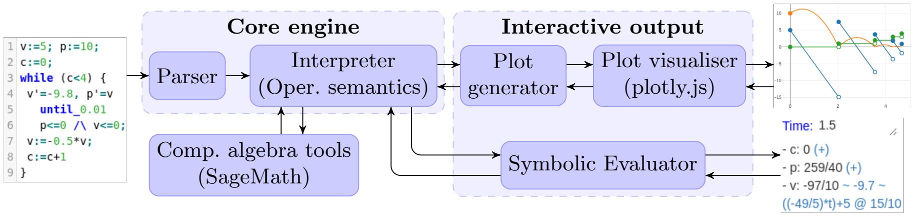
$$\mathtt{p} := 5\,;\mathtt{v} := 0\,;\mathtt{while}\ \mathtt{true}\ \mathtt{do}\ \{\ \mathtt{b}\ \}$$

Figure 4 on the left presents the trajectory generated by the bouncing ball (calculated by our tool Lince). Note that since $\epsilon = 0.01$ the bouncing ball reaches below ground, as shown in Figure 4 on the right.

Other examples of a event-triggered program can be seen in Lince's website.

3 Architecture

Lince's architecture is depicted in Figure 5; it uses an example of a bouncing ball to illustrate user interactions. The dashed rectangles correspond to the two main components of Lince: the one on the left parses and evaluates hybrid programs according to the previously presented operational semantics (*Core engine*); whilst the one on the right depicts the trajectories produced by hybrid programs according to parameters specified by the user (*Interactive output*). Incoming arrows in the figure denote an input relation and analogously outgoing arrows denote an output relation. The components in Figure 5 are explained in detail below.



```
v:=5; p:=10;
c:=0;
while (c<4) {
 v'=-9.8, p'=v
   until_0.01
   p<=0 /\ v<=0;
 v:=-0.5*v;
 c:=c+1
}
```

Core engine
Parser
Interpreter (Oper. semantics)
Comp. algebra tools (SageMath)
Interactive output
Plot generator
Plot visualiser (plotly.js)
Symbolic Evaluator
10
5
0
-5
-10
-15
0
2
4

```
Time: 1.5
- c: 0 (+)
- p: 259/40 (+)
- v: -97/10 ~ -9.7 ~
((-49/5)*t)+5 @ 15/10
```

Fig. 5. Depiction of Lince's architecture

User interaction. The user interacts with Lince at two different stages: (a) when inputting a hybrid program for analysis and (b) when interacting with Lince's output interfaces. The latter case consists of adjusting different parameters for observing the generated plots in an optimal way, and selecting a time instant for precisely evaluating the program at hand. Plot parameters include the time range of observation, visibility of variable's trajectories, and options to display additional information about the trajectory (more details below).

Core engine. Lince's parser and interpreter were developed in Scala, an object-oriented programming language with functional features [31]. We chose Scala because (1) it is fully interoperable with Java and its many existing libraries, (2) it has a functional core that facilitates code maintenance, and (3) it can be directly translated into JavaScript using ScalaJS,[6] which facilitates the use of web technologies.

The interpreter implements the small-step operational semantics introduced before (Figure 3). A key feature of our implementation is that it extensively uses the computer algebra tool SageMath [37]. This has two different purposes: first, to solve systems of differential equations present in hybrid programs; and second to correctly evaluate if-then-else statements, as discussed before. Regarding the latter, recall from the introduction that we cannot simply use Scala's predicate functions for evaluating Boolean conditions, because such functions tend to give wrong results in the presence of real numbers (due to the finite precision problem). Instead, Lince uses SageMath, and its ability to perform advanced symbolic manipulation, to check whether a Boolean condition is true or not. This approach prioritises the correct evaluation of trajectories over the tool's performance.

Interactive output. Plots in Lince are generated by repeatedly asking the interpreter to evaluate a given hybrid program at different time instants. These evaluations are then fed to the plot generation tool, which generates a plot from this data set. The plot generator actually provides more information than just the values obtained from the interpreter: it also tells (1) the boundary times of differential constructs and (2) where in time conditional statements are evaluated (together with their underlying Boolean condition)

[6] https://www.scala-js.org

Users can interact with the plot visualiser in different ways. They can (1) adjust the time range for observation; (2) increase the data set obtained from the interpreter (button "resample"), (3) toggle the highlighting of the termination of differential constructs and the evaluation of conditional statements (button "show/hide jumps"), (4) and toggle the visibility of variable's trajectories (by clicking on the respective label). These features are handled by a JavaScript library Plotly.js,[7] which interacts with our plot generator.

4 Denotational Semantics and Adequacy

We now complement our operational semantics with a denotational semantics by resorting to Moggi's *computational monads* [28,29]. Specifically, we base on a variant of the writer monad [40] to provide a denotational semantics to our while-language. We start by briefly recalling the definition of a monad.

Definition 10 (Monad). A monad $(T, \eta, (-)^\star)$ on a category $\mathbf{C}$ is an endomap T on $|\mathbf{C}|$, together with a $|\mathbf{C}|$-indexed class of morphisms $\eta_X \colon X \to TX$ and a so-called *Kleisli lifting* sending each $f \colon X \to TY$ to $f^\star \colon TX \to TY$ and obeying *monad laws*: $\eta^\star = \mathrm{id}$, $f^\star \eta = f$, $(f^\star g)^\star = f^\star g^\star$ (it follows from this definition that T extends to a functor and η to a natural transformation).

We will mostly work in the category $\mathbf{Set}$ of sets and functions.

Definition 11 (Generalised Writer Monad). Given a monoid $(M, \cdot, 1)$ in a category $\mathbf{C}$, a *monoid module* is a $\mathbf{C}$-object E equipped with a morphism $\bullet \colon M \times E \to E$ that obeys the laws $1 \bullet e = e$ and $(m \cdot n) \bullet e = m \bullet (n \bullet e)$. Every monoid-module pair (M, E) induces a monad, which we call the *generalised writer monad*. Concretely, on $\mathbf{Set}$, we obtain: $T = M \times (-) \cup E$, the unit is defined as $\eta_X(x) = (1, x)$ and Kleisli lifting is defined as follows:

$$\begin{aligned}
f^\star(x) &= (m \cdot n, z) &\quad& \text{if} \quad f(x) = (m, y),\ f(y) = (n, z)\\
f^\star(x) &= m \bullet e && \text{if} \quad f(x) = (m, y),\ f(y) = e\\
f^\star(x) &= e && \text{if} \quad f(x) = e
\end{aligned}$$

This yields a joint generalisation of the writer monad ($E = \emptyset$) and the exception monad ($M = 1$).

Example 12 (Duration Monad [15]). By taking $M = [0, \infty)$ and $E = [0, \infty]$ in Definition 11, we obtain what we call the *duration monad*. Intuitively, the carrier $TX = [0, \infty) \times X \cup [0, \infty]$ keeps track of durations of computations and their terminal values if they terminate – if they do not terminate then there is no terminal value and the respective duration is allowed to be infinite.

In order to give a semantics to while-loops we will also need the following notion.

[7] https://plot.ly/javascript/

Definition 13 (Elgot Monad). A monad $(T, \eta, (-)^\star)$ on a category $\mathbf{C}$ with coproducts is called *Elgot* if it is equipped with an iteration operator $(-)^\dagger$ that sends each morphism $f\colon X \to T(Y + X)$ to a specified morphism $f^\dagger\colon X \to TY$ in such a way that certain coherence conditions are satisfied [2,16].

Using general results [39], we will start by building a monad with a partially defined iteration operator. Then we will quotient this monad suitably to obtain the desired monad, and to which the partial iteration operator will transfer along the quotient morphism. Intuitively, this quotiening procedure builds an extensional semantics from an intensional one [1]: whilst the former monad distinguishes programs such as `wait 1 ; wait 1` and `wait 2` (intensionality), the latter monad regards them as equal (extensionality). In other words, the quotiented monad abstracts from the possible ways a specific program can be built via sequential composition. The strategy just described is inspired by [15], where an analogous procedure was used for providing a *duration semantics* to a while-language.

Let us fix a set S, which we regard as a set of possible contents of a global store. We will later instantiate S with the function space of environments $\mathcal{X} \to$ `CTerm`, but for the time being a specific choice of S is not relevant.

We introduce the following instance of the generalised writer monad in $\mathbf{Set}$:

$$\hat{H}X = \Big(\sum\nolimits_{r\in[0,\infty)} S^{[0,r)}\Big)^{*} \times X \cup \Big(\sum\nolimits_{r\in[0,\infty)} S^{[0,r)}\Big)^{\omega}$$

This involves sets $S^{[0,r)}$ of *trajectories* valued in S, in particular we identify the empty function $!: [0,0) \to S$ as the *empty trajectory.* An element of $\hat{H}X$ is thus either a finite sequence of trajectories followed by an element of X or an infinite sequence of trajectories. The involved monoid is simply the free monoid over $M = \sum_{r\in[0,\infty)} S^{[0,r)}$ and M^ω is the corresponding monoid module under concatenation with finite words.

Next, by $\langle x_1, \ldots, x_n\rangle$ we will denote a finite list of elements $x_1, \ldots, x_n$ and by $\langle x_1, \ldots, x_n, \ldots\rangle$, an infinite list of elements $x_1, \ldots, x_n, \ldots$ By $u \mathbin{+\!\!+} w$ we will denote list concatenation (equal to u if u is infinite).

Definition 14 (Guardedness). A morphism $f\colon X \to \hat{H}(Y + X)$ is *guarded* if $f(x) = (w, \mathsf{inr}\, x')$ implies that w is a non-empty list.

Proposition 15. *For every guarded map $f\colon X \to \hat{H}(Y + X)$, there is a* unique *function $f^\dagger\colon X \to \hat{H}Y$ that satisfies the fixpoint equation $f^\dagger = [\eta, f^\dagger]^\star f$.*

Proof. We use the fact that $\hat{H}X$ is isomorphic to the following final coalgebra:

$$\nu\gamma.\,\Big(X + \sum\nolimits_{r\in[0,\infty)} S^{[0,r)} \times \gamma\Big)$$

and that the monad structure on $\hat{H}$ is canonically induced by the final coalgebra structure. In this form the desired statement is shown more generally in [39]. □

Note that the map $f\colon X \to \hat{H}(Y+X)$ sending x to $(\epsilon, \mathsf{inr}\,x)$, where ϵ is the empty word and $\mathsf{inr} : X \to Y + X$ is the right injection, is not guarded. Hence, the iteration operator of $\hat{H}$ is properly partial. By general results [39], $\hat{H}$ satisfies the laws of iteration and is thus a *guarded Elgot* monad [22]. The next step is to quotient the monad $(\hat{H}, \eta, (-)^\star)$ to enforce extensionality. To that end, we will resort to the operation,

$$^\frown : \sum\nolimits_{r\in[0,\infty)} S^{[0,r)} \times \sum\nolimits_{r\in[0,\infty]} S^{[0,r)} \to \sum\nolimits_{r\in[0,\infty]} S^{[0,r)}$$

for concatenating trajectories: given $f\colon [0,r) \to S$ and $g\colon [0,s) \to S$ with finite r, $f ^\frown g\colon [0, r+s) \to S$ is defined as follows:

$$t \mapsto \begin{cases} f(t) & \text{if } t < r \\ g(t-r) & \text{otherwise} \end{cases}$$

We then define a "weak bisimulation" relation $\approx$ on $\hat{H}X$ generated by the clauses,

$$\begin{aligned} (\langle f_1,\dots,f_n\rangle, x) &\approx (\langle g_1,\dots,g_m\rangle, x) & \qquad (f_1{}^\frown\cdots{}^\frown f_n = g_1{}^\frown\cdots{}^\frown g_m) \\ \langle f_1, f_2, \dots\rangle &\approx \langle g_1, g_2, \dots\rangle & \qquad (f_1{}^\frown f_2{}^\frown\cdots = g_1{}^\frown g_2{}^\frown\dots) \end{aligned}$$

Let $HX = \hat{H}X/_{\approx}$. This yields the following functor $H\colon \mathbf{Set} \to \mathbf{Set}$:

$$HX = \sum\nolimits_{r\in[0,\infty)} S^{[0,r)} \times X \cup \sum\nolimits_{r\in[0,\infty]} S^{[0,r)} \tag{2}$$

together with a surjective natural transformation $\rho\colon \hat{H} \to H$. Consider the map $! : [0,0) \to S$, and for every function $f : [0,\infty) \to S$ denote by $f_{|A}$ its restriction to a domain $A \subseteq [0,\infty)$. We fix the following right-inverse of ρ: $\upsilon_X(f,x) = (\langle f\rangle, x)$, $\upsilon_X(f\colon [0,r) \to S) = \langle f, !, !, \dots\rangle$, and $\upsilon_X(f\colon [0,\infty) \to S) = \langle f_{|[0,1)}, f_{|[1,2)}, \dots\rangle$. It is easy to see that υ is natural in X.

Theorem 16. *H is a generalised writer monad with $\left(\sum_{r\in[0,\infty)} S^{[0,r)}, !, ^\frown\right)$ as the monoid and $\left(\sum_{r\in[0,\infty]} S^{[0,r)}, ^\frown\right)$ as the corresponding monoid module. Moreover, $\rho\colon \hat{H} \to H$ is a monad morphism.*

Proof. The monoid and the monoid module laws are easy to check. The fact that ρ is a monad morphism follows directly by case distinction. □

We next use the retraction (ρ, υ) to transfer the guarded Elgot monad structure from $\hat{H}$ to H as follows.

Lemma 17. *The following statements hold.*

1. *For every $f\colon X \to HY$, the map $\upsilon f\colon X \to \hat{H}Y$ is guarded.*
2. *For every guarded $f\colon X \to \hat{H}Y$ the equation $\rho f^\dagger = \rho(\upsilon\rho f)^\dagger$ holds.*

$$\begin{aligned}
\llbracket \mathtt{x} \coloneqq \mathtt{t} \rrbracket(\sigma) &= (!\,, \sigma \triangledown [\mathtt{t}\sigma/\mathtt{x}]) \\
\llbracket \bar{\mathtt{x}}' = \bar{\mathtt{u}}\ \mathtt{for}\ \mathtt{d} \rrbracket(\sigma) &= (\lambda t \in [0, \mathtt{d}).\, \sigma \triangledown [\phi_\sigma(t)/\mathtt{x}], \sigma \triangledown [\phi_\sigma(\mathtt{d})/\mathtt{x}]) \\
\llbracket \mathtt{p}\ ;\ \mathtt{q} \rrbracket(\sigma) &= \llbracket \mathtt{q} \rrbracket^\star(\llbracket \mathtt{p} \rrbracket(\sigma)) \\
\llbracket \mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{p}\ \mathtt{else}\ \mathtt{q} \rrbracket(\sigma) &= \mathit{if}\ \mathtt{b}\sigma\ \mathit{then}\ \llbracket \mathtt{p} \rrbracket(\sigma)\ \mathit{else}\ \llbracket \mathtt{q} \rrbracket(\sigma) \\
\llbracket \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \{\ \mathtt{p}\ \} \rrbracket(\sigma) &= (\lambda\sigma.\, \mathit{if}\ \mathtt{b}\sigma\ \mathit{then}\ (H\,\mathsf{inr})\llbracket \mathtt{p} \rrbracket(\sigma)\ \mathit{else}\ \eta(\mathsf{inl}\,\sigma))^\dagger(\sigma)
\end{aligned}$$

Fig. 6. Denotational semantics.

Proof. The first clause is obvious by definition. Let us stick to the second one. Let $M = \sum_{r \in [0,\infty)} S^{[0,r)}$, and note the following concrete description of $f^\dagger \colon X \to M^\star \times Y \cup M^\omega$ for a given guarded $f^\dagger \colon X \to M^\star \times (Y + X) \cup M^\omega$:

$$\begin{aligned}
f^\dagger(x) &= (w_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} w_n, y) && \text{if} \quad f(x) = (w_1, \mathsf{inr}\, x_1), \ldots, f(x_n) = (w_n, \mathsf{inl}\, y) \\
f^\dagger(x) &= w_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} w_n && \text{if} \quad f(x) = (w_1, \mathsf{inr}\, x_1), \ldots, f(x_n) = w_n \\
f^\dagger(x) &= w_1 \mathbin{+\!\!+} \ldots && \text{if} \quad f(x) = (w_1, \mathsf{inr}\, x_1), \ldots
\end{aligned}$$

In the first scenario, we unfold the fixpoint finitely many times and succeed; in the second scenario, we unfold the fixpoint finitely many times and hit divergence; in the third scenario, we unfold the fixpoint indefinitely. The guardedness assumption is crucial for the last scenario, for otherwise we could potentially have an infinite sequence of empty lists $w_1, w_2, \ldots$ from which we would not be able to obtain an infinite list by concatenation.

Now, the effect of applying the quotienting morphism ρ to $f^\dagger(x)$ amounts to converting the lists of trajectories $w_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} w_n$ and $w_1 \mathbin{+\!\!+} \ldots$ to single trajectories via the $^\frown$ and $!$ operators. Contrastingly, in $\rho(\upsilon\rho f)^\dagger(x)$ we first flatten the lists w_i via $^\frown$ and $!$, then calculate the iteration and then apply $^\frown$ and $!$ again. It is thus clear that both $\rho f^\dagger(x)$ and $\rho(\upsilon\rho f)^\dagger(x)$ produce the same result. □

From Lemma 17 and [16, Theorem 20] we automatically obtain,

Theorem 18. *$(H, \eta, (-)^\star)$ is an Elgot monad with the Elgot iteration sending $f \colon X \to H(Y + X)$ to $\rho(\upsilon f)^\dagger \colon X \to HY$.*

Lemma 17 ensures that ρ is an iteration congruence, which allows to transfer iteration from $\hat{H}$ to H using [16, Theorem 20].

Now, let us fix $S = \mathtt{CTerm}^{\mathcal{X}}$ (2) and proceed by defining the denotational semantics of our while-language. Intuitively, a program $\mathtt{p}$ will be interpreted as a map $\llbracket \mathtt{p} \rrbracket \colon \mathtt{CTerm}^{\mathcal{X}} \to H(\mathtt{CTerm}^{\mathcal{X}})$ that given an environment, providing values for program variables, returns a *trajectory valued on* $\mathtt{CTerm}^{\mathcal{X}}$. This trajectory is either *successful i.e.* an element of the set $\sum_{r \in [0,\infty)} (\mathtt{CTerm}^{\mathcal{X}})^{[0,r)} \times \mathtt{CTerm}^{\mathcal{X}}$, where the element on the right represents the last value of the trajectory, or *divergent i.e.* an element of the set $\sum_{r\in[0,\infty]}(\texttt{CTerm}^{\mathcal{X}})^{[0,r)}$. The definition of $[\![\texttt{p}]\!]$ is inductive over the structure of $\texttt{p}$ and is given in Figure 6.

In order to establish soundness and adequacy between the small-step operational semantics and the denotational one, we will use an auxiliary device. Namely, we will introduce a *big-step* operational semantics that will serve as a midpoint between the two previously introduced semantics. We will show that the small-step semantics is equivalent to the big-step one and then establish soundness and adequacy between the big-step semantics and the denotational one. The desired result then follows by transitivity.

The big-step rules are presented in Figure 7 and follow the same reasoning than the small-step ones. Next, we need the following result to formally connect these two styles of semantics.

$$\textbf{(diff-stop}\Downarrow\textbf{)}\quad \frac{t < \texttt{d}}{\overline{\texttt{x}}' = \overline{\texttt{t}}\ \texttt{for}\ \texttt{d}\,,\sigma\,,t\ \Downarrow\ \mathit{stop}\,,\sigma\nabla[\phi_\sigma(t)/\overline{\texttt{x}}]}$$

$$\textbf{(diff-skip}\Downarrow\textbf{)}\quad \frac{}{\overline{\texttt{x}}' = \overline{\texttt{t}}\ \texttt{for}\ \texttt{d}\,,\sigma\,,\texttt{d}\ \Downarrow\ \mathit{skip}\,,\sigma\nabla[\phi_\sigma(\texttt{d})/\overline{\texttt{x}}]}$$

$$\textbf{(asg}\Downarrow\textbf{)}\quad \frac{}{\texttt{x} := \texttt{t}\,,\sigma\,,0\ \Downarrow\ \mathit{skip}\,,\sigma\nabla[\texttt{t}\sigma/\texttt{x}]} \qquad\qquad \textbf{(seq-stop}\Downarrow\textbf{)}\quad \frac{\texttt{p}\,,\sigma\,,t\ \Downarrow\ \mathit{stop}\,,\sigma'}{\texttt{p}\,;\texttt{q}\,,\sigma\,,t\ \Downarrow\ \mathit{stop}\,,\sigma'}$$

$$\textbf{(seq-skip}\Downarrow\textbf{)}\quad \frac{\texttt{p}\,,\sigma\,,t\ \Downarrow\ \mathit{skip}\,,\sigma' \qquad \texttt{q}\,,\sigma'\,,t'\ \Downarrow\ r\,,\sigma''}{\texttt{p}\,;\texttt{q}\,,\sigma\,,t+t'\ \Downarrow\ r\,,\sigma''} \qquad (r \in \{\mathit{stop}, \mathit{skip}\})$$

$$\textbf{(if-true}\Downarrow\textbf{)}\quad \frac{\texttt{b}\sigma = \top \qquad \texttt{p}\,,\sigma\,,t\ \Downarrow\ r\,,\sigma'}{\texttt{if}\ \texttt{b}\ \texttt{then}\ \texttt{p}\ \texttt{else}\ \texttt{q}\,,\sigma\,,t\ \Downarrow\ r\,,\sigma'} \qquad (r \in \{\mathit{stop}, \mathit{skip}\})$$

$$\textbf{(if-false}\Downarrow\textbf{)}\quad \frac{\texttt{b}\sigma = \bot \qquad \texttt{q}\,,\sigma\,,t\ \Downarrow\ r\,,\sigma'}{\texttt{if}\ \texttt{b}\ \texttt{then}\ \texttt{p}\ \texttt{else}\ \texttt{q}\,,\sigma\,,t\ \Downarrow\ r\,,\sigma'} \qquad (r \in \{\mathit{stop}, \mathit{skip}\})$$

$$\textbf{(wh-false}\Downarrow\textbf{)}\quad \frac{b\sigma = \bot}{\texttt{while}\ \texttt{b}\ \texttt{do}\ \{\ \texttt{p}\ \}\,,\sigma\,,0\ \Downarrow\ \mathit{skip}\,,\sigma}$$

$$\textbf{(wh-true}\Downarrow\textbf{)}\quad \frac{b\sigma = \top \qquad \texttt{p}\,;\texttt{while}\ \texttt{b}\ \texttt{do}\ \{\ \texttt{p}\ \}\,,\sigma\,,t\ \Downarrow\ r\,,\sigma'}{\texttt{while}\ \texttt{b}\ \texttt{do}\ \{\ \texttt{p}\ \}\,,\sigma\,,t\ \Downarrow\ r\,,\sigma'} \qquad (r \in \{\mathit{stop}, \mathit{skip}\})$$

Fig. 7. Big-step Operational Semantics

Lemma 19. *Given a program* $\texttt{p}$*, an environment* σ *and a time instant* t*,*

1. *if* $\texttt{p}\,,\sigma\,,t\ \to\ \texttt{p}'\,,\sigma'\,,t'$ *and* $\texttt{p}'\,,\sigma'\,,t'\ \Downarrow\ \mathit{skip}\,,\sigma''$ *then* $\texttt{p}\,,\sigma\,,t\ \Downarrow\ \mathit{skip}\,,\sigma''$*;*
2. *if* $\texttt{p}\,,\sigma\,,t\ \to\ \texttt{p}'\,,\sigma'\,,t'$ *and* $\texttt{p}'\,,\sigma'\,,t'\ \Downarrow\ \mathit{stop}\,,\sigma''$ *then* $\texttt{p}\,,\sigma\,,t\ \Downarrow\ \mathit{stop}\,,\sigma''$*.*

Proof. The proofs follows by induction over the derivation of the small step relation. □

Theorem 20. *The small-step semantics and the big-step semantics are related as follows. Given a program* $\mathtt{p}$*, an environment* σ *and a time instant* t*,*

1. $\mathtt{p}\,,\sigma\,,t \Downarrow \mathit{skip}\,,\sigma'$ *iff* $\mathtt{p}\,,\sigma\,,t \to^{\star} \mathit{skip}\,,\sigma'\,,0$*;*
2. $\mathtt{p}\,,\sigma\,,t \Downarrow \mathit{stop}\,,\sigma'$ *iff* $\mathtt{p}\,,\sigma\,,t \to^{\star} \mathit{stop}\,,\sigma'\,,0$.

Proof. The right-to-left direction is obtained by induction over the length of the small-step reduction sequence using Lemma 19. The left-to-right direction follows by induction over the proof of the big-step judgement using Proposition 7. □

We now can connect the operational and the denotational semantics in the expected way.

Theorem 21 (Soundness and Adequacy). *Given a program* $\mathtt{p}$*, an environment* σ *and a time instant* t*,*

1. $\mathtt{p}\,,\sigma\,,t \to^{\star} \mathit{skip}\,,\sigma'\,,0$ *iff* $[\![\mathtt{p}]\!](\sigma) = (h\colon [0,t) \to \mathtt{CTerm}^{\mathcal{X}}, \sigma')$*;*
2. $\mathtt{p}\,,\sigma\,,t \to^{\star} \mathit{stop}\,,\sigma'\,,0$ *iff either* $[\![\mathtt{p}]\!](\sigma) = (h\colon [0,t') \to \mathtt{CTerm}^{\mathcal{X}}, \sigma'')$ *or* $[\![\mathtt{p}]\!](\sigma) = h\colon [0,t') \to \mathtt{CTerm}^{\mathcal{X}}$*, and in either case with* $t' > t$ *and* $h(t) = \sigma'$*.*

Here, “soundness” corresponds to the left-to-right directions of the equivalences and “adequacy” to the right-to-left ones.

Proof. By Theorem 20, we equivalently replace the goal as follows:

1. $\mathtt{p}\,,\sigma\,,t \Downarrow \mathit{skip}\,,\sigma'$ iff $[\![\mathtt{p}]\!](\sigma) = (h\colon [0,t) \to \mathtt{CTerm}^{\mathcal{X}}, \sigma')$;
2. $\mathtt{p}\,,\sigma\,,t \Downarrow \mathit{stop}\,,\sigma'$ iff either $[\![\mathtt{p}]\!](\sigma) = (h\colon [0,t') \to \mathtt{CTerm}^{\mathcal{X}}, \sigma'')$ or $[\![\mathtt{p}]\!](\sigma) = h\colon [0,t') \to \mathtt{CTerm}^{\mathcal{X}}$, and in either case with $t' > t$ and $h(t) = \sigma'$.

Then the “soundness” direction is obtained by induction over the derivation of the rules in Fig. 7. The “adequacy” direction follows by structural induction over $\mathtt{p}$; for while-loops, we call on the fixpoint law $[\eta, f^{\dagger}]^{\star} f = f^{\dagger}$ of Elgot monads. □

5 Conclusions and future work

We introduced a small-step operational semantics for interpreting hybrid programs and moreover provided a denotational counterpart via the notion of Elgot monad. The two semantics were then linked by a soundness and adequacy theorem. We regard these results as a firm stepping stone for developing tools and techniques for hybrid systems engineering. In fact, we illustrated our results with the implementation of an interpreter – Lince – and showed its potential for detecting design errors in hybrid programs at an early development phase.

The development of Lince and its theoretical foundations open up research lines that we intend to explore in the near future; we briefly present some of them next.

***Calculi and program equivalence*.** Our denotational semantics entails a natural notion of program equivalence (denotational equivalence) which inherently includes classical laws of iteration and a powerful *uniformity* principle [36], thanks to the use of Elgot monads. We intend to further explore the equational theory of our language so that we can safely refactor/simplify hybrid programs under consideration. Such a theory should include equations like,

$$(\mathtt{x} \coloneqq 1\,;\mathtt{x} \coloneqq 2) \;=\; \mathtt{x} \coloneqq 2, \text{ and } \quad (\mathtt{wait}\,1\,;\mathtt{wait}\,2) \;=\; \mathtt{wait}\,3$$

thus encompassing not only the usual laws of imperative programming but also axiomatic principles behind the notion of time.

***New program constructs*.** Our while-language is intended to be as simple as possible whilst harbouring the core, uncontroversial features of hybrid programming. This was decided so that we could use the language as both a theoretical and practical basis for advancing hybrid programming. A particular case that we wish to explore, with this language and its semantics as basis, is the introduction of new program constructs, including *e.g.* the non-deterministic choice $\mathtt{p} + \mathtt{q}$ or exception operations $\mathtt{raise}(\mathtt{exc})$. Denotationally, this corresponds to combining the presented monad H with other monads representing the desired features [26,27].

***Robustness*.** One important aspect of hybrid programming is that programs should be *robust*: *i.e.* small variations in their input should *not* result in big changes in their output [35,23,12]. We wish to extend Lince with features for automatically detecting non-robust programs. A main source of non-robustness are conditional statements $\mathtt{if}\;\mathtt{b}\;\mathtt{then}\;\mathtt{p}\;\mathtt{else}\;\mathtt{q}$, for which small variations of σ can change the validity of $\mathtt{b}\sigma$ and consequently can cause a switch between execution branches. Currently, we are working on the systematic detection of such conditional statements in hybrid programs, by taking advantage of the notion of δ-perturbation [20]. The reader can already check a preliminary version of this feature in Lince (window "Perturbations up-to").

References

1. Abramsky, S.: Intensionality, definability and computation. In: Johan van Benthem on Logic and Information Dynamics, pp. 121–142. Springer (2014)
2. Adámek, J., Milius, S., Velebil, J.: Elgot theories: a new perspective on the equational properties of iteration. Mathematical Structures in Computer Science **21**(2), 417–480 (2011)
3. Benvenuti, L., Bresolin, D., Collins, P., Ferrari, A., Geretti, L., Villa, T.: Assume-guarantee verification of nonlinear hybrid systems with Ariadne. Int. J. Robust. Nonlinear Control **24**(4), 699–724 (2014)
4. Bouissou, O., Chapoutot, A.: An operational semantics for simulink's simulation engine. In: ACM SIGPLAN Notices. vol. 47, pp. 129–138. ACM (2012)
5. Broman, D.: Hybrid simulation safety: Limbos and zero crossings. In: Principles of Modeling, pp. 106–121. Springer (2018)
6. Chaochen, Z., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Information Processing Letters **40**(5), 269–276 (1991)

7. Copp, D.A., Sanfelice, R.G.: A zero-crossing detection algorithm for robust simulation of hybrid systems jumping on surfaces. Simulation Modelling Practice and Theory **68**, 1–17 (2016)
8. Davoren, J.M.: On hybrid systems and the modal μ-calculus. In: Hybrid Systems V. Lecture Notes in Computer Science, vol. 1567, pp. 38–69. Springer (1997)
9. Foster, S., Thiele, B., Cavalcanti, A., Woodcock, J.: Towards a UTP semantics for Modelica. In: International Symposium on Unifying Theories of Programming. pp. 44–64. Springer (2016)
10. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: Spaceex: Scalable verification of hybrid systems. In: 23rd International Conference on Computer Aided Verification (CAV). Springer (2011)
11. Fritzson, P.: Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach. John Wiley & Sons (2014)
12. Goebel, R., Hespanha, J., Teel, A.R., Cai, C., Sanfelice, R.: Hybrid systems: generalized solutions and robust stability. NOLCOS04': Nonlinear Control Systems 2004, 6th Symposium, Stuttgart, Germany, 1-3 September. Elsevier **37**(13), 1–12 (2004)
13. Goebel, R., Sanfelice, R.G., Teel, A.R.: Hybrid dynamical systems. IEEE Control Systems **29**(2), 28–93 (2009)
14. Goncharov, S., Jakob, J., Neves, R.: A semantics for hybrid iteration. In: 29th International Conference on Concurrency Theory, CONCUR 2018. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
15. Goncharov, S., Neves, R.: An adequate while-language for hybrid computation. In: Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019. pp. 11:1–11:15. PPDP '19, ACM, New York, NY, USA (2019)
16. Goncharov, S., Schröder, L., Rauch, C., Piróg, M.: Unifying guarded and unguarded iteration. In: International Conference on Foundations of Software Science and Computation Structures. pp. 517–533. Springer (2017)
17. Henzinger, T.A.: The theory of hybrid automata. In: LICS96': Logic in Computer Science, 11th Annual Symposium, New Jersey, USA, July 27-30, 1996. pp. 278–292. IEEE (1996)
18. Henzinger, T.A., Ho, P.H., Wong-toi, H.: Hytech: A model checker for hybrid systems. Software Tools for Technology Transfer **1**, 460–463 (1997)
19. Klee, H.: Simulation of dynamic systems with MATLAB and Simulink. CRC Press (2007)
20. Kong, S., Gao, S., Chen, W., Clarke, E.: dreach: δ-reachability analysis for hybrid systems. In: International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems. pp. 200–205. Springer (2015)
21. Lee, E.A., Seshia, S.A.: Introduction to embedded systems: A cyber-physical systems approach. MIT Press (2016)
22. Levy, P.B., Goncharov, S.: Coinductive resumption monads: Guarded iterative and guarded elgot. In: Proc. 8rd international conference on Algebra and coalgebra in computer science (CALCO 2019). LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
23. Liberzon, D., Morse, A.S.: Basic problems in stability and design of switched systems. IEEE Control systems **19**(5), 59–70 (1999)
24. Liu, B., Kong, S., Gao, S., Zuliani, P., Clarke, E.M.: Towards personalized prostate cancer therapy using delta-reachability analysis. In: Proceedings of the 18th Inter-

national Conference on Hybrid Systems: Computation and Control. pp. 227–232. ACM (2015)

25. Liu, X., Stechlinski, P.: Infectious Disease Modeling. Springer (2017)
26. Lüth, C., Ghani, N.: Composing monads using coproducts. In: Wand, M., Jones, S.L.P. (eds.) ICFP'02: Functional Programming, 7th ACM SIGPLAN International Conference, Pittsburgh, USA, October 04 - 06, 2002. pp. 133–144. ACM (2002)
27. Manes, E., Mulry, P.: Monad compositions i: general constructions and recursive distributive laws. Theory and Applications of Categories **18**(7), 172–208 (2007)
28. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989. pp. 14–23. IEEE Computer Society (1989)
29. Moggi, E.: Notions of computation and monads. Information and computation **93**(1), 55–92 (1991)
30. Neves, R.: Hybrid programs. Ph.D. thesis, Minho University (2018)
31. Odersky, M., Spoon, L., Venners, B.: Programming in scala. Artima Inc (2008)
32. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010)
33. Rajkumar, R.R., Lee, I., Sha, L., Stankovic, J.: Cyber-physical systems: the next computing revolution. In: DAC'10: Design Automation Conference, 47th ACM/IEEE Conference, Anaheim, USA, June 13-18, 2010. pp. 731–736. IEEE (2010)
34. Schupp, S., Ábrahám, E.: Spread the work: Multi-threaded safety analysis for hybrid systems. In: Software Engineering and Formal Methods - 16th International Conference, SEFM 2018. Springer (2018)
35. Shorten, R., Wirth, F., Mason, O., Wulff, K., King, C.: Stability criteria for switched and hybrid systems. Society for Industrial and Applied Mathematics (review) **49**(4), 545–592 (2007)
36. Simpson, A., Plotkin, G.: Complete axioms for categorical fixed-point operators. In: Logic in Computer Science, LICS 2000. pp. 30–41 (2000)
37. Stein, W., et al.: Sage Mathematics Software (Version 6.4.1). The Sage Development Team (2015), `http://www.sagemath.org`
38. Suenaga, K., Hasuo, I.: Programming with infinitesimals: A while-language for hybrid system modeling. In: International Colloquium on Automata, Languages, and Programming. pp. 392–403. Springer (2011)
39. Uustalu, T.: Generalizing substitution. RAIRO-Theoretical Informatics and Applications **37**(4), 315–336 (2003)
40. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text. Lecture Notes in Computer Science, vol. 925, pp. 24–52. Springer (1995)
41. Winskel, G.: The formal semantics of programming languages: an introduction. MIT press (1993)
42. Witsenhausen, H.: A class of hybrid-state continuous-time dynamic systems. IEEE Transactions on Automatic Control **11**(2), 161–167 (1966)
43. Zhan, N., Wang, S., Zhao, H.: Formal Verification of Simulink/Stateflow Diagrams. Springer (2017)