Formal simulation and visualisation of hybrid programs

Pedro Mendes
University of Minho, Portugal
pg50685@alunos.uminho.pt

Ricardo Correia
University of Minho, Portugal
pg47607@alunos.uminho.pt

Renato Neves
INESC-TEC & University of Minho, Portugal
nevrenato@di.uminho.pt

José Proença
CISTER, Faculty of Sciences of the University of Porto, Portugal
jose.proenca@fc.up.pt

The design and analysis of systems that combine computational, discrete behaviour with physical processes' continuous dynamics – such as movement, velocity, and voltage – is a famous, challenging task. Several theoretical results from programming theory emerged in the last decades to tackle the issue; some of which are the basis of a *proof-of-concept* tool, called Lince, that aids in the analysis of such systems, by presenting simulations of their respective behaviours.

Being a proof-of-concept, the tool is quite limited w.r.t. usability, and when attempting to apply it to a set of common, concrete problems, involving autonomous driving and others, it either simply cannot simulate them or fails to provide a satisfactory user-experience.

This work thus complements the aforementioned theoretical approaches with a more practical perspective by improving Lince along several dimensions: to name a few, richer syntactic constructs, more operations, more informative plotting systems and errors messages, and a better performance overall. We illustrate our improvements via a variety of examples that involve both autonomous driving and electrical systems.

1 Introduction

Motivation and context. This paper concerns the design and analysis of hybrid systems by means from a programming-oriented perspective. This view emerged recently in a series of works [25, 22, 10, 15], and revolves around the idea of importing principles and techniques from programming theory to better handle the (often) complex behaviour of hybrid systems. In this context programs combine standard program constructs, such as conditionals and while-loops, with certain kinds of differential statement meant to express the dynamics of physical processes, such as time, energy, and motion. Consider the following example of such a program:

$$p' = v, v' = 2 \texttt{ for } 1 \texttt{ ; } p' = v, v' = -2 \texttt{ for } 1 \tag{1}$$

In a nutshell, it is a sequential composition $(\texttt{;})$ of two simpler programs where each expresses how the position (p) and velocity (v) of a vehicle evolve over time. The program on the left $(p' = v, v' = 2 \texttt{ for } 1)$ is a differential statement that reads as "the vehicle accelerates at the rate of $2^{m}/_{s^2}$ for 1 second". Similarly the other program corresponds to a deceleration. Both position and velocity over time are presented in Fig. 1, where we see that the vehicle travelled precisely 2 meters and then stopped.

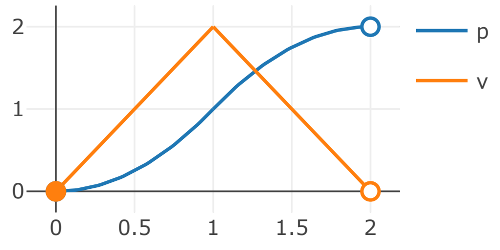


Figure 1: Simulation of (1).

Submitted to:
FMAS 2024

Actually there has been a rapid proliferation of such systems, not only in the domain of autonomous driving but also in the medical industry and industrial infrastructures, among others [25, 12, 19, 22]. This spurred extensive research on languages, semantics, and tools for their design and analysis. An example is our work [9, 10] on the semantics of hybrid programs – *i.e.* those that combine program constructs with differential statements, such as in (1) – from which arises a mathematical basis for reasoning about their behaviour, both operationally and denotationally. A proof-of-concept tool, called Lince, also emerged from this: its engine is a previously developed operational semantics [9, 10] that yields trajectories of hybrid programs, just as we saw in Fig. 1. However because our focus was rather theoretical, the tool was not developed with usability in mind, and thus lacks basic features for tackling a broad range of important scenarios. Let us illustrate this problem with a very simple example.

Problem scenario. Suppose that we wish to move a stationary object a distance of *dist* meters – a basic task in autonomous driving. For simplicity assume that we have access only to the acceleration rates a^{m}/and-a$^{m}/, where$a $> 0. Ourmissioncanbeaccomplishedbytakingthefollowingvariationof$ Eq. (1)$, forasuitableduration$ t.T
$\int_0^t v_a(x)\,dx \ + \ \int_0^t v_{-a}(x)\,dx$ where $v_a(x) = a \cdot x$ and $v_{-a}(x) = v_a(t) + -a \cdot x$ are the velocity functions w.r.t. the time intervals $[0,t]$ and $[t, 2 \cdot t]$ associated with the program's execution. We now observe, by recalling Fig. 1, that the value *dist* corresponds to the area of a triangle with basis $2 \cdot t$ and height $v_a(t)$. This geometric shape yields the equations,

$$\begin{cases} dist & = {}^1\!/_2 \cdot (2 \cdot t) \cdot v_a(t) \ \ \textbf{(area)} \\ v_a(t) & = a \cdot t \ \ \textbf{(height)} \end{cases} \quad \Longrightarrow \quad t = \sqrt{\frac{dist}{a}}$$

Finally observe that if $dist = 3$ and $a = 1$ then $t = \sqrt{3}$. Unfortunately the previous version of Lince does not support square root operations which renders our mission impossible to accomplish.

Contributions and outline. As already alluded to, this paper complements our previous theoretical work on the semantics of hybrid programming [9, 10]. Specifically it improves our proof-of-concept tool Lince so that it can handle a myriad of important scenarios, whilst maintaining both its simplicity and theoretical underpinnings. The improvements were made along different dimensions, and we highlight the most relevant ones next[1].

Extension of basic operations. As illustrated before, the previous version of Lince lacked essential arithmetic operations for handling most basic tasks. Thus as the first main contribution we added standard arithmetic operations, including divisions, trigonometric functions, and square root extractions. Notably the fact that many of these operations are partial required us to extend the operational semantics developed in [10] (the main engine of Lince) with the possibility of failure. The extended semantics is detailed in Section 2 and it is of course the basis of the new engine behind improved Lince.

Extension of numerical methods. Again because our focus in previous work was rather theoretical the previous version of Lince was unable to simulate standard scenarios in hybrid programming. A main reason for this was our method of obtaining solutions of systems of ordinary differential equations (ODEs), which although *exact* lacked in scalability. Precisely for this reason we now integrate a complementary, numerical solver in Lince with the obvious compromise that the solutions obtained for such systems are no longer exact.

The benefits of the extended language (and respective semantics), the numerical solver, and a number of quality-of-life features, are summarised in Section 3 and illustrated with a standard, running example concerning the famous concept of harmonic oscillation.

[1]The improved version can be checked online at `http://arcatools.org/lince`.

Extension of visualisation mechanisms. Lince is constituted by two core components: the simulator which, by recurring to the aforementioned operational semantics, parses a received program and presents its output w.r.t. a *single* time instant. And the visualiser which presents (a sample of) the trajectory over time respective to the program at hand, by querying the operational semantics for a certain sequence of time instants. After trying to properly visualise the behaviour of several types of hybrid program with Lince we identified two major limitations w.r.t. this architecture. First many real-world problems involve multiple spatial dimensions and thus the described view of trajectories over time is often not the best representation of the behaviour of a hybrid program. Second the user is frequently interested in observing the overall program behaviour for varying initial conditions, concerning for example position and velocity. We therefore present in Section 4 an improved visualiser for Lince that precisely addresses these problems. We illustrate it via another classical scenario in autonomous driving, *viz.* manoeuvring around an obstacle.

In Section 5 we illustrate that, whilst keeping its simplicity, Lince can now handle complex central problems in the theory of hybrid systems; we focus specifically on the task of one player pursuing another, *e.g.* a vehicle, a drone, or simply a projectile. Such pursuit games were discussed for example in [21, 2, 5, 18], from an (hybrid-)automata, state-chart, and duration calculus perspective. Here we present a programming-oriented approach. Finally in Section 6 we discuss future work and conclude.

Related work. Several tools for the design and analysis of hybrid systems were already proposed (see for example [25, 16, 15, 8, 3, 7, 10]). However only a few are committed to a programming-oriented approach, rooted on formal semantics, and with effective simulation capabilities. The only ones we are aware of are [15] and our tool Lince [10]. Interestingly both cases adopt complementary approaches as well: the former harbours a very powerful concurrent language, particularly well-suited for large-scale distributed systems. The latter, harbouring a sequential while-language, aims at being minimalistic whilst still capturing a broad range of interesting problems on which to study different aspects of (pure) hybrid computation at a suitable abstraction level.

Aside from the obvious pedagogical benefit, our minimalistic approach also allows us to capitalise on different programming theories more easily. For example already in [10] we connected our tool to a compositional, denotational semantics – particularly well-suited to study hybrid program equivalence and combinations with other paradigms. An analogous concurrent semantics for [15] would be notoriously more difficult to achieve (*cf.* [27, 29]). Similarly our language is amenable to algebraic reasoning in the style of (weak) Kleene algebras [17, 14] whilst the connection between the latter and concurrent object-oriented programming (as adopted in [15]) is less clear.

2 Lince's foundations extended with the possibility of failure

We now extend part of Lince's foundations with the possibility of failure. Specifically we present an extension of the language in [10] with partial operations, such as division and square root extraction, and introduce a corresponding operational semantics. As explained in the introduction, such is necessary for extending Lince to 'real-world problems' whilst preserving its merit of having a firm, mathematical basis.

Language. First we postulate a finite set $X = \{x_1, \ldots, x_n\}$ of variables and a stock of partial functions $f : \mathbb{R}^n \rightharpoonup \mathbb{R}$ that contains the usual arithmetic operations. Then we define expressions and boolean conditions via the following BNF grammars,

$$e ::= x \mid f(e, \ldots, e) \qquad\qquad b ::= e \leq e \mid b \wedge b \mid b \vee b \mid \neg b \mid \texttt{tt} \mid \texttt{ff}$$

We omit the explanation of these grammars as they are widely used (see *e.g.* [29, 27]). Next, we qualify as 'linear' those expressions e which aside from the use of variables involve only the operations $+$ and $r \cdot (-)$ for some $r \in \mathbb{R}$. For example the expression $2 \cdot x$ is linear but the expression $x \cdot x$ is not. The concept of linearity is key in the grammar of hybrid programs which we present next.

Programs are built according to the following BNF grammars,

$$\mathtt{a} ::= x_1' = \ell_1, \ldots, x_n' = \ell_n \ \mathtt{for} \ e \mid x := e$$
$$\mathtt{p} ::= \mathtt{a} \mid \mathtt{p}\mathtt{;}\mathtt{p} \mid \mathtt{if} \ b \ \mathtt{then} \ \mathtt{p} \ \mathtt{else} \ \mathtt{p} \mid \mathtt{while} \ b \ \mathtt{do} \ \{ \ \mathtt{p} \ \}$$

where the terms ℓ_i $(1 \leq i \leq n)$ are linear expressions. We qualify as 'atomic' those hybrid programs that are built according to the first grammar. They can be either classical assignments or *differential* statements as described in the introduction. The linearity constraint is here imposed merely to ensure that the latter kind of statement will always have unique solutions, which renders our semantics more lightweight whilst still being able to treat a broad range of problems (see more details in [10]).

The language of hybrid programs $\mathtt{p}$ itself is simply the usual while-language [29, 27] extended with the aforementioned differential statements. It is easy to check that our grammar indeed extends that in the previous version of Lince [10] where *all* expressions involved in the assignments and the durations of differential statements had to be linear. This has of course significant implications in the operational semantics introduced in [10].

Operational semantics. We need a series of preliminaries. First for simplicity we abbreviate differential statements $x_1' = \ell_1, \ldots, x_n' = \ell_n \ \mathtt{for} \ e$ simply to $\vec{x}' = \vec{\ell} \ \mathtt{for} \ e$, where $\vec{x}'$ and $\vec{\ell}$ abbreviate the corresponding vectors of variables $x_1' \ldots x_n'$ and linear expressions $\ell_1 \ldots \ell_n$. We call functions of the type $\sigma : X \to \mathbb{R}$ *environments*; they map variables to the respective valuations. We use the notation $\sigma[\vec{x} \mapsto \vec{v}]$ to denote the environment that maps each x_i in $\vec{x}$ to v_i in $\vec{v}$ and the remaining variables as in σ. Finally we denote by $\phi_\sigma^{\vec{x}'=\vec{\ell}} : \mathbb{R}_{\geq 0} \to \mathbb{R}^n$ the (unique) solution of a system of differential equations $\vec{x}' = \vec{\ell}$ with σ as the initial condition (recall our previous constraint about linearity). When clear from context, we omit both the superscript and subscript in $\phi_\sigma^{\vec{x}'=\vec{\ell}}$. Next, for an expression e the notation $[\![e]\!](\sigma)$ denotes the standard (partial) interpretation of expressions [29, 27] according to σ, and analogously for $[\![b]\!](\sigma)$ where b is a boolean expression. For example $[\![x+1]\!](\sigma) = \sigma(x) + 1$ and $[\![1/x]\!](\sigma)$ is undefined if $\sigma(x) = 0$.

We now present an operational semantics for the language. Following traditions in programming theory [23, 29, 27], we present it from two different, complementary perspectives, which gives a much more complete understanding of the language's features. Specifically we present the semantics in two different styles: one formalises the idea of a machine "running" a hybrid program and describes its step-by-step evolution. The other abstracts away from all intermediate steps of this machine and is therefore generally more suitable to reason about "input-output behaviours". Whilst the former style is the basis of Lince's new version, the latter style is conceptually more intuitive and therefore we present it first. The current section concludes with a proof that both semantics are in fact equivalent.

Our 'big-step' operational semantics is given by an 'input-output' relation $\Downarrow$ which relates programs $\mathtt{p}$, environments σ, and time instants t to outputs v. The expression $\mathtt{p}, \sigma, t \Downarrow v$ can be read as "at time instant t the program $\mathtt{p}$ starting from state σ outputs v". The relation $\Downarrow$ is built inductively according to the rules in Fig. 2. Specifically the first three rules describe how differential statements are evaluated: first one computes the duration $[\![e]\!](\sigma)$ of the differential statement at hand and an error is raised if $[\![e]\!](\sigma)$ is undefined; otherwise the output v is the respective modified state (as dictated by the differential statement) paired with one of the flags *stop* or *skip*. Intuitively the flag *stop* indicates that we 'reached' the time instant at which the program needs to be evaluated and therefore the evaluation can stop moving forward in time, which fact is reflected in rule (**seq-stop**). The flag *skip* is simply the negation of *stop*.

The remaining rules follow analogous principles and therefore we refrain from detailing them – instead we will briefly show how the semantics works via instructive, concrete examples.

$$\textbf{(diff-skip)}\quad \frac{[\![e]\!](\sigma) = t}{\vec{x}' = \vec{\ell}\ \texttt{for}\ e, \sigma, t \Downarrow \mathit{skip}, \sigma[\vec{x} \mapsto \phi(t)]}$$

$$\textbf{(diff-stop)}\quad \frac{[\![e]\!](\sigma) > t}{\vec{x}' = \vec{\ell}\ \texttt{for}\ e, \sigma, t \Downarrow \mathit{stop}, \sigma[\vec{x} \mapsto \phi(t)]} \qquad \textbf{(diff-err)}\quad \frac{[\![e]\!](\sigma)\ \text{undefined}}{\vec{x}' = \vec{\ell}\ \texttt{for}\ e, \sigma, t \Downarrow \mathit{err}}$$

$$\textbf{(asg-skip)}\quad \frac{}{x := e, \sigma, 0 \Downarrow \mathit{skip}, \sigma[x \mapsto [\![e]\!](\sigma)]} \qquad \textbf{(asg-err)}\quad \frac{[\![e]\!](\sigma)\ \text{undefined}}{x := e, \sigma, t \Downarrow \mathit{err}}$$

$$\textbf{(seq-skip)}\quad \frac{\texttt{p}, \sigma, t \Downarrow \mathit{skip}, \tau \qquad \texttt{q}, \tau, u \Downarrow v}{\texttt{p;q}, \sigma, t+u \Downarrow v}$$

$$\textbf{(seq-stop)}\quad \frac{\texttt{p}, \sigma, t \Downarrow \mathit{stop}, \tau}{\texttt{p;q}, \sigma, t \Downarrow \mathit{stop}, \tau} \qquad \textbf{(seq-err)}\quad \frac{\texttt{p}, \sigma, t \Downarrow \mathit{err}}{\texttt{p;q}, \sigma, t \Downarrow \mathit{err}}$$

$$\textbf{(if-true)}\quad \frac{[\![b]\!](\sigma) = \texttt{tt} \qquad \texttt{p}, \sigma, t \Downarrow v}{\texttt{if}\ b\ \texttt{then p else q}, \sigma, t \Downarrow v}$$

$$\textbf{(if-false)}\quad \frac{[\![b]\!](\sigma) = \texttt{ff} \qquad \texttt{q}, \sigma, t \Downarrow v}{\texttt{if}\ b\ \texttt{then p else q}, \sigma, t \Downarrow v} \qquad \textbf{(if-err)}\quad \frac{[\![b]\!](\sigma)\ \text{undefined}}{\texttt{if}\ b\ \texttt{then p else q}, \sigma, t \Downarrow \mathit{err}}$$

$$\textbf{(wh-true)}\quad \frac{[\![b]\!](\sigma) = \texttt{tt} \qquad \texttt{p;while}\ b\ \texttt{do}\ \{\ p\ \}, \sigma, t \Downarrow v}{\texttt{while}\ b\ \texttt{do}\ \{\ \texttt{p}\ \}, \sigma, t \Downarrow v}$$

$$\textbf{(wh-false)}\quad \frac{[\![b]\!](\sigma) = \texttt{ff}}{\texttt{while}\ b\ \texttt{do}\ \{\ \texttt{p}\ \}, \sigma, 0 \Downarrow \mathit{skip}, \sigma} \qquad \textbf{(wh-err)}\quad \frac{[\![b]\!](\sigma)\ \text{undefined}}{\texttt{while}\ b\ \texttt{do}\ \{\ \texttt{p}\ \}, \sigma, t \Downarrow \mathit{err}}$$

Figure 2: Extension of the big-step operational semantics in [10] with the possibility of failure.

Example 2.1. Let us consider the following very simple program,

$$x' = -1\ \texttt{for}\ 1\ \texttt{;}\ x := {}^{1}\!/\!_{x}$$

which continuously decreases the value of variable x during 1 second and then applies the (discrete) operation $x := {}^{1}\!/\!_{x}$. Suppose as well that our initial state is the environment σ defined by $x \mapsto 1$. Then by an application of rule **(diff-stop)** one deduces that this program outputs the environment $x \mapsto 1 - t$ at every time instant $t < 1$. On the other hand, by an application of rules **(diff-skip)**, **(asg-err)**, and **(seq-skip)** one deduces that the evaluation of the program fails at every time instant $t \geq 1$.

Notably the fact that failure occurs only at the time instants $t \geq 1$ is a fundamental difference w.r.t. the famous hybrid programming language detailed in [25]. In the *op. cit.* the language was designed in the spirit of Kleene algebra, which in particular forces the previous program to be *indistinguishable* from *e.g.* the program $x := {}^{x}\!/\!_{0}$. Whilst such a feature could be desirable in some verification scenarios it is clearly unnatural in a simulation-based environment such as ours.

Let us continue unravelling prominent features of our semantics with another example. Consider the following hybrid program,

$$\texttt{while}\ x \neq 0\ \texttt{do}\ \{\ x' = -1\ \texttt{for}\ {}^{x}\!/\!{}_{2}\ \}\ ;\ x := {}^{1}\!/\!{}_{x}$$

paired with the environment $x \mapsto 1$. This program is an instance of a so-called Zeno loop: *viz.* the loop involved unfolds infinitely many times with the duration of each iteration becoming shorter and shorter (see details *e.g.* in [10]). In this particular case it is straightforward to check that the duration of the i-th iteration is given by ${}^{1}\!/\!{}_{2^i}$, and thus that the total duration $\sum_{i=1}^{\infty} {}^{1}\!/\!{}_{2^i}$ of the loop will be 1. By applying the operational rules in Fig. 2 one can successfully evaluate the program at every time instant $t < 1$, but not at time instant $t = 1$ since such requires a complete unfolding of the loop which is of course computationally unfeasible. What is more, the potential point of failure $x := {}^{1}\!/\!{}_{x}$ in the program above can never occur, as the Zeno loop makes it impossible to actually reach the command in the evaluation.

Next, the semantics in the aforementioned 'small-step' style is given in the form of a relation $\rightarrow$ that is defined inductively according to the rules in Fig. 3. These rules follow an analogous reasoning to the ones in Fig. 2 so we refrain from repeating explanations.

As detailed in Corollary 1 our small-step semantics is deterministic. This is of course a key property in what concerns its implementation and subsequent use in Lince for simulating hybrid programs. The corollary is based on the following theorem.

Theorem 2.1. For every program $\mathtt{p}$, environment σ, and time instant t there is *at most one* applicable reduction rule.

Let $\rightarrow^{\star}$ be the transitive closure of the small-step relation $\rightarrow$ that was previously presented. Intuitively $\rightarrow^{\star}$ represents an evaluation of one or more steps according to the small-step semantics. If $\mathtt{p}, \sigma, t \rightarrow^{\star} v$ we call v 'non-terminal' whenever it is of the form $\mathtt{p}', \sigma', t'$ for some hybrid program $\mathtt{p}'$, environment σ', and time instant t'; we call v 'terminal' otherwise.

Corollary 1 (Determinism)**.** Consider a program $\mathtt{p}$, an environment σ, and a time instant t. If $\mathtt{p}, \sigma, t \rightarrow^{\star} v$ and $\mathtt{p}, \sigma, t \rightarrow^{\star} u$ with both v and u terminal then we have $v = u$.

Proof. Follows by induction on the number of reduction steps and Theorem 2.1. □

Next we will show that the small-step semantics and its big-step counterpart are indeed equivalent. We will use the two following results for this effect.

Lemma 2.1. Given a program $\mathtt{p}$, an environment σ and a time instant t

1. if $\mathtt{p}, \sigma, t \rightarrow \mathtt{p}', \sigma', t'$ and $\mathtt{p}', \sigma', t' \Downarrow \mathit{skip}, \sigma''$ then $\mathtt{p}, \sigma, t \Downarrow \mathit{skip}, \sigma''$;
2. if $\mathtt{p}, \sigma, t \rightarrow \mathtt{p}', \sigma', t'$ and $\mathtt{p}', \sigma', t' \Downarrow \mathit{stop}, \sigma''$ then $\mathtt{p}, \sigma, t \Downarrow \mathit{stop}, \sigma''$;
3. if $\mathtt{p}, \sigma, t \rightarrow \mathtt{p}', \sigma', t'$ and $\mathtt{p}', \sigma', t' \Downarrow \mathit{err}$ then $\mathtt{p}, \sigma, t \Downarrow \mathit{err}$;

Proof. Follows by induction over the rules concerning the small-step relation. □

Proposition 1. For all program $\mathtt{p}$ 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 $\mathtt{p}, \sigma, \mathtt{t} \rightarrow \mathit{skip}, \sigma', \mathtt{t}'$ then $\mathtt{p}, \sigma, t+s \rightarrow \mathit{skip}, \sigma', t'+s$. If $\mathtt{p}, \sigma, t \rightarrow \mathit{err}$ then $\mathtt{p}, \sigma, t+s \rightarrow \mathit{err}$

Proof. Follows straightforwardly by induction over the rules concerning the small-step relation and the algebraic properties of addition. □

(asg$^{\to}$) $x := e, \sigma, t \;\to\; \mathit{skip}, \sigma[x \mapsto [\![e]\!](\sigma)], t$

(asg-err$^{\to}$) $x := e, \sigma, t \;\to\; \mathit{err}$ (if $[\![e]\!](\sigma)$ undefined)

(diff-stop$^{\to}$) $\vec{x}' = \vec{\ell}\ \mathtt{for}\ e, \sigma, t \;\to\; \mathit{stop}, \sigma[\vec{x} \mapsto \phi(t)], 0$ (if $[\![e]\!](\sigma) > t$)

(diff-skip$^{\to}$) $\vec{x}' = \vec{\ell}\ \mathtt{for}\ e, \sigma, t \;\to\; \mathit{skip}, \sigma[\vec{x} \mapsto \sigma(t)], t - [\![e]\!](\sigma)$ (if $[\![e]\!](\sigma) \leq t$)

(diff-err$^{\to}$) $\vec{x}' = \vec{\ell}\ \mathtt{for}\ e, \sigma, t \;\to\; \mathit{err}$ (if $[\![e]\!](\sigma)$ undefined)

(if-true$^{\to}$) $\mathtt{if}\ b\ \mathtt{then}\ \mathtt{p}\ \mathtt{else}\ \mathtt{q}, \sigma, t \;\to\; \mathtt{p}, \sigma, t$ (if $[\![b]\!](\sigma) = \mathtt{tt}$)

(if-false$^{\to}$) $\mathtt{if}\ b\ \mathtt{then}\ \mathtt{p}\ \mathtt{else}\ \mathtt{q}, \sigma, t \;\to\; \mathtt{q}, \sigma, t$ (if $[\![b]\!](\sigma) = \mathtt{ff}$)

(if-err$^{\to}$) $\mathtt{if}\ b\ \mathtt{then}\ \mathtt{p}\ \mathtt{else}\ \mathtt{q}, \sigma, t \;\to\; \mathit{err}$ (if $[\![b]\!](\sigma)$ undefined)

(wh-true$^{\to}$) $\mathtt{while}\ b\ \mathtt{do}\ \{\ \mathtt{p}\ \}, \sigma, t \;\to\; \mathtt{p};\mathtt{while}\ b\ \mathtt{do}\ \{\ \mathtt{p}\ \}, \sigma, t$ (if $[\![b]\!](\sigma) = \mathtt{tt}$)

(wh-false$^{\to}$) $\mathtt{while}\ b\ \mathtt{do}\ \{\ \mathtt{p}\ \}, \sigma, t \;\to\; \mathit{skip}, \sigma, t$ (if $[\![b]\!](\sigma) = \mathtt{ff}$)

(wh-err$^{\to}$) $\mathtt{while}\ b\ \mathtt{do}\ \{\ \mathtt{p}\ \}, \sigma, t \;\to\; \mathit{err}$ (if $[\![b]\!](\sigma)$ undefined)

(seq-stop$^{\to}$) $\dfrac{\mathtt{p}, \sigma, t \;\to\; \mathit{stop}, \sigma', t'}{\mathtt{p};\mathtt{q}, \sigma, t \;\to\; \mathit{stop}, \sigma', t'}$ **(seq-skip$^{\to}$)** $\dfrac{\mathtt{p}, \sigma, t \;\to\; \mathit{skip}, \sigma', t'}{\mathtt{p};\mathtt{q}, \sigma, t \;\to\; \mathtt{q}, \sigma', t'}$

(seq-err$^{\to}$) $\dfrac{\mathtt{p}, \sigma, t \;\to\; \mathit{err}}{\mathtt{p};\mathtt{q}, \sigma, t \;\to\; \mathit{err}}$ **(seq$^{\to}$)** $\dfrac{\mathtt{p}, \sigma, t \;\to\; \mathtt{p}', \sigma', t'}{\mathtt{p};\mathtt{q}, \sigma, t \;\to\; \mathtt{p}';\mathtt{q}, \sigma', t'}$ (if $\mathtt{p}' \neq \mathit{stop}$ and $\mathtt{p}' \neq \mathit{skip}$)

Figure 3: Extension of the small-step operational semantics in [10] with the possibility of failure.

Theorem 2.2 (Equivalence)**.** The small-step semantics and the big-step semantics are related in the following manner. Given a program 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 $\mathtt{p}, \sigma, t \to^{\star} \mathit{stop}, \sigma', 0$;
3. $\mathtt{p}, \sigma, t \Downarrow \mathit{err}$ iff $\mathtt{p}, \sigma, t \to^{\star} \mathit{err}$.

Proof. The right-to-left direction is obtained by induction over the length of the small-step reduction sequence using Lemma 2.1. The left-to-right direction follows by induction over the big-step derivations together with Proposition 1. □

3 An Improved Simulator for Hybrid Programs

This section summarises several improvements made to Lince's simulator of hybrid programs since its original publication [11]. These include (1) more expressive assignments and differential statements (by virtue of the results in the preceding section); (2) a more user-friendly program syntax (by means of syntactic sugar); (3) more informative error messages; and (4) a numerical solver of systems of ordinary differential equations. In order to render our summary more lively we complement it with a running example involving an RLC circuit in series with an On-Off source. It is designed to stabilise voltage across the capacitor in the circuit at a specific value.

Running example: RLC circuits and harmonic oscillation. We present in Fig. 4 the simulation of an *RLC circuit in series* (RLCS). This simulation models an electric system composed of a resistor, a capacitor, an inductor, and a power source connected in series. The power source strategically switches on and off, as a way to stabilise voltage across the capacitor at a target value (say, $10V$). Such systems are known to yield interesting results that are practically relevant for energy storage voltage control systems, which help to mitigate voltage imbalances that could otherwise damage electronic equipment. More details about such circuits and associated differential equations are available for example in [30, 13]. We present in Fig. 4 two variations of an RLCS circuit: one in which the capacitor voltage is in an underdamped regime – with a resistance `rU` of 0.5Ω, a capacitance `c` of $0.047F$, and an inductance `l` of $0.047H$ – and one in which the capacitor voltage is in an overdamped regime – with a resistance `rO` of 4Ω and the same values as before for the capacitance and inductance. The general idea of our program is that the associated controller will read the voltage across the capacitor (variable `under` for the underdamped case, `over` for the overdamped one) every 0.01 seconds, and set the voltage at the source either to 0 (off) or $18V$ (on) depending on the value read.

```
under:=0; dU:=0; vU:=0; rU:=0.5;
over:=0;  dO:=0; vO:=0; rO:=4;
c:=0.047; l:=0.047;

while true do {
  if (under<10) then vU:=18;
                else vU:=0;
  if (over<10)  then vO:=18;
                else vO:=0;
  under'=dU, over'=dO,
  dU'=-(dU*rU/l)
      -under/(l*c)+vU/(l*c),
  dO'=-(dO*rO/l)
      -over/(l*c)+vO/(l*c)
  for 0.01;
}
```

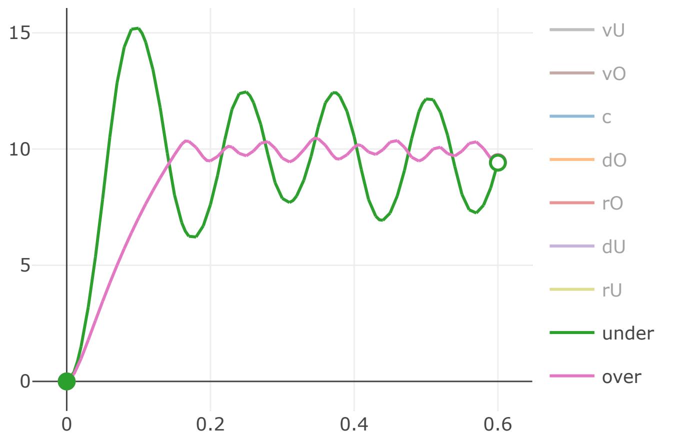


Figure 4: *Hybrid program (left) and its plot (right) of two variations of an RLC circuit that tries to maintain the voltage in the capacitor at* 10V

Improvement's summary. The program just described is highly problematic for the original version of Lince. This is due to two fundamental reasons related to the ODEs involved: specifically (1) the equations used in the ODEs violate the linearity condition presented in Section 2 (they include variable multiplications); and (2) the original solver of ODEs, mentioned in the introduction, fails to produce solutions after few iterations, due to the sheer, exponential growth of the involved expressions' size. We detail these issues and others next.

Richer expressions. As illustrated in the introduction and in the previous RLCS example, there are several essential, non-linear operations that are necessary to accomodate if one wishes to employ Lince in the analysis of diverse, common hybrid scenarios. We therefore now permit non-linear expressions outside of ODEs, essentially by using as basis the grammar of hybrid programs that was described in Section 2. Thus expressions outside the ODEs can now include for example the operations: division and multiplication of variables, more complex mathematical functions (such as square root extraction, exponentials, logarithms, minimum/maximum, and (co)sine), and mathematical constants (namely pi and Euler's constant). The full set of supported operations is detailed in Appendix A.

As for expressions inside ODEs, the linearity constraint is kept but the associated parser is much less rigid. A core feature is that it now tries to convert non-linear expressions into linear ones via algebraic laws. For example, it converts the non-linear expression $x \cdot 5$ into the linear one $5 \cdot x$. Most notably, it converts the non-linear expression $x \cdot y$ into a scalar multiplication $s \cdot y$ if it detects that x is constant and its value is s. Such a feature is in fact critical in our RLCS example, where we multiply variables in the respective ODEs.

More informative error messages. Several errors were undetected at an early stage of the simulation process, which resulted in unintelligible error messages in many situations. We thus added and improved the detection and notification of several key errors occurring in typical usages of Lince, including when: (1) a partial function fails (such as in division by 0); (2) a variable is not properly initialised; (3) the number of arguments of a function is incorrect; (4) the solver fails to solve a system of ODEs; and (5) ODEs contain non-linear expressions after de-sugaring. For example, in our RLCS simulation when defining `c` to be 0 we now obtain the error "*Error: the divisor of the division 'rU/(c)' is zero.*". In our experience, this more precise detection and notification of errors drastically improved user experience.

Numerical solver. As already mentioned, several hybrid programs such as our RLCS example cannot be properly handled by the (exact) solver of ODEs (*viz.* SageMath [28]) used by Lince. We have therefore implemented an alternative, numerical solver based on the popular fourth-order Runge-Kutta method [20]. At the theoretical level, this only required a small adaptation of the operational semantics presented in Section 2. Specifically we no longer assume that the solution $\phi_\sigma^{\vec{x}'=\vec{\ell}}$ associated to a system of ODEs $\vec{x}' = \vec{\ell}$ and an initial condition σ is exact. At the practical level, this allowed us to keep the size of expressions involved in computations highly manageable thus allowing Lince to cover a broader range of examples such as the RLCS.

4 An Improved Visualiser for Hybrid Programs

Many hybrid programs cannot be easily understood by simply plotting values of variables over time. For example, in some cases one may wish to analyse the movement of a vehicle in a 2D plane, or to analyse how its behaviour varies due to changes in its initial position and velocity. This section presents an extension of Lince's visualisation capabilities in these two directions. In the same spirit of the preceding section, we complement our description with a running example.

Running example: avoiding and manoeuvring around obstacles. The *Automatic Emergency Braking* (AEB) system is an autonomous driving device that after reading its distance to an obstacle and its current velocity, decides whether to decelerate until stopping [1]. Here we present a more advanced version of the AEB that after stopping also manoeuvres around the obstacle – clearly a process involving two or even three spatial dimensions. Such a system is called *Automatic Emergency Braking with an Overtaking Manoeuvre* (AEBOM).

The continuous dynamics of the AEBOM (*i.e.* the differential equations involved) is typically given by Dubins dynamics which essentially describe the object's orientation over time (an angle) and its effect on the object's velocity along the different spatial dimensions [26]. We adopt this approach as well. For simplicity we additionally assume that our object is a robot that is able to rotate around itself. The overall process of our AEBOM is thus as follows: move forward until detecting the obstacle and in which case decelerate until stopping; then rotate to the left and move forward a prescribed number of meters (that depends on the obstacle's size); then rotate right and move forward again a prescribed number of meters; and finally repeat the last step.

Fig. 5 depicts the original visualisation of the AEBOM simulation on the left, and a customised 2D visualisation that uses our extension on the right. The respective implementation of the AEBOM (found in **??**) is not relevant to show at this stage, because our focus is at the moment on describing new visualisation mechanisms and not features concerning code. Observe as well that the plot on the right provides novel insightswith respect to the one on the left: whilst in the former it is clear that the robot does not collide with the obstacle and performs the overtaking manoeuvre safely, in the latter it is much harder to see that the same occurs. We provide more details about our improved plotting system next.

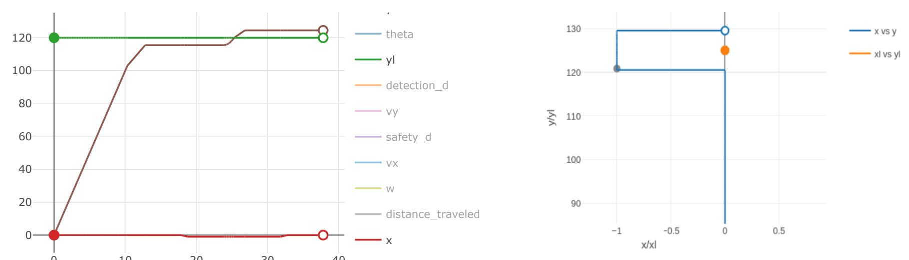


Figure 5: Plot of our AEBOM using the traditional plotting system in Lince on the left, and a new customised 2D plot relating `x` with `y` (the robot's coordinates) and `xl` with `yl` (the obstacle's coordinates) on the right.

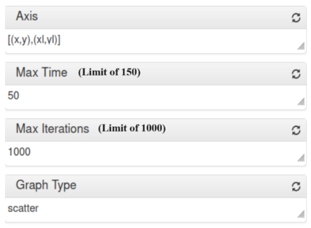
Higher-dimensional trajectories and beyond. Our new visualisation framework in Lince uses the Plotly JavaScript library to display plots[2]. Among other things, we now support 2D and 3D scatter plots, and include dedicated markers such as the large circles indicating the start and end points of trajectories (visible in all plots in this paper). When hovering over these markers, extra information is displayed, *e.g.* the respective values, relevant information about the conditionals involved, and potential warnings. We also exploit the animation functionality of Plotly in plots that do not include the time component, by moving a highlighting circle through the trajectories capturing how values vary throughout time. This is feature is active by default. To take all these possiblities into account, Lince allows the user to adjust different settings of the plot under analysis so that she can obtain the best possible configuration for her needs. We very briefly detail such settings next:

- *Axis*: Allows defining the relationships between variables which will automatically be presented in the respective plots. For example, by setting `[x, y, v]`, if the graph type is scatter, three separate graphs will be generated where the vertical axis represents each of the variables `x`, `y`, and `v`, while the horizontal axis represents time. Choosing which variables to map to the axes is crucial for proper data analysis, allowing direct visual comparisons between different variables over time or with each other.

[2] http://plotly.com/

- *Max Time*: Refers to the duration of the simulation.
- *Max Iterations*: Specifies the maximum number of iterations (in while-loops) that the simulation can perform.
- *Graph Type*: Defines the type of graph to be used for visualising the simulation data, by selecting from the available types ('scatter' or 'scatter3d'). In a nutshell, a scatter plot is a 2D graph used to display the relationship between two variables, with data points plotted in the two-dimensional plane. Scatter3D serves the same purpose but involves three variables, with data points plotted in the three-dimensional space.

The summarised settings are presented in Fig. 6, where the values there listed are the ones used to obtain the plot in Fig. 5 on the right.



Axis
[(x,y),(xl,yl)]

Max Time (Limit of 150)
50

Max Iterations (Limit of 1000)
1000

Graph Type
scatter

Figure 6: Input boxes that allow for the configuration of the visualisation.

Variability of initial conditions. As mentioned before, it is highly relevant take into account how the behaviour of a hybrid program varies due to changes in its initial conditions. In the AEBOM previously described in particular, it is of fundamental importance to understand how the robot manoeuvres around an obstacle w.r.t. to different initial positions and velocities – for it is unrealistic to expect that it moves with well-known, exact conditions. A similar, more general discussion can be consulted in [26].

In order to address this aspect we extended Lince in two steps: first its syntax now allows the listing of different initial conditions at the same time. Such is illustrated in Fig. 7 on the left, with a snippet of code used to specify initial values w.r.t. our robot in the AEBOM example. The latter's initial position (x,y) for example, can now be either (0,0), (2,0), or (4,0); and similarly we have different initial velocities (vy) towards the obstacle, 4, 8, and 12 $^m/_s$. Second Lince now pre-processes such listings in the code and derives all possible combinations of initial conditions, which of course yields several hybrid programs at once (in the standard syntax). These data is then fed into Lince's visualiser which presents multiple simulations overlapped in the same plot. Such is seen in Fig. 7 on the right, again with our AEBOM example, where we see that our robot behaves in the same way under different initial conditions.

```
//Initial Conditions
x  :=[0,2,4];
y  :=  0;
vx :=  0;
vy :=[4,8,12];
xl :=[0,2,4];
yl :=[120,135,150];

...
```

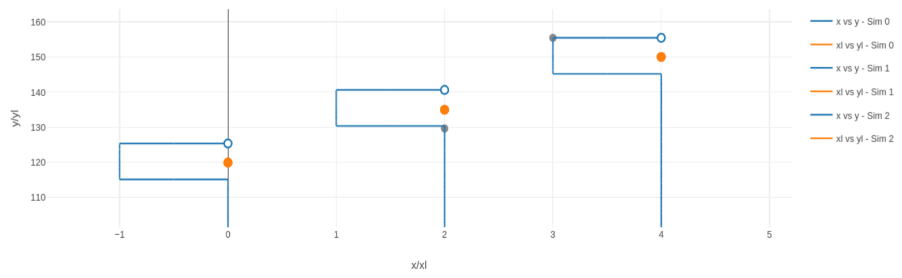


Figure 7: Visualisation of multiple simulations overlapped concerning the AEBOM.

5 Lince at Work: a showcase of the overall list of improvements

This section illustrates the overall list of improvements made to Lince (as described in the preceding sections) working together in the design and analysis of a complex hybrid scenario – specifically we focus on a multi-dimensional pursuit game between two players (for example two drones) [21, 2, 5, 18]. Our illustration incides on two aspects: Lince's capacity to actually simulate such scenarios, with optimally configured 3D plotting systems; Lince's time performance w.r.t. simulations of this kind, as a way to attest its scalability to even more complex systems.

Pursuit Games. Pursuit games are a captivating class of problems involving multiple agents, where at least one them (the pursuer) aims to capture or reach another (the evader) [21, 2, 5, 18, 26]. Such games are extensively studied across various disciplines, including mathematics, game theory, robotics, and computer science, due to their practical and theoretical significance. Indeed they model a wide range of real-world situations, from military and security operations to animal behaviour and industrial applications.

In this section we explore a specific 3D pursuit game, where we perceive the pursuer as a drone that attempts to capture another one in the three-dimensional space. This scenario is particularly challenging, due to the additional complexity introduced by the third dimension which requires a higher level of planning and coordination between the drones' movements. In order to model this problem we base our game's continuous dynamics on Dubins dynamics [26], *i.e.* as in Section 4 but now in three dimensions.

Our overarching strategy for the pursuer is to simply point its orientation to the evader's position at every iteration in a certain while-loop. Of course there are other options, such as that of (variations of) *Dubins paths* [26, 4], but our version already suffices to properly illustrate Lince at work. Technically our approach utilises the angular velocity tensor to perform 3D infinitesimal rotations [6]. Additionally we use the cross product between the projection of the relative velocity vector and the relative position vector in each plane to determine the orientation of rotation among the three axes. We do not show here the coding details of all these processes, since this is unnecessary for our illustration. However the interested reader can consult details about these in [4, 6], and furthermore the complete code of our program is presented in Appendix B.2.

We now show the simulation of our game in Lince across different scenarios. In the first case, the pursuer starts from the position `(300,300,600)` with a velocity of `(-20,-10,0)`$^{m}/_{s}$, while the evader begins at the position `(600,600,500)` with a velocity of `(10,0,10)`$^{m}/_{s}$. The pursuer's angular velocity along each axis is `(1/20)*2*pi()`$^{rad}/_{s}$ (20 seconds to complete a full rotation); and for the evader `(1/40)*2*pi()`$^{rad}/_{s}$ (40 seconds to complete a full rotation). The pursuer is allowed to actuate every 0.1s, and it wins the game if it reaches a distance of less than one meter w.r.t. the evader. Finally, for simplicity we assume a predefined set of movements for the latter player. Using these parameters, we simulated the corresponding program in Lince and generated a 3D scatter plot of the positional variables for both the pursuer and the evader, resulting in the graphical representation shown in the Fig. 8 after 73 seconds.

We can see that the decision strategy for the pursuer adopted in this hybrid program successfully guided it to the evader, resulting in a capture at the position `(691.26,441.92,561.12)` after 27.7 seconds. However if we change the initial velocity of the evader to a higher value, such as `(20,0,9)`$^{m}/_{s}$, we obtain worse results as depicted in Fig. 9: actually in this case we never see the pursuer capturing the evader, for the simulation is forced to stop before that. In other words, Lince yields a timeout which highlights that the tool still has some constraints performance-wise. Of course since the magnitude of the evader's velocity is lower than that of the pursuer's, a capture is theoretically possible.

Finally by taking advantage of the variability results presented in Section 4 we very briefly study the effects of angular velocity in this pursuit game. Specifically we adjust the angular velocity of the pursuer

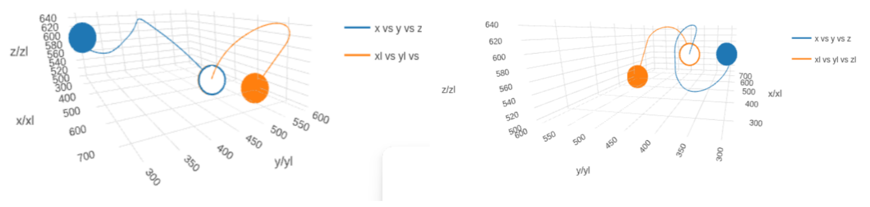


Figure 8: The figure presents two plots for the same initial conditions. The blue trajectory corresponds to the trajectory of the pursuer and the orange trajectory corresponds to the trajectory of the evader.

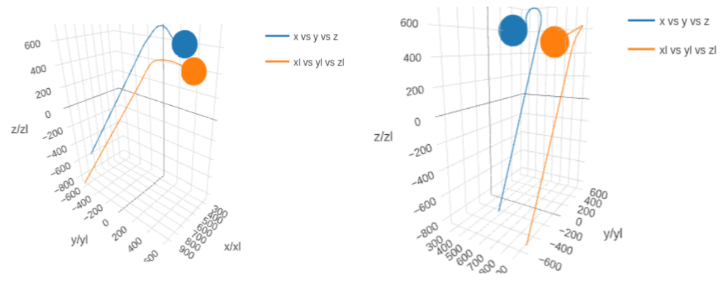


Figure 9: The figure presents two plots for the same initial conditions. The blue trajectory corresponds to the trajectory of the pursuer and the orange trajectory corresponds to the trajectory of the evader. In this case we do not actually see the pursuer capturing the evader.

along each axis to be either `(1/40)*2*pi()`$^{rad}/_{s}$ or `(1/100)*2*pi()`$^{rad}/_{s}$, whilst maintaining the other initial conditions from the first scenario. The resulting graphical representation (after 220 seconds) is shown in Fig. 10. From the graphical representations we observe that the pursuer successfully captures the evader when the angular velocity is `(1/40)*2*pi()`$^{rad}/_{s}$ at the position `(692.07,415.62,464.63)` in 34.8 seconds. However with an angular velocity of `(1/100)*2*pi()rad/s`, the simulation cannot present a capture.

These simulations showcase Lince's ability to model and simulate complex scenarios, thus providing valuable insights into a system's behavior.

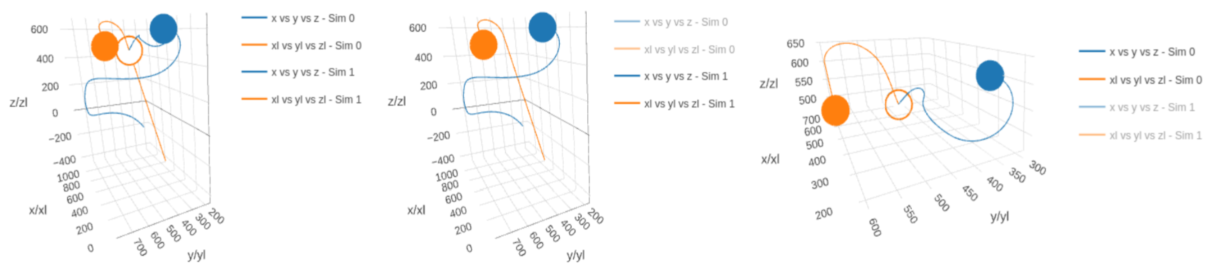


Figure 10: Multiple simulations which result from varying the pursuer's angular velocity. Specifically we see two simulations (the leftmost and rightmost plot correspond to the same simulation but under different perspective). The first (Sim 0) corresponds to the angular velocity of `(1/40)*2*pi()`$^{rad}/_{s}$, and the second (Sim 1, *i.e.* plot in the middle) has velocity `(1/100)*2*pi()`.

A brief overview of Lince's time performance. As shown in the previous example, Lince still has a few limitations concerning performance. In order to give the reader a more concrete overall idea of them we provide next an overview of how Lince fares perfomance-wise against the examples presented in this paper. First we give further context on how Lince operates.

The first main observation is that now that Lince is equipped with an effective numerical solver (recall Section 3) it can operate in two starkly different ways: one analytical with exact methods that rely on SageMath's framework [28], the other numerical, based on progressively closer approximations as described in Section 3. Both operation modes have significant differences performance-wise: most notably the former is obviously slower and gives timeouts much more frequently than the latter (recall our RLCS example in Section 3). Interestingly the bottleneck hinges not only on the employment of a precise solver, but also on the fact that:

1. this solver is external to Lince, specifically our tool needs to interact with a server, with all the usual delays that this implies;
2. along the evaluation of a hybrid program, Lince needs to simplify resulting expressions over and over to make them tractable (due to them being symbolic and not numerical).

We saw first-hand in Section 3 how all these extra tasks running behind the curtains inhibit Lince to simulate programs such as the RCLS circuit. The numerical solver, on the other hand, avoid these problems, but at the cost of less precision which may have deep implications if one wishes to have full guarantees that a simulation is correct, particularly if the system at hand is chaotic [24]. Needless to say, to find methods that have the virtues of both approaches is a very interesting challenge.

Table 1 lists several execution times of Lince against different variations of the examples presented in the paper. The columns 'Sampling time' and 'Nº of Iterations' refer respectively to the rate at which computational tasks need to be performed and the total number of times the while-loop in the program involved is unfolded. The columns 'Time Symb-Server' and 'Time Symb-Total' refer respectively to the time spent by Lince in communications with the server, when using exact methods, and the respective total time needed for presenting the plot. The column 'Time Numerical-Total' refers to the time taken to present the when using the numerical solver. Concerning rows, we basically list all examples explored in the paper with the proviso that AEB stands only for the first part of the AEBOM, specifically the task of stopping before hitting an obstacle. This table yields several relevant remarks, even if some are non-surprising:

- The AEB example is the only case in which exact methods can be used successfully. This is because it uses a very simple system of differential equations, which the exact solver can easily handle.
- Also in the AEB example we observe that when using exact methods, server processing accounts for approximately 99% of the total time spent on producing the respective simulation.
- The numerical mechanisms in the AEB example yield simulations significantly faster than in the exact counterpart, as expected.
- The total times taken to numerically simulate the RLCS and AEBOM examples are shorter than in the Pursuit Games example. This is because these two examples involves fewer computations and the Pursuit Games case uses a 3D scatter plot, which is of course more computationally intensive than the 2D scatter plot.
- Larger sampling times imply reduced times in generating both the exact and numerical plots, due to the decreased number of computational operations. However, this comes at the cost of lower

Table 1: An overview of Lince's time performance w.r.t. the examples discussed in this paper. We consider different sampling times, number of iterations, and both exact and approximate methods.

	Sampling Time	**Nº of Iterations**	**Time Symb-Server**	**Time Symb-Total**	**Time Numerical-Total**
RLCS	0.01s	1000	-	-	11.46s
	0.1s	1000	-	-	10.98s
	1s	150	-	-	1.14s
AEB	0.01s	184	23.56s	23.70s	0.41s
	0.1s	19	13.04s	13.08s	0.18s
	1s	2	11.90s	11.97s	0.14s
AEBOM	0.01s	1000	-	-	8.85s
	0.1s	128	-	-	0.62s
	1s	21	-	-	0.35s
Pursuit Games	0.01s	1000	-	-	66.60s
	0.1s	322	-	-	18.26s
	1s	150	-	-	7.85s

capacity of controllers to actuate on physical processes such as movement, velocity, and time. Such is critical for example in the context of autonomous driving and embedded systems in general.

6 Conclusion and future work

We presented an improved version of Lince, which can now handle a much broader class of hybrid programs whilst at the same time providing a better user experience overall. As previously discussed, this required an extension, with the possibility of failure, of the operational semantics introduced in [10], and the implementation of an efficient numerical solver, among other things.

We believe that our work opens up several research paths that we would like to explore next. For example, thanks to the numerical solver it is now straightforward to extend our language with non-linear differential equations, which widens even more the range of programs that Lince can currently tackle. Another interesting research path is the addition of probabilistic constructs to Lince, such as measure sampling. We conjecture that this could be handled easily in Lince via a random-number generator and part of the implemented variability mechanisms that were presented in Section 4.

References

[1] Proctor Acura: *Technology Guide: What is an Automatic Braking System?* https://www.proctoracura.com/automatic-braking-system-guide.

[2] Tom Anderson, Rogério de Lemos, John S Fitzgerald & Amer Saeed (1993): *On formal support for industrial-scale requirements analysis*. Springer.

[3] Davide Bresolin, Luca Geretti, Tiziano Villa & Pieter Collins (2015): *An introduction to the verification of hybrid systems using ARIADNE*. *Coordination Control of Distributed Systems*, pp. 339–346.

[4] Xuan-Nam Bui, J.-D. Boissonnat, P. Soueres & J.-P. Laumond (1994): *Shortest path synthesis for Dubins non-holonomic robot*. In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pp. 2–7 vol.1, doi:10.1109/ROBOT.1994.351019.

[5] Zhou Chaochen, Anders P Ravn & Michael R Hansen (1993): *An extended duration calculus for hybrid real-time systems*. Springer.

[6] Garanin Dmitry (2008): *Rotational motion of rigid bodies*. `https://www.lehman.edu/faculty/dgaranin/Mechanics/Mechanis_of_rigid_bodies.pdf`.

[7] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang & Oded Maler (2011): *SpaceEx: Scalable Verification of Hybrid Systems*. In: *23rd International Conference on Computer Aided Verification (CAV)*, Springer.

[8] Peter Fritzson (2014): *Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach*. John Wiley & Sons.

[9] Sergey Goncharov & Renato Neves (2019): *An Adequate While-Language for Hybrid Computation*. In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*, PPDP '19, ACM, New York, NY, USA, pp. 11:1–11:15.

[10] Sergey Goncharov, Renato Neves & José Proença (2020): *Implementing Hybrid Semantics: From Functional to Imperative (Extended Version)*. Technical Report, CISTER-Research Centre in Realtime and Embedded Computing Systems.

[11] Sergey Goncharov, Renato Neves & José Proença (2019): *Lince: Lightweight Prototyping of Hybrid Programs (full version)*. Technical Report, doi:10.5281/zenodo.3518838.

[12] Volkan Gunes, Steffen Peter, Tony Givargis & Frank Vahid (2014): *A survey on concepts, applications, and challenges in cyber-physical systems*. *Transactions on Internet and Information Systems* 8(12), pp. 4242–4268.

[13] Ahammodullah Hasan, Md Abdul Halim & MA Meia (2019): *Application of linear differential equation in an analysis transient and steady response for second order RLC closed series circuit*. *American Journal of Circuits, Systems and Signal Processing* 5(1), pp. 1–8.

[14] Peter Höfner (2009): *Algebraic calculi for hybrid systems*. Ph.D. thesis, University of Augsburg. Available at `http://opus.bibliothek.uni-augsburg.de/volltexte/2010/1481/`.

[15] Eduard Kamburjan, Stefan Mitsch & Reiner Hähnle (2022): *A hybrid programming language for formal modeling and verification of hybrid systems*. *Leibniz Transactions on Embedded Systems* 8(2), pp. 04–1.

[16] Harold Klee (2007): *Simulation of dynamic systems with MATLAB and Simulink*. CRC Press.

[17] Dexter Kozen (1997): *Kleene Algebra with Tests*. *ACM Transactions on Programming Languages and Systems* 19(3), pp. 427–443.

[18] Thomas Krilavicius (2005): *Bestiarium of Hybrid Systems*. draft, Mar.

[19] Edward A. Lee & Sanjit A. Seshia (2016): *Introduction to embedded systems: A cyber-physical systems approach*. MIT Press.

[20] Gabrielle Maioli (2015): *Métodos numéricos para equações diferencias ordinárias*.

[21] Zohar Manna & Amir Pnueli (1993): *Verifying hybrid systems*. In: *Hybrid Systems*, Springer, pp. 4–35.

[22] Renato Neves (2018): *Hybrid programs*. Ph.D. thesis, Minho University.

[23] E-R Olderog (1992): *Nets, terms and formulas: three views of concurrent processes and their relationship*. Cambridge University Press.

[24] Lawrence Perko (2013): *Differential equations and dynamical systems*. 7, Springer Science & Business Media.

[25] André Platzer (2010): *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg.

[26] André Platzer (2018): *Logical Fundations of Cyber-Physical Systems*. In: *Logical Fundations of Cyber-Physical Systems*, Springer, pp. 85–87.

[27] John C Reynolds (1998): *Theories of programming languages*. Cambridge University Press.

[28] W. A. Stein et al. (2015): *Sage Mathematics Software (Version 6.4.1)*. The Sage Development Team. `http://www.sagemath.org`.

[29] Glynn Winskel (1993): *The formal semantics of programming languages: an introduction*. MIT press.

[30] Yue Zhang & Anurag Srivastava (2021): *Voltage Control Strategy for Energy Storage System in Sustainable Distribution System Operation*. *Energies* 14(4), doi:10.3390/en14040832. Available at `https://www.mdpi.com/1996-1073/14/4/832`.

A Functions supported by Lince

Lince supports the arithmetic operations in Table 2, the mathematical functions in Table 3, and the mathematical constants in Table 4.

Operation	Instruction
Division	/
Multiplication	*
Addition	+
Subtraction	-
Remainder	%

Table 2: *Arithmetic operations supported by Lince.*

Function	Instruction	Function	Instruction	Function	Instruction	Function	Instruction
Power	pow(..) or ^	**Minimum**	min(...,...)	**Arcsine**	arcsin(...)	**Hiperbolic cosine**	cosh(...)
Square root	sqrt(..)	**Sine**	sin(...)	**Arccosine**	arccos(...)	**Hiperbolic tangent**	tanh(...)
Exponen-tiation	exp(...)	**Cosine**	cos(...)	**Arctangent**	arctan(...)	**Logarithm**	log(...)
Maximum	max(...,...)	**Tangent**	tan(...)	**Hyperbolic sine**	sinh(...)	**Base-10 logarithm**	log10(...)

Table 3: *Mathematical functions supported by Lince.*

Constant	Instruction
Pi	pi()
Euler	e()

Table 4: *Mathematical constants supported by Lince.*

B Full code of examples in the main text

B.1 AEBOM

```
x:=0; y:=0;vx:=0; vy:=10;xl:=0; yl:=120;
detection_d:=100;
safety_d:=5;
a:=4;
theta:=0;
w:=(1/20)*2*pi();

//Obstacle -> 1m by 1m

while (y + vy*0.1-yl > detection_d) do {
y'=vy,vy'= 0 for 0.1;
}

stoping_time:=vy/a;
distance_traveled:=vy * stoping_time + 0.5 * (-a) * stoping_time^2;

while (yl - y > safety_d + distance_traveled) do {
y'=vy,vy'= 0 for 0.1;
}
```

```
y'=vy,vy'=-a for stoping_time;
vy:=0;

w:=-w;
theta'=w for (pi()*0.5)/(-w);

x'=vx,vx'=-a for sqrt(1/a);
x'=vx,vx'=a for sqrt(1/a);
vx:=0;

w:=-w;
theta'=w for (pi()*0.5)/w;

while (y<yl) do {
y'=vy,vy'=a for 0.1;
}

stoping_time:=vy/a;
y'=vy,vy'=-a for stoping_time;
vy:=0;

theta'=w for (pi()*0.5)/w;

x'=vx,vx'=a for sqrt(1/a);
x'=vx,vx'=-a for sqrt(1/a);
vx:=0;

w:=-w;
theta'=w for (pi()*0.5)/(-w);
```

B.2 Pursuit Games

```
// Initial position and velocity of the pursuer
x := 300; vx := -20;
y := 300; vy := -10;
z := 600; vz := 10;

// Initial position and velocity of the evader
xl := 600; vxl := 10;
yl := 600; vyl := 10;
zl := 500; vzl := 0;

// Angular velocity of the pursuer
aw_x := (1/20) * 2 * pi();
aw_y := (1/20) * 2 * pi();
aw_z := (1/20) * 2 * pi();

// Angular velocity of the evader
awl_x := (1/40) * 2 * pi();
awl_y := (1/40) * 2 * pi();
awl_z := (1/40) * 2 * pi();

// Counter
cont := 0;

// Decision time
sampling_time := 0.1;

// Minimum collision distance
dist_min_col := 1;
```

```
// Vectorial product of each axis
vect_P_x := 0;
vect_P_y := 0;
vect_P_z := 0;

// Variables that stores the angular velocity decision
w_x := 0;
w_y := 0;
w_z := 0;
wl_x := 0;
wl_y := 0;
wl_z := 0;

//Variables that stores the relative positions and velocities
dx := 0;
dy := 0;
dz := 0;
vrelx := 0;
vrely := 0;
vrelz := 0;

// Run the following programme whilst the distance between the pursuer and the evader
    is greater than
//the collision distance
while (sqrt((x-xl)^2 + (y-yl)^2 + (z-zl)^2) > dist_min_col) do {

    //Conditional structures to establish the evader path
    if (cont <= 100) then {
        wl_x := 0;
        wl_y := 0;
        wl_z := 0;
    } else {
        if (cont <= 200) then {
            wl_x :=  awl_x;
            wl_y := 0;
            wl_z := -awl_z;
        } else {
            if (cont <= 300) then {
                wl_x := 0;
                        wl_y := -awl_y;
                        wl_z := -awl_z;
            } else {
                wl_x := 0;
                                        wl_y := 0;
                                        wl_z := 0;
            }
        }
    }

    // The counter is incremented
    cont := cont + 1;

    //Update distances and relative velocities
    dx := xl - x;
    dy := yl - y;
    dz := zl - z;
    vrelx := vxl - vx;
    vrely := vyl - vy;
    vrelz := vzl - vz;

    // Determine the value of the vetorial product between the relative velocity
        vector and the relative position vector for each axis
    vect_P_x := vrely * dz - vrelz * dy;
    vect_P_y :=  vrelz * dx - vrelx * dz;
    vect_P_z := vrelx * dy - vrely * dx;

    // Curve along the Z axis
    if (vect_P_z >= 0)
    then {w_z := -aw_z;}
    else {w_z := aw_z;}

    // Curve along the X axis
    if (vect_P_x >= 0)
    then {w_x := -aw_x;}
    else {w_x := aw_x;}

    // Curve along the Y axis
    if (vect_P_y >= 0)
    then {w_y := -aw_y;}
    else {w_y := aw_y;}

    // Differential equations
    x' = vx,y' = vy,z' = vz,vx' = w_y * vz - w_z * vy,vy' = w_z * vx - w_x * vz,vz' =
        w_x * vy - w_y * vx,
    xl' = vxl,yl' = vyl,zl' = vzl,vxl' = wl_y * vzl - wl_z * vyl,vyl' = wl_z * vxl -
        wl_x * vzl,vzl' = wl_x * vyl - wl_y * vxl for sampling_time;
}
```