Towards Formal Verification of Hybrid Synchronous Programs with Refinement Types

Serra Z. Dane[1], Jiawei Chen[1], Marc Pouzet[2], and Jean-Baptiste Jeannin[1]

[1] University of Michigan, Ann Arbor, MI, USA
{sdane, chenjw, jeannin}@umich.edu
[2] ENS, PSL University and INRIA, Paris, France
marc.pouzet@ens.fr

Abstract. Cyber-physical systems (CPS) such as autonomous cars, aircraft, and robots are often also safety-critical; thus it is imperative that they operate as intended with a high degree of certainty. Formal verification has been employed to verify the software controlling these systems, but due to their complexity, is usually performed on an abstract model rather than the executable code. Synchronous programming languages extended with differential equations promise both rigorous modeling and sufficient expressiveness to implement executable controller code, and recent developments have introduced formal verification of strictly discrete-time programs. Extending these verification techniques to hybrid systems enables precise modeling of the environment for a wider variety of programs to be both verified and executed. We formalize the operational semantics of initial value problems and zero-crossing detection expressed in a synchronous programming language, extend its type system for verification thereof, and prove its soundness.

Keywords: Hybrid Systems · Synchronous Programming · Type Theory · Semantics · Cyber-Physical Systems.

1 Introduction

Cyber-physical systems (CPS), systems in which software operates in a physical environment, are fundamental for modeling real-world systems such as robots and embedded systems. Because of their physical effects, the software running these systems is often safety-critical. Rigorous techniques such as formal methods are preferred over testing due to the complex interactions present. While formal verification is typically done on an abstract mathematical model of the system, direct verification of the executable code controlling a CPS provides a more convincing result. This necessitates the development of semantics that can accurately characterize the program. However, this is complicated by the fact that CPS are often hybrid: the controller runs in discrete, synchronous steps, while the physical environment evolves continuously (often modeled by ODEs), so the overall behavior mixes discrete updates with continuous-time dynamics. A promising direction is therefore to bring verification and implementation closer

together in a single programming language with precise hybrid semantics, so that verified properties directly correspond to the behavior of executable controller code. The programming language for writing models must be mathematically precise (that is, with unambiguous semantics and implementation) such that all validation steps can be performed on the source, to give confidence on the model itself and to automatically transfer some of the properties to the implementation.

Synchronous programming languages provide a strong foundation for this goal such as Zélus [9], Lustre [10], Esterel [8], Signal [19] and SCADE [13]. Their stream-based, deterministic model of computation matches the structure of many control programs, and their discipline of logical time aligns well with embedded execution. Moreover, modern synchronous languages such as Zélus extend this discrete-time core with continuous-time dynamics expressed as ordinary differential equations (ODEs), enabling programmers to model hybrid CPS in a unified setting where discrete control logic coexists with continuous evolution [9]. In such hybrid synchronous programs, the connection between discrete and continuous behavior is often mediated by zero-crossings: discrete events occur when a guard function over the continuous state crosses zero, triggering resets or mode changes. Zero-crossings are thus central to both modeling and execution.

Recent work has shown that refinement types can be used to verify safety properties of discrete-time synchronous programs directly in the source language. Invariants are encoded as type refinements and discharging the resulting proof obligations with SMT solving [12]. However, extending this approach from discrete-time streams to hybrid programs raises fundamental semantic and proof-theoretic challenges. Continuous behavior exists outside of discrete-time computation so safety must be shown for both continuous and discrete time, and zero-crossings need to be defined precisely enough to support a sound theory. At the same time, continuous-time verification benefits from proof principles that differ from purely discrete induction; in particular, differential invariants, as developed in differential dynamic logic, provide a well-established way to prove that a property is preserved along ODE solutions [25].

This paper develops a refinement-type-based verification framework for hybrid synchronous programs that supports execution while providing formal guarantees against safety specifications. We present the following contributions:

1. A formal, modular definition of zero-crossings for hybrid synchronous programs.
2. Operational semantics and typing rules for verifying a synchronous programming language with continuous dynamics and event-triggered resets based on the hybrid synchronous language Zélus.
3. A definition and proof of type safety for the hybrid synchronous refinement type metatheory.

2 Preliminaries

In this section, we present the preliminaries required for the development of our theory.

2.1 Synchronous Programming

Synchronous languages model computations as *streams* whose values evolve over *logical time* [10,8,19,13]. A program reacts in discrete logical instants (clock cycles), producing a value for each stream at every instant, and all computations within an instant must complete before the next instant begins. This yields a deterministic model of computation that is well suited to control software and embedded systems where predictability is essential. Under the *synchronous hypothesis*, each reaction is treated as instantaneous at the level of logical time, providing deterministic interaction points with the environment [7]. This discipline also supports modular compilation and compositional reasoning about stream-based controllers [6]. These properties make synchronous programming a natural foundation for hybrid systems: discrete controller updates occur at logical instants, while continuous plant evolution (and event detection) can be analyzed over the real-time intervals between reactions.

2.2 Hybrid Systems

Hybrid systems combine *discrete* control with *continuous* physical dynamics. A standard mathematical model is the *hybrid automaton*: a finite set of modes, an ODE (flow) associated with each mode, and guarded discrete transitions that may reset the continuous state [1,15]. Executions alternate between continuous evolution that follows the active mode's ODE and instantaneous jumps that change mode and/or apply a reset. This model is well suited to CPS: controllers update at discrete instants (e.g., periodic sampling and actuation), while the plant evolves in continuous time between updates. In this work we adopt a hybrid *synchronous* programming perspective via Zélus, which extends a Lustre-style synchronous language with ODE-based continuous evolution and event-triggered resets, allowing discrete control and continuous dynamics to be expressed and executed within a single source language [5,9].

2.3 Differential Dynamic Logic (dL)

Our type system relies on proof rules for reasoning about continuous evolution in the style of Platzer's *differential dynamic logic* (dL) [26]. In dL, an ODE evolution has the form

$$\dot{x} = f(x) \;\&\; Q(x),$$

where $\dot{x}$ denotes the time derivative of the state $x(t)$ along a trajectory, i.e., $\dot{x}(t) = \frac{d}{dt}x(t)$. Here the state variable x evolves continuously according to the vector field f for an arbitrary duration, *restricted* to times during which the evolution-domain constraint Q holds. Modal formulas $[\alpha]\phi$ (resp. $\langle\alpha\rangle\phi$) state that ϕ holds after all (resp. some) terminating runs of program α; we consider only the former case.

A central proof technique we adopt is the *differential invariant* rule. Informally, to prove that a predicate P holds throughout an ODE evolution restricted

to Q, it suffices to show that P is preserved differentially while Q holds:

$$\frac{Q \vdash [\dot{x} := f(x)]\dot{P}}{P \vdash [\dot{x} = f(x)\ \&\ Q\,]\ P}\ (\text{DI})$$

Here $[\,\alpha\,]\phi$ means ϕ holds after all terminating runs of α, and $\dot{x} = f(x)\ \&\ Q$ denotes continuous evolution $\dot{x} = f(x)$ restricted to states satisfying Q. The formula $\dot{P}$ is the symbolic time derivative of P along the flow, and $\dot{x} := f(x)$ assigns the derivative symbol $\dot{x}$ to $f(x)$ to reduce checking $\dot{P}$ to an algebraic condition under Q. Note that this differential evaluation can stop even though Q does not become $false$, since in dL an ODE command is non-deterministic in its duration [26]. Unlike dL, where Q explicitly restricts the ODE and there is no determined duration of evaluation, adapting this rule to zero-crossings requires additional safeguards, established in subsequent sections.

2.4 Refinement Types

Refinement types strengthen a base type b with a logical predicate φ, written $\{v : b \mid \varphi(v)\}$, to specify subsets of values of b satisfying φ. For example, positive reals can be written $\{v : \mathsf{real} \mid v > 0\}$ [16]. Operationally, refinement typing supports generating verification conditions that can be discharged by automated solvers, while keeping the core language type system largely unchanged [29]. We use refinements to express *safety* properties of hybrid state variables and to encode inductive invariants that must hold across both continuous evolution and discrete resets.

2.5 Verification Via Typing

Verification via typing uses a type system as a lightweight, compositional proof discipline: a program that type-checks is guaranteed to satisfy a class of semantic safety properties, with the guarantee justified by a *type soundness* theorem relating typing to the language's operational semantics [32]. Liquid Types combine Hindley–Milner inference with predicate abstraction to automatically infer refinements strong enough to prove program invariants [29]. Other approaches internalize program logics into types: Hoare Type Theory integrates Hoare-style pre/postconditions into the type of effectful computations, enabling modular reasoning about stateful code within the type system [24]. Typing-based verification has also been explored in the context of synchronous programming, where refinement types are used to relate executable models and safety specifications within a unified language and metatheory [12]. However, this line of work focuses on a restricted fragment of the language limited to purely synchronous constructs, leaving open the challenge of handling richer hybrid features addressed in the present work.

3 Formal Definition of Zero-Crossings

In this section we present a formal definition of *zero-crossings* based on the observable behavior of real-valued guard functions. Intuitively, a zero-crossing occurs when a function $g(t)$ changes sign, i.e., it passes from negative to positive or vice versa. In hybrid systems modelers, however, such sign tests are typically not primitive semantic constructs. Conditions like $x \geq 0$ are implemented via an internal mechanism known as *zero-crossing detection* [11,3,20].

There does not appear to be a single standard formal definition for zero-crossings: candidate definitions based solely on sign change disagree on subtle behaviors near zero, leading to implementation-dependent behavior and ambiguity in formal reasoning [3]. Since Zélus depends on zero-crossings to trigger discrete computation, this definition is fundamental to all the subsequent formalism.

Notation. To make our results as general as possible, we introduce a new notation. For any $a \in \{-\infty\} \cup \mathbb{R}$ and $b \in \mathbb{R} \cup \{+\infty\}$, we write $(\!|a; b|\!)$ for:

- $(\!|a; b|\!) \triangleq [a; b]$ if $a \in \mathbb{R}$ and $b \in \mathbb{R}$;
- $(\!|a; b|\!) \triangleq (-\infty; b]$ if $a = -\infty$ and $b \in \mathbb{R}$;
- $(\!|a; b|\!) \triangleq [a; +\infty)$ if $a \in \mathbb{R}$ and $b = +\infty$;
- $(\!|a; b|\!) \triangleq (-\infty; +\infty)$ if $a = -\infty$ and $b = +\infty$.

Note that $(\!|a; b|\!) \subseteq \mathbb{R}$ and cannot include $-\infty$ or $+\infty$.

Basic hypotheses and setup. Throughout this section, let f be a function defined as **continuous** on $(\!|\ell; u|\!)$ with $\ell \in \{-\infty\} \cup \mathbb{R}$, $u \in \mathbb{R} \cup \{+\infty\}$ and $\ell \leq u$. We consider only continuous functions. Let $z \in (\!|\ell; u|\!)$ such that $f(z) = 0$. The goal is to determine **under which circumstances z should be considered a zero-crossing.** We limit ourselves to negative-to-positive zero-crossings; the opposite case is symmetric.

We first note that the function f might be identically zero on an interval of non-zero length around z. We would like to precisely define this segment, which we will write $(\!|a; b|\!)$. For this purpose, let us define a and b as:

- $a \triangleq \inf\{a \in (\!|\ell; u|\!) \mid \forall x \in (\!|a; z|\!], f(x) = 0\}$;
- $b \triangleq \sup\{b \in (\!|\ell; u|\!) \mid \forall x \in [z; b|\!), f(x) = 0\}$.

3.1 Case Distinction

We now have the following properties for a and b:

- $a \in \{-\infty\} \cup \mathbb{R}$ and $b \in \mathbb{R} \cup \{+\infty\}$;
- $\forall x \in (\!|a; b|\!), f(x) = 0$;
- $(\!|a; b|\!) \subseteq (\!|l; u|\!)$, i.e. $\ell \leq a \leq b \leq u$;
- if $a > l \geq -\infty$, then $\forall \varepsilon > 0, \exists x \in [a - \varepsilon; a), f(x) \neq 0$;
- if $b < u \leq +\infty$, then $\forall \varepsilon > 0, \exists x \in (b; b + \varepsilon], f(x) \neq 0$.

If $a = b$, the function f is only equal to zero at one point $z = a = b$, and non-zero right before and right after: we will say that the function is *passing* through $z = a = b$. If $a < b$, the function f is identically zero on $(\!|a; b|\!)$, we will say that the function f is *staying* on $(\!|a; b|\!)$.

With this definition, we identify 85 cases arising from various combinations of behaviors to the left and right of a candidate zero-crossing, as well as passing and staying, summarized in the extended version of the paper and Figure 1.

3.2 Desirable properties

1. If there exist $x, y \in (\!|\ell; u|\!)$ such that $x < y$, $f(x) < 0$ and $f(y) > 0$, then there exists a zero-crossing $z \in (x; y)$.
2. The function f should be strictly negative somewhere left of, and strictly positive somewhere right of a zero-crossing z, formally:
   $\exists x, y \in (\!|\ell; u|\!), x < z < u$, $f(x) < 0$ and $f(y) > 0$.
3. A passing case should be a zero-crossing if and only if its corresponding staying case is also a zero-crossing.

Another decision is which point should be the zero-crossing in the staying case where $a < b$. We would typically want the zero-crossing to be a or b, but which one is a matter of convention. For this paper, we will pick b (but the subsequent developments can easily be adapted if picking a).

In light of these considerations, we define zero-crossings as follows:

Definition 1. *$z \in \mathbb{R}$ is a zero-crossing for a continuous function f if and only if there exist $a \in \mathbb{R}$ and $b \in \mathbb{R}$ with $a \leq b$ and $z = b$ such that:*

1. $\forall x \in [a, b], f(x) = 0$;
2. $\forall \epsilon > 0, \exists x \in [a - \epsilon; a), f(x) < 0$;
3. $\forall \epsilon > 0, \exists x \in (b; b + \epsilon], f(x) > 0$.

4 Syntax

We model the hybrid syntax (Fig. 2) as a subset of the existing Zélus syntax [5]. We extend this core syntax with reset clauses, which are essential for our hybrid examples; although resets are supported by the Zélus implementation, they are omitted from the minimal formal syntax in [5] and later introduced as equations in [4]. We syntactically distinguish among expressions that are purely continuous-time, *ce*, purely discrete-time, *de*, and hybrid, *e*. Continuous expressions consist of constants, variables, arithmetic operations, and tuples, and form the fragment that can appear in differential equations and zero-crossing guards. The last operator represents the left limit of the continuous variable x. Discrete expressions are inherited from prior work on a purely discrete subset of Zélus [12]; they include standard synchronous constructs such as local bindings, recursion, function application, and fby, the operator that initializes a stream with one value and specifies its subsequent values with another expression.

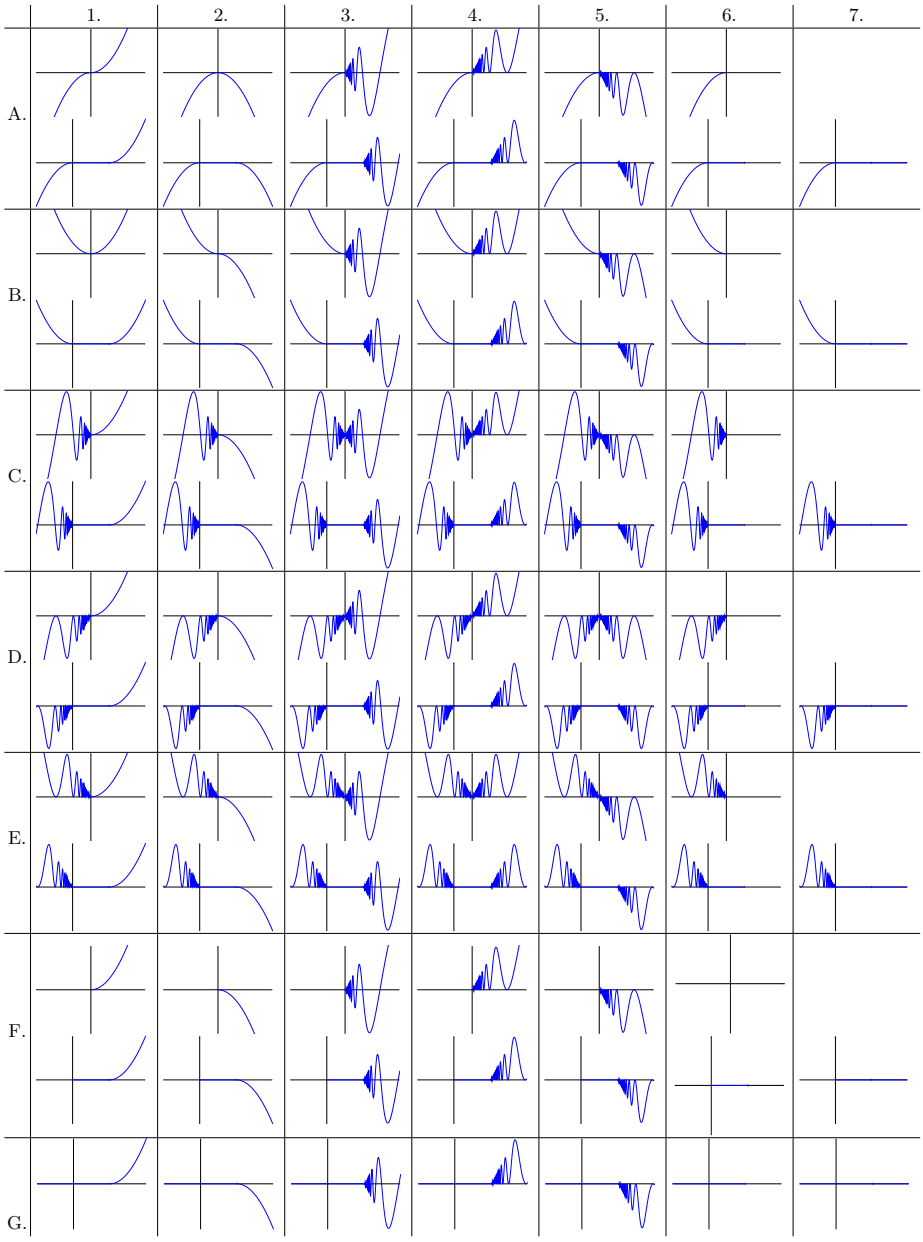

Fig. 1. Visualization of the 85 different cases studied in this paper: Rows A–G represent the seven possible behaviors of f on arbitrarily small left neighborhoods of a, and columns 1–7 represent the symmetric seven behaviors on arbitrarily small right neighborhoods of b. Each panel shows the function shape corresponding to the intersection of the row and column cases. For each case (e.g. E4), there are two subcases, for the passing case ($a = b$) and the staying case ($a < b$). All cases are explained in detail in the extended version of the paper. For cases in line G and column 7, only the staying case is applicable.

$$
\begin{aligned}
ce &::= c \mid x \mid \mathsf{last}\ x \mid ce_1 \oplus ce_2 \mid (ce_1, \ldots, ce_m) && \text{Continuous Expressions}\\
de &::= c \mid x \mid \mathsf{let}_h\ (x : \tau) = de_1\ \mathsf{in}\ de_2 && \text{Discrete Expressions}\\
&\quad \mid \mathsf{let\ rec}_h\ (x : \tau) = de_1\ \mathsf{in}\ de_2 \mid f\ x \mid de_1\ \mathsf{fby}\ de_2\\
&\quad \mid \mathsf{delay}(de) \mid \mathsf{if}\ x\ \mathsf{then}\ de_1\ \mathsf{else}\ de_2 \mid (de_1, ..., de_m)\\
e &::= \mathsf{let\ rec\ der}\ \mathbf{x}^m = ce_f\\
&\quad \mathsf{init}\ de_0\ (\mathsf{reset\ up}(ce_i) \to de_{r,i})_{i\in[1,n]}\ \mathsf{in}\ ce && \text{Hybrid Expressions}\\
D &::= \mathsf{let}\ x : \tau = ce \mid \mathsf{let}\ x : \tau = de && \text{Global Definitions}\\
&\quad \mid \mathsf{let}\ f\ (x : \tau_1) : \tau_2 = de
\end{aligned}
$$

Fig. 2. Program Syntax

The central hybrid construct is *let rec der ... init ... reset ... in ce*, which describes a continuous segment governed by an ODE together with event triggered resets that restart the segment when an upward zero-crossing is detected. Tuples provide a uniform way to represent multi-dimensional continuous state, which is needed because many of our examples evolve over several continuous variables simultaneously. Finally, global definitions, D, are top-level assignments that form the initial environment of a program and may be used to define constant variables or discrete functions.

4.1 Specification Syntax

We derive our specification syntax (Fig. 3) from prior work on refinement types for synchronous programming [12]. A refinement type has the form $\{v : b \mid \varphi\}$, where b is a base type and φ constrains the values of v. In our hybrid setting, however, φ is restricted to an invariant trace predicate of the form $\square p$, where p is a state predicate over continuous expressions. Intuitively, this means that refinements specify safety properties that must hold throughout the execution of a hybrid program, rather than only at a single logical instant.

A key difference is the omission of "point-wise" temporal predicates such as $\bigcirc$ and $\mathsf{hd}()$, which apply instantaneously to a specific point in time and are less useful for continuous programs. Consequently, all type refinements become invariant specifications. The syntax of discrete-time types is unchanged apart from requiring invariant types. Although continuous programs are represented in Zélus using floating-point numbers, we make the simplifying assumption that their representation is precise enough to be treated as reals and defer to the existing literature for formally treating floating-point errors.

5 Motivating Examples

We begin by providing some motivating examples of systems exhibiting hybrid behavior, and demonstrate how they can be verified using our formalism (Section 8). Additional examples can be found in the extended version of the paper.

$$
\begin{aligned}
\tau &::= b \mid \{v : b \mid \varphi\} && \text{(Types)}\\
b &::= \mathsf{float} \mid b_1 \times b_2 \times \ldots \times b_n && \text{(Base Types)}\\
p, q &::= \mathsf{true} \mid \mathsf{false} \mid x \mid ce_1 = ce_2 \mid ce_1 > ce_2 && \text{(State Predicates)}\\
&\quad \mid p \wedge q \mid \neg p\\
\varphi &::= \Box\, p && \text{(Trace Predicates)}
\end{aligned}
$$

Fig. 3. Hybrid Type Syntax

5.1 Water Tank

A hybrid system often has multiple reset conditions and these resets do not always have direct influence over the variable of interest. The following program demonstrates a controller which is responsible for filling or draining a tank of water to maintain its level between two setpoints. In this case, the resets do not directly set the water level but instead influence its rate of change. For simplicity, we assume the flow rate can react instantaneously to controller inputs.

```
let rec der (level, flow) =(flow init 5., 0. init 5.) reset
    | up(level -. 9.) -> (last level, -5.)
    | up(-.level +. 1.) -> (last level, 5.)
in (level, flow)
```

Ensure: $1 \leq level \leq 9$ at all times

5.2 Event-Triggered Automatic Braking

This system features a vehicle equipped with an automatic braking controller modeled after [22]. The vehicle begins behind a stationary obstacle with an initial velocity and acceleration. The vehicle's starting position is assumed to be far enough behind the obstacle that a collision-free trajectory is possible given the vehicle's maximum braking deceleration. The controller is triggered to react when the position at which the vehicle will stop given maximum braking force begins to surpass the position of the obstacle (plus a safety margin), at which point the controller applies full braking deceleration. The vehicle stops decelerating when its velocity reaches zero. The property to verify is that the vehicle's position never exceeds that of the obstacle, hence remaining collision-free.

```
let x_obs = 5.; brake = -0.2
let rec der (x,v,a)  = (v init 0., a init 1., 0. init 1.) reset
  | up(x -. (v *. v /. (2. *. brake)) +. 0.1 -. x_obs) ->
      (last x, last v, brake)
  | up(-. v ) -> (last x, last v, 0.)
in (x, v, a)
```

Ensure: $x < x_{obs}$

6 Hybrid Stream Semantics

We extend the semantics of the existing discrete-time synchronous language MARVeLus, [12] which expresses a single step of a stream program's execution with the judgment $S; \sigma \vdash e \overset{v}{\hookrightarrow} e'$. A program, represented by e, is executed in an environment $S; \sigma$ of functions and terms respectively. The use of separate environments is consistent with previous formalisms [12,5]. This results in a value v, which is the output of the expression for a given instant in time, as well as a rewritten expression e' to be executed in the next instant. Intuitively, the value v represents the concrete value represented by the program e, while e' represents the program's future behavior.

The semantics of MARVeLus are equipped to model a strictly discrete-time subset of Zélus. Consequently, the goal of our formalism is to not only extend the MARVeLus semantics with continuous dynamics, but to also ensure our extension is compatible with the original discrete-time semantics so that the combined semantics can express hybrid behaviors.

Mixing of continuous and discrete time components is particularly challenging. Fortunately, Zélus addresses this with its handling of physical and logical time, allowing the two to synchronize exclusively at the occurrence of upward zero-crossings [5]. When a Zélus program executes, either the continuous or the discrete components are executed, but never both simultaneously. The discrete program executes an entire synchronous iteration in zero physical time, at which point the system integrates the continuous-time dynamics until a zero-crossing detection is triggered. If or when this occurs, the discrete program resumes with logical time advanced by one instant and variable values possibly updated by the continuous dynamics. This cycle repeats indefinitely, resulting in program execution that alternates between continuous- and discrete-time phases. We note that continuous (hybrid) programs may contain discrete subterms, but not the other way around. This is consistent with the design of Zélus. Therefore, a hybrid program has all its continuous components at the top-level and never inside any discrete components.

We begin by defining the semantics of continuous expressions and predicates, following a similar convention to dL [26]. Given a finite set of variables Var, a continuous expression ce of base type b is evaluated in a state $\mathscr{S} : Var \rightharpoonup \mathbb{R}$ as $[\![ce]\!] : \mathscr{S} \to b$, defined as:

Definition 2. *For a state $\omega : \mathscr{S}$:*

$$
\begin{aligned}
[\![x]\!]\omega &= \omega(x) && \text{if } x \text{ is a variable} \\
[\![c]\!]\omega &= c && \text{if } c \text{ is a constant} \\
[\![ce_1 \spadesuit ce_2]\!]\omega &= [\![ce_1]\!]\omega \spadesuit [\![ce_2]\!]\omega && \text{where } \spadesuit \text{ is a binary arithmetic operator} \\
[\![p \clubsuit q]\!]\omega &= [\![p]\!]\omega \clubsuit [\![q]\!]\omega && \text{where } \clubsuit \text{ is a logical connective} \\
[\![\neg p]\!]\omega &= \neg [\![p]\!]\omega \\
[\![(ce_1, ..., ce_n)]\!]\omega &= ([\![ce_1]\!]\omega, ...[\![ce_n]\!]\omega)
\end{aligned}
$$

In the case of vector expressions, such as $\mathbf{x}$ defined as $(x_1, ...x_n)$, we implicitly apply necessary projection and concatenation operations but otherwise treat them as a single expression. Note that our convention is reversed from that of [26] to the expression-then-environment notation, to be more consistent with other operational semantics.

Syntactically determining the solution for a dynamical system is beyond the scope of the present work; we assume the existence of solutions of the dynamical systems. Consequently, we define an abstract solver "flow" and its accompanying semantic rule written in terms of such a solution to obtain the continuous evolution of the system until the next zero-crossing:

$$\frac{\begin{array}{l}\forall i.\ \forall t' \in [0, t_r].\ \dfrac{d\,\rho(t)(x_i)}{dt}(t') \ = \ [\![ce_i]\!]_{i\in[1,n]}\rho(t') \qquad \rho(0)(\mathbf{x}) = \mathbf{v_0} \\ \forall y, \forall i.\ \forall t' \in [0, t_r].y \neq x_i \implies \left(\rho(y) = \sigma(y) \wedge \dfrac{d\,\rho(t)(y)}{dt}(t') = 0\right) \\ t_r = +\infty \vee t_r \text{ is the first zero-crossing of } \{[\![u_j]\!]\rho(t) \mid j \in [1, m]\}\end{array}}{\sigma \vdash \mathsf{flow}\big(\mathsf{der}\ \mathbf{x} = ce,\ \mathbf{v_0},\ \{u_j\}_{j\in[1,m]}\big) \Downarrow \big(t \mapsto \rho(t),\ t_r\big)}$$

Here, a dynamical system composed of one or more differential equations, their initial states, zero-crossing detection functions, and the surrounding global environment has a solution given by the time-varying state $\rho(t)$ defined from $t = 0$ to $t = t_r \in \mathbb{R}^*_+ \cup \{+\infty\}$ (strictly positive or infinity), equivalently $t_r \in (0, +\infty]$ at which point t_r is the earliest time satisfying the chosen zero-crossing predicate for u. Consequently, $\rho(t_r)$ represents the system state at the moment a zero crossing occurs (if it reaches the zero-crossing). Continuous global definitions are also imported into ρ from σ; we follow the convention of dL in requiring that they are constant [26].

Definition 3. *For time $t \in \mathbb{R}^*_+ \cup \{+\infty\}$, t is the* first zero-crossing *of a set of n functions $\{f_i : \mathbb{R}^*_+ \cup \{+\infty\} \to \mathbb{R} \mid i \in [1, n]\}$ if:*

- *$\exists i \in [1, n]$. t_r is a zero-crossing of $f_i(t_r)$, and*
- *$\forall t' < t_r$. $\nexists i \in [1, n]$. t' is a zero-crossing of $f_i(t_r)$*

Here, in $\rho : [0, t_r + \epsilon] \to \mathscr{S}$, for a sufficient ϵ to verify that t_r is a zero-crossing according to Definition 1 and $\mathscr{S} : Var \rightharpoonup \mathbb{R}$ which associates a finite set of variables to real values.

We define two semantic rules (S-LETREC-DER-FIN) and (S-LETREC-DER-INF) for the two possibilities of hybrid program execution—finite and infinite continuous segments respectively. The judgment returns a "trace" of the program's continuous execution and the length for which said segment had evolved. In the finite case, the judgment also provides the output of the program after the reset. (S-LETREC-DER-FIN) first evaluates the expression de_0 defining the continuous segment's initial condition using discrete computation. Then the semantic judgment on flow is invoked to obtain a trace of the program's dynamical behavior until a zero-crossing is encountered. It is necessary to disambiguate zero-crossings in the case where two guards u_i and u_j simultaneously approach

(S-Letrec-Der-Fin)

$$\frac{\begin{array}{c} S;\sigma \vdash de_0 \overset{v_0}{\hookrightarrow} (de_0)' \qquad \mathsf{flow}(\mathsf{der}\ x^m = ce_f, v_0, (ce_{u,i})_{i\in[1,n]}) \Downarrow (\rho, t_r) \\ t_r < +\infty \qquad j = \mathsf{min}(\{j \mid j \in [1,n] \wedge t_r \text{ is a zero-crossing of } [\![ce_{u,j}]\!]\rho(t)\}) \\ S;\sigma,[x^m \mapsto \rho(t_r)(x^m) :: \mathsf{nil}] \vdash de_{r,j} \overset{v_{r,j}}{\hookrightarrow} de'_{r,j} \end{array}}{\begin{array}{c} S;\sigma \vdash \mathsf{let\ rec\ der}\ \mathbf{x}^m : \tau = ce_f\ \mathsf{init}\ de_0\ \mathsf{reset}\ (\mathsf{up}(ce_{u,i}) \to de_{r,i})_{i\in[1,n]}\ \mathsf{in}\ ce \\ \overset{(t\mapsto([\![ce]\!]\rho(t)),t_r);[\![ce]\!][x^m\mapsto v_{r,j}]}{\hookrightarrow} \\ \mathsf{let\ rec\ der}\ \mathbf{x}^m : \tau = ce_f\ \mathsf{init}\ v_{r,j}\ \mathsf{reset}\ (\mathsf{up}(ce_{u,i}) \to de_{r,i})_{i\in[1,n]}[de_{r,j} \mapsto de'_{r,j}]\ \mathsf{in}\ ce \end{array}}$$

(S-Letrec-Der-Inf)

$$\frac{S;\sigma \vdash de_0 \overset{v_0}{\hookrightarrow} (de_0)' \qquad \mathsf{flow}(\mathsf{der}\ x^m = ce_f, v_0, (ce_{u,i})_{i\in[1,n]}) \Downarrow (\rho, +\infty)}{\begin{array}{c} S;\sigma \vdash \mathsf{let\ rec\ der}\ \mathbf{x}^m : \tau = ce_f\ \mathsf{init}\ de_0\ \mathsf{reset}\ (\mathsf{up}(ce_{u,i}) \to de_{r,i})_{i\in[1,n]}\ \mathsf{in}\ ce \\ \overset{(t\mapsto([\![ce]\!]\rho(t)),+\infty);\mathsf{nil}}{\hookrightarrow} \\ [\infty] \end{array}}$$

Fig. 4. Hybrid Stream Semantics of the form $S;\sigma \vdash e \overset{(v(t),t_r);[v_r|\mathsf{nil}]}{\hookrightarrow} e'$

zero. The semantics of Zélus deterministically chooses the first zero-crossing by syntactic order. Finally, when a zero-crossing is encountered, the j-th expression representing its reset is evaluated with the values of the continuous variables immediately prior to reset populated in its context. (S-LETREC-DER-INF) has a similar structure for its initial and continuous phases.

$[\infty]$ denotes the result of a continuous phase which never resets. This is distinct from the classic notion of "getting stuck" in operational semantics; the program still evolves in continuous time but requires no intervention from discrete resets to remain safe. Thus no discrete computation is triggered.

Note that a rewrite of a hybrid program can produce one of two outputs, respectively for the finite and infinite cases:

- $(t \mapsto [\![ce]\!]\rho(t), t_r); [\![ce]\!][x^m \mapsto v_{r,j}] : \left(([0,t_r] \to b) \times \mathbb{R}_+^* \cup \{\infty\}\right); b$ for programs where the current continuous segment concludes with a zero-crossing detection; $[x^m \mapsto v_{r,j}]$ represents the program's state after the discrete computation following the zero-crossing detection and b is the base type of ce, the expression to be evaluated in the context of the hybrid definition.
- $(t \mapsto [\![ce]\!]\rho(t), t_r); \mathsf{nil} : ([0,t_r] \to b) \times \mathbb{R}_+^* \cup \{\infty\}$ for programs with an infinite continuous segment and thus do not produce a reset value

6.1 Predicate Semantics

Verifying that a hybrid program satisfies a given predicate requires formally establishing the requirements for each phase of execution to guarantee its satisfaction. In the discrete-time case [12], this requires showing that the program's value computed (hence its "output") at each instant satisfies the requirements

for the predicate at that instant. The hybrid case introduces some additional complexity. Although the program still only computes outputs on discrete-time events, one must also guarantee that the program's dynamics do not violate the specification in the continuous-time phase between these events. Naturally, this requires knowing that a program's execution satisfies the specification for the entire duration of all continuous-time phases, and that furthermore, any discrete computations instantaneously satisfy the specification as well. Since we are primarily interested in verifying safety conditions, we focus on invariant properties of the form $\Box p$.

Definition 4. *A hybrid program e satisfies an invariant property $\Box q$ if:*

- *For any continuous execution of e over time $t \in [0, t_r], t_r \in \mathbb{R}_+^* \cup \{+\infty\}$ producing a trace $\rho(t)$, $[\![q]\!]\rho(t)$ is true for all $t \in [0, t_r]$*
- *For any discrete computation $e_r \xhookrightarrow{v_r} e'_r$ within e which resets its state producing the post-reset state $[x \mapsto v_r]$, $[\![q]\!][x \mapsto v_r]$ is true.*

Note that, in our semantics, t represents the time elapsed since the beginning of the *current* continuous segment, not global time. Consequently, we require an invariant property to hold for any continuous or discrete execution that occurs at *any* time.

A pathological case that arises in hybrid systems is Zeno behavior, in which the number of discrete-time events within a given span of time tends towards infinity, as demonstrated by a bouncing ball with decaying velocity. As a result, there is a finite time beyond which the system cannot progress. Hybrid modelers of all kinds must contend with such behaviors and there are approaches to mitigating issues that arise in simulating them [18]. We make no claim of addressing liveness in these cases. Instead we note that since our predicate semantics is oriented around the execution trace of the program rather than physical time, invariant properties do not make potentially unsafe claims about the system's safety beyond reachable times if Zeno behavior exists. In all other cases, the time of the execution would indeed extend to infinity.

7 The Hybrid Type System

The type system of the hybrid extension is constructed with adequate conditions so that a well-typed program is guaranteed to provably satisfy a given safety property, encoded as a refinement type. We take inspiration from the sound inference rules of dL and consequently structure our typing rules similarly.

A crucial difference that arises when applying these techniques to verifying Zélus programs is that there is no exact equivalent to dL's evolution domain, which immediately interrupts continuous evolution if the system begins to exit said domain, in Zélus. While upward zero-crossing detection can approximate an evolution domain, the safe application of this technique is nontrivial. An evolution domain simply examines the instantaneous value of the system to determine whether to interrupt the continuous segment, so it is easy to guard

against, for instance, a value exceeding a given number. However, a zero-crossing detector, per Definition 1, can only do the same if its guard is first negative for a nonzero period of time. As a result, the properties that a zero-crossing detector can "guard" against can only be assumed if the zero-crossing detector can be verified to be "active" at that moment. We introduce the following operator:

Definition 5.

$$Active(v, \{u_1, ..., u_n\}) \triangleq \bigwedge \{\Box(u_i \leq 0) \mid [\![(u_i < 0 \ \vee \ (u_i = 0 \Rightarrow \dot{u}_i < 0))]\!][x \mapsto v]\}.$$

which, for a given value v representing the current state of the system, provides the predicates based on the guards of the zero-crossing detectors that can be assumed to be true until at least the next zero-crossing (or for all time if none occur). This is helpful because not all properties can be proven with a differential invariant alone. Thus, this function helps determine which parts of a property can be assumed true in the presence of zero-crossing detectors that will interrupt the program before they are violated. To prove an invariant property I is true for a continuous segment with an initial value v_0, it is necessary to find a ϕ that is provable via differential invariants such that $(\phi \wedge Active(v_0, \{u_1, ...u_n\})) \implies I$.

A zero-crossing guard u_i is "active" if at any point during the execution of the continuous segment $t \in (0, t_r]$, $[\![u_i]\!]\rho(t) = 0$ implies that t is a zero-crossing of u_i. Definition 5 considers only zero-crossings that can be determined to be active knowing only their instantaneous position and derivative. This is desirable for the proof because it requires no knowledge of the dynamical system's explicit solution. Requiring that u_i begins negative is obvious; it will require a nonzero amount of time for u_i to evolve towards zero thus satisfying Definition 1. The second condition, also allowing u_i to be considered active if it begins at zero with a negative derivative, arises because u_i will immediately become negative during the continuous segment, thus also requiring nonzero time to continuously evolve towards zero. Here, $\dot{u}_i$ represents the symbolic time derivative of the expression, as defined in [26]. This allows, for instance, verifying that a bouncing ball does not fall through the ground immediately after a bounce, since it would be moving away from the ground.

Safety of a program's execution is proven by induction on segments consisting of continuous integration possibly interrupted by a zero-crossing and a discrete computation which initializes the next segment. This is captured in the typing rule (T-DER-FIN) (Fig. 5). For each segment, the initial value must first be verified safe. Then, using differential reasoning based on the zero-crossings that are active at the beginning of the continuous period, the continuous part is proven to also obey the invariant. This involves determining which parts of the invariant are proven with knowledge that the active zero-crossings will interrupt execution before they are violated, and then verifying that the remaining portion is provable via a differential invariant. Although these are expressed as existentials in the typing rule, they can be determined based on user-provided annotations or simple heuristic methods in implementation. Concretely, one portion of the

$$\text{(T-DER-FIN)}\quad \frac{\begin{array}{r}(T1)\ \exists\phi_I.\Gamma, x:\{v:b|\phi(v)\}, \dot{x}:\{v:b|v=e_f\} \vdash e_0 : \{v:b|\phi(v) \wedge \phi'_I(\dot{x}) \wedge \\ ((\phi_I(x) \wedge \mathit{Active}(v,\{u_1 ... u_n\}) \implies \phi(x))\} \\ (T2)\ \forall i \in [1,n], \exists \phi_{i,I}\ .\ \Gamma, x:\{v:b|\phi(v) \wedge u_i(v)=0\}, \dot{x}:\{v:b|v=e_f\} \vdash \\ e_{r,i} : \{v:b|\phi(v) \wedge \phi'_{i,I}(\dot{x}) \wedge \\ ((\phi_{i,I}(x) \wedge \mathit{Active}(v,\{u_1 ... u_n\}) \implies \phi(x))\} \\ (T3)\ \Gamma, x:\{v:b|\phi(v)\} \vdash ce : \tau\end{array}}{\Gamma \vdash \mathsf{let\ rec\ der}\ x:\{v:b|\phi(v)\} = e_f\ \mathsf{init}\ e_0\ \mathsf{reset}\ \begin{array}{l}\mid \mathsf{up}(u_1) \to e_{r,1} \\ \vdots \\ \mid \mathsf{up}(u_n) \to e_{r,n}\end{array}\ \mathsf{in}\ ce : \tau}$$

$$\text{(T-DER-INF)}\ \frac{}{\Gamma \vdash [\infty] : \tau}$$

Fig. 5. Stream Typing Rules for Hybrid Expressions

safety predicate may be designated by the user as the part to be established by differential reasoning. In this way, the existential form of the rule reflects proof flexibility for the metatheory, while still admitting a practical implementation.

Finally, all possible resets of the system are proven safe with respect to their triggering conditions to ensure that they are safe initial conditions for subsequent continuous integration. The infinite case types the residual term resulting from rewriting a hybrid expressions with an infinite continuous segment. Since the previous expression would have been well-typed through an application of (T-DER-FIN) which guarantees safety until the next reset, an absence of resets would vacuously imply safety for an infinite amount of time. Thus, no additional verification is necessary.

7.1 Type Safety

We prove type safety roughly following the classical syntax-guided manner of Wright and Felleisen [32].

Environment correspondence. We write $\Gamma \approx (S;\sigma)$ for the standard correspondence between a typing environment Γ and a runtime configuration $(S;\sigma)$ (control state S with store/history σ). Formally,

$$\Gamma \approx (S;\sigma) \quad \triangleq \quad \forall x \in \mathsf{dom}(\Gamma).\ x \in \mathsf{dom}(\sigma)\ \wedge\ (S;\sigma) \vdash \sigma(x) : \Gamma(x).$$

That is, every variable declared in Γ is bound in σ, and its runtime value is well-typed with respect to Γ.

Discrete preservation Our hybrid extension shares the discrete sublanguage and its small-step semantics with MARVeLus. Therefore, we reuse the discrete type preservation result [12], specifically Lemma 4 (Type Preservation), for all purely discrete evaluation and stepping that occurs inside reset branches and other

non-continuous terms. Furthermore, correspondence is shown to be preserved by terms introduced into the discrete environment by continuous components.

Lemma 1 (First up-crossing implies preservation of the active domain). *Assume*

$$flow(\mathsf{der}\ x = e_f,\ x_0,\ \{u_1, \ldots, u_n\}) \Downarrow (\rho, t_r)$$

with $t_r < +\infty$, and suppose branch i triggers at t_r (i.e., t_r is the first upward zero-crossing among the monitored signals). Let Active *be the predicate from Definition 5. Then:*

$$\forall t \in [0, t_r].\ \mathsf{Active}(\rho(0)(x), \{u_1, \ldots, u_n\}) \qquad \textit{and} \qquad u_i(\rho(t_r)(x)) = 0$$

where i is the index of the first zero-crossing per Definition 3, i.e. the evolution-domain facts contributed by Active *are preserved up to t_r.*

Proof can be found in the extended version of the paper.

Lemma 2 (Monitored-domain preservation for non-terminating flow). *Assume*

$$flow(\mathsf{der}\ x = e_f,\ x_0,\ \{u_1, \ldots, u_n\}) \Downarrow (\rho, +\infty).$$

Let Active *be the predicate from Definition 5. Then:*

$$\forall t \geq 0.\ \mathsf{Active}(\rho(0)(x), \{u_1, \ldots, u_n\}).$$

Proof can be found in the extended version of the paper.

Lemma 3 (Continuous Type Preservation).

A hybrid program e in an environment Γ corresponding with $(S; \sigma)$, which is well-typed ($\Gamma \vdash e : \{v : b \mid \Box\phi(v)\}$), and steps ($S; \sigma \vdash e \overset{(v(t),t_r);[v_r|\mathsf{nil}]}{\hookrightarrow} e'$), has the following hold:

1. $\Box_{[0,t_r]}\phi(v(t))$
2. $\Gamma \vdash v_r : \{v : b \mid \Box\phi(v)\}$ *if t_r is finite*
3. $\Gamma \vdash e' : \{v : b \mid \Box\phi(v)\}$

We prove that a well-typed continuous equation, with sufficiently safe initial conditions per (T-DER-FIN), remains well-typed after one continuous segment, and that its generated trajectory satisfies the safety predicate throughout the corresponding integration interval. For the finite case, we use the operational semantics to obtain a trace $\rho(t)$ up to the first monitored event time t_r; Lemma 1 shows that the conditions induced by the monitored guards hold for all $t \in [0, t_r]$.

We then combine these domain facts with the differential invariant provided by the typing premises (via Platzer's differential invariant rule [26]) to conclude that the continuous segment is safe (1). We then show that, in the case of finite t_r, the assumptions for discrete type preservation are met, and that the value obtained by executing the discrete program comprising the reset preserves the

invariant (2). Furthermore, we show that this constitutes an initial condition sufficiently safe so that the subsequent continuous segment can then be proven safe (3). Per (S-LETREC-DER-FIN), this value is rewritten as the initial condition for the next iteration, thus enabling an inductive proof of safety. For the infinite case, the same argument applies, except that the monitored-domain lemma holds for all time, yielding a direct proof of (1) and an inert $[\infty]$ residual that is typed directly. The complete proof can be found in the extended version of the paper.

8 Verifying the Examples

We provide brief proof sketches for using our formalism to verify safety of the examples introduced in Section 5. Detailed proofs can be found in the extended version of the paper.

8.1 Water Tank

This system is proven safe by establishing that the invariant $(1 \leq \mathit{level} \leq 9)$ holds throughout execution. Per (T-DER-FIN), the initial condition satisfies the invariant, so we may assume safety at the start of the first segment. During each continuous evolution, safety is maintained because the active zero-crossing guards become critical before either bound is violated, forcing a reset while the invariant still holds. Each reset then re-establishes $(1 \leq \mathit{level} \leq 9)$ and drives the dynamics back inward, yielding a safe starting state for the next segment. Repeating this argument over successive continuous segments and resets yields, by induction, that the invariant holds globally.

8.2 Event-Triggered Automatic Braking

This system is proven safe by first proving that the invariant $\Box(x - \frac{v^2}{2b} < x_{obs})$ holds. Per (T-DER-FIN), the initial condition can be proven safe with the trivial differential invariant $\phi_I = \mathsf{true}$ since the zero-crossing guard, which is identical to the invariant, is negative. Since the reset expression sets the acceleration exactly equal to the acceleration required to stop at exactly the position of the obstacle, the zero-crossing guard stays at zero and thus the invariant must shift back to ϕ_I to be proven using the differential invariant technique.

More examples and their corresponding proofs can be found in the extended version of the paper.

9 Related Work

Hybrid Semantics: Precision is essential in hybrid semantics, and is particularly challenging when discrete program executions must be scheduled along with continuous dynamics, as imprecise semantics leads to unpredictable models of their

interactions. This has been addressed in the Zélus language [5] which restricts interactions between continuous and discrete executions to mitigate inconsistencies that may arise from solver implementation. Benveniste *et al.* characterize the hybrid semantics of Zélus with a *non-standard* semantics based on infinitesimal time steps and idealized micro-steps [3]. In contrast, our work stays within the conventional time representation common to other formalisms of hybrid semantics but makes the semantics of event detection (zero-crossings) explicit so that sound reasoning principles can be stated directly. Differential dynamic logic (dL) provides a compositional deductive verification framework for hybrid systems, with proof rules tailored to hybrid programs and differential equations [25]. Operational accounts in Ptolemy II and related work emphasize executability, determinism, and discrete/continuous interaction (often via superdense time) [20]. Our work is complementary: rather than proposing a general proof calculus or a particular execution model, we formalize language-level event detection (zero-crossings) and its interaction with continuous evolution so that metatheory and type-based verification can rely on a precise account of events.

Verifying Hybrid Systems and CPS: Verification of hybrid systems has been extensively studied in control theory, with approaches like barrier certificates [28] which can efficiently provide sufficient safety conditions. However, there is also interest in verifying systems described by programming languages as they may offer a more straightforward transfer to real-world implementations. Lingua Franca makes timing and concurrency semantics explicit for CPS using specific models of computation, enabling deterministic-by-construction designs [21]. Recent work also formalizes subsets of Lingua Franca in rewriting logic to support analysis in Maude [23]. These efforts reinforce the broader trend of semantics-driven verification; our contribution addresses the analogous need for hybrid programs by establishing zero-crossings and continuous evolution in a verification-friendly semantics. Similarly, KeYmaera X is an interactive and automated theorem prover for hybrid systems based on dL, with tactic-based proof search and a small trusted core [14]. Furthermore, Plaidypvs [31] embeds dL into PVS, supporting standard dL-style proofs while additionally enabling reuse of existing PVS theories to reason over broad classes of hybrid programs [31]. While our approach in comparison is more limiting in the systems that are verifiable, we accept this compromise in the interest of pursuing automated verification via the type checker and direct execution of verified source code. Another related work is HABS, a hybrid programming language for formal modeling and verification of hybrid systems [17], which uses a similar synchronization mechanism. HABS translates programs into differential dynamic logic and discharges the resulting proof obligations in KeYmaera X, whereas our work targets Zélus and performs verification at compile time via refinement typing. There are also specification-based runtime monitoring approaches for cyber-physical systems [2]. These methods monitor and predict CPS behaviors during simulation-time or at runtime, while our work instead targets compile-time verification through a refinement type system for statically establishing safety properties.

Verification via Types: Liquid types enable automated type-checking by restricting refinements to a decidable fragment [29], and LiquidHaskell shows these ideas scale to realistic programs without changing the dynamic semantics [30]. This motivates using types as a lightweight automated verification interface. Our setting differs in that correctness depends not only on value-level invariants but also on the meaning of continuous evolution and event detection, which necessitates a formal characterization of zero-crossings for type soundness and verification.

10 Discussion

The formalization of zero-crossings and hybrid synchronous semantics lays the critical groundwork for a language capable of simultaneous formal verification and execution. Our framework interprets a hybrid program's execution as a stream of alternating continuous and discrete executions. This enables invariants to be proven via induction. Due to the unique semantics induced by the Zélus zero-crossing detector `up`(), establishing sufficient conditions to successfully prove safety via induction proved to be surprisingly complex. Nevertheless, this method provides an approach to syntactically verify the correctness of a synchronous program with hybrid components using tactics inspired by differential dynamic logic [26]. Compared to *dL*, synchronizing discrete events is more challenging due to our handling of zero-crossings. We gain an executable, compositional language-level proof interface but lack an explicit evolution-domain constraint Q as in *dL*. Compared to *hybrid automata*, we gain determinism and code alignment, but must handle low-level event corner cases more explicitly. Our current formalization handles discrete transitions induced by zero-crossing events of continuous expressions, but does not yet account for inherently discrete or exogenous events outside that mechanism.

Although the restriction of our predicates to strictly invariant properties allowed for simpler proofs, it may be useful to expand the proof rules to account for other temporal operators or logics, like Signal Temporal Logic which ranges over time intervals. Furthermore, the formalism would benefit greatly from automation in an SMT-based type system like that of Liquid Haskell [30], or combined with MARVeLus [12] to form a complete, automatically verifiable hybrid subset of Zélus. Nevertheless, in its current state, our formalism demonstrates that it is possible to characterize zero-crossings and synchronous hybrid programs that rely on them to verify safety properties.

Acknowledgments

The authors would like to thank Timothy Bourke, Grégoire Bussone, Charles de Haro, Paul Jeanmaire and Partha Roop for interesting discussions on earlier versions of this paper. This work was funded in part by National Science Foundation grants CCF-2348706 and CCF-2426474, and the U.S. Federal Aviation Administration through the FAST Grant Program, Grant Number NG-FAS-0016, Funding Opportunity Number 23-FAA-FAST-001 under the supervision

of Joshua Glottmann. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the FAA.

References

1. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) Hybrid Systems. Lecture Notes in Computer Science, vol. 736, pp. 209–229. Springer (1993). https://doi.org/10.1007/3-540-57318-6_30
2. Bartocci, E., Deshmukh, J., Donzé, A., Fainekos, G., Maler, O., Ničković, D., Sankaranarayanan, S.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Lectures on Runtime Verification: Introductory and Advanced Topics, pp. 135–175. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_5
3. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: Non-standard semantics of hybrid systems modelers. Journal of Computer and System Sciences **78**(3), 877–910 (2012). https://doi.org/10.1016/j.jcss.2011.08.009
4. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: Non-standard semantics of hybrid systems modelers. Journal of Computer and System Sciences **78**(3), 877–910 (2012)
5. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code. In: Proceedings of the ninth ACM international conference on Embedded software - EMSOFT '11. p. 137. ACM Press, Taipei, Taiwan (2011). https://doi.org/10.1145/2038642.2038664
6. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., De Simone, R.: The synchronous languages twelve years later. Proceedings of the IEEE **91**(1), 64–83 (2003). https://doi.org/10.1109/JPROC.2002.805826
7. Berry, G.: Real time programming: Special purpose or general purpose languages. In: Ritter, G. (ed.) Information Processing '89: Proceedings of the IFIP 11th World Computer Congress. pp. 11–17. Elsevier Science Publishers B.V. (North-Holland) (1989)
8. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19**(2), 87–152 (1992). https://doi.org/10.1016/0167-6423(92)90005-V
9. Bourke, T., Pouzet, M.: Zélus: A synchronous language with ODEs. In: Proceedings of the 16th international conference on Hybrid systems: computation and control. pp. 113–118 (2013). https://doi.org/10.1145/2461328.2461348
10. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: a declarative language for real-time programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 178–188 (1987). https://doi.org/10.1145/41625.41641
11. Cellier, F.E.: Continuous System Modeling. Springer (1991). https://doi.org/10.1007/978-1-4757-3922-0
12. Chen, J., de Mendonça, J.L.V., Ayele, B.S., Bekele, B.N., Jalili, S., Sharma, P., Wohlfeil, N., Zhang, Y., Jeannin, J.B.: Synchronous Programming with Refinement Types **8**(ICFP), 268:938–268:972 (Aug 2024). https://doi.org/10.1145/3674657
13. Colaço, J.L., Pagano, B., Pouzet, M.: Scade 6: A formal language for embedded critical software development (invited paper). In: 2017 International Symposium on Theoretical Aspects of Software Engineering (TASE). pp. 1–11. IEEE (2017). https://doi.org/10.1109/TASE.2017.8285623

14. Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: Keymaera X: An axiomatic tactical theorem prover for hybrid systems. In: International Conference on Automated Deduction. pp. 527–538. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_36
15. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings 11th Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 278–292. IEEE (1996). https://doi.org/10.1109/LICS.1996.561342
16. Jhala, R., Vazou, N.: Refinement types: A tutorial. Found. Trends Program. Lang. **6**(3–4), 159–317 (Oct 2021). https://doi.org/10.1561/2500000032
17. Kamburjan, E., Mitsch, S., Hähnle, R.: A hybrid programming language for formal modeling and verification of hybrid systems. Leibniz Transactions on Embedded Systems **8**(2), 04–1 (2022)
18. Konečný, M., Taha, W., Bartha, F., Duracz, J., Duracz, A., Ames, A.: Enclosing the behavior of a hybrid automaton up to and beyond a Zeno point. Nonlinear Analysis: Hybrid Systems **20**, 1–20 (May 2016). https://doi.org/10.1016/j.nahs.2015.10.004
19. Le Guernic, P., Gautier, T., Le Borgne, M., Le Maire, C.: Programming real-time applications with SIGNAL. Proceedings of the IEEE **79**(9), 1321–1336 (Sep 1991). https://doi.org/10.1109/5.97301
20. Lee, E.A., Zheng, H.: Operational semantics of hybrid systems. In: Hybrid Systems: Computation and Control (HSCC 2005). Lecture Notes in Computer Science, vol. 3414, pp. 25–53 (2005). https://doi.org/10.1007/978-3-540-31954-2_2
21. Lin, S., Manerkar, Y.A., Lohstroh, M., Polgreen, E., Yu, S.J., Jerad, C., Lee, E.A., Seshia, S.A.: Towards building verifiable CPS using lingua franca. ACM Transactions on Embedded Computing Systems **22**(5s), 155:1–155:24 (2023). https://doi.org/10.1145/3609134
22. Loos, S.M., Platzer, A., Nistor, L.: Adaptive Cruise Control: Hybrid, Distributed, and Now Formally Verified. In: Butler, M., Schulte, W. (eds.) FM 2011: Formal Methods, vol. 6664, pp. 42–56. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_6, series Title: Lecture Notes in Computer Science
23. Marin, M., Ölveczky, P.C., Reja, M., Rukhaia, M., Bae, K., Dundua, B.: Semantics and formal analysis of Lingua Franca CPS specifications in rewriting logic. Tech. rep., University of Oslo / collaborators (technical report) (2024)
24. Nanevski, A., Morrisett, G., Birkedal, L.: Hoare type theory, polymorphism and separation. Journal of Functional Programming **18**(5-6), 865–911 (2008). https://doi.org/10.1017/S0956796808006953
25. Platzer, A.: Differential dynamic logic for hybrid systems. Journal of Automated Reasoning **41**(2), 143–189 (2008). https://doi.org/10.1007/s10817-008-9103-8
26. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-63588-0
27. Platzer, A.: A Complete Uniform Substitution Calculus for Differential Dynamic Logic. Journal of Automated Reasoning **59**(2), 219–265 (Aug 2017). https://doi.org/10.1007/s10817-016-9385-1
28. Prajna, S., Jadbabaie, A.: Safety Verification of Hybrid Systems Using Barrier Certificates. pp. 477–492. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_32
29. Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 159–169 (2008). https://doi.org/10.1145/1375581.1375602

30. Vazou, N., Seidel, E.L., Jhala, R.: Liquid Haskell: Experience with refinement types in the real world. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. pp. 39–51 (2014). https://doi.org/10.1145/2633357.2633366
31. White, L., Titolo, L., Slagel, J.T., Muñoz, C.: A temporal differential dynamic logic formal embedding. In: Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 162–176. CPP 2024, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3636501.3636943
32. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1), 38–94 (1994). https://doi.org/10.1006/inco.1994.1093

A Case Distinction

We have identified the following 7 cases for $x < a$:

A f is strictly negative in a ball left of a. Formally: $\ell < a$ and $\exists \eta > 0, \forall x \in [a-\eta, a), f(x) < 0$.
Examples: $(x-a)^{2k+1}$, $-|x-a|^k$, $-\exp(-\frac{1}{(x-a)^2})$.

B f is strictly positive in a ball left of a. Formally: $\ell < a$ and $\exists \eta > 0, \forall x \in [a-\eta, a), f(x) > 0$.
Examples: $(x-a)^{2k}$, $|x-a|^k$, $\exp(-\frac{1}{(x-a)^2})$.

C f is both strictly positive and strictly negative in any ball left of a. Formally: $\ell < a$ and $\forall \eta > 0, \exists x, y \in [a-\eta, a)$ such that $f(x) > 0$ and $f(y) < 0$.
Examples: $(x-a)^k \sin(\frac{1}{x-a})$, $\exp(-\frac{1}{(x-a)^2}) \sin(\frac{1}{x-a})$.

D f is negative or zero in a ball left of a, and is both negative and zero in any ball left of a. Formally: $\ell < a$ and $\exists \eta > 0, \forall x \in [a-\eta, a), f(x) \leq 0$ and $\forall \eta > 0, \exists x, y \in [a-\eta, a)$ such that $f(x) < 0$ and $f(y) = 0$.
Examples: $-(x-a)^{2k} \sin^2(\frac{1}{x-a})$, $-\exp(-\frac{1}{(x-a)^2}) \sin^2(\frac{1}{x-a})$.

E f is positive or zero in a ball left of a, and is both positive and zero in any ball left of a. Formally: $\ell < a$ and $\exists \eta > 0, \forall x \in [a-\eta, a), f(x) \geq 0$ and $\forall \eta > 0, \exists x, y \in [a-\eta, a)$ such that $f(x) > 0$ and $f(y) = 0$.
Examples: $(x-a)^{2k} \sin^2(\frac{1}{x-a})$, $\exp(-\frac{1}{(x-a)^2}) \sin^2(\frac{1}{x-a})$.

F $\ell = a \in \mathbb{R}$. This is equivalent to $\forall x \in [\ell; b], x = 0$.

G $\ell = a = -\infty$.

The 7 cases for $x > b$ are symmetric:

1. f is strictly positive in a ball right of b. Formally: $b < u$ and $\exists \eta > 0, \forall x \in (b, b+\eta], f(x) > 0$.
   Examples: $(x-b)^k$, $|x-b|^k$, $\exp(-\frac{1}{(x-b)^2})$.
2. f is strictly negative in a ball right of b. Formally: $b < u$ and $\exists \eta > 0, \forall x \in (b, b+\eta], f(x) < 0$.
   Examples: $-(x-b)^k$, $-|x-b|^k$, $-\exp(-\frac{1}{(x-b)^2})$.
3. f is both strictly positive and strictly negative in any ball right of b. Formally: $b < u$ and $\forall \eta > 0, \exists x, y \in (b, b+\eta]$ such that $f(x) > 0$ and $f(y) < 0$.
   Example: $(x-b)^k \sin(\frac{1}{x-b})$.
4. f is positive or zero in a ball right of b, and is both positive and zero in any ball right of b. Formally: $b < u$ and $\exists \eta > 0, \forall x \in (b, b+\eta], f(x) \geq 0$ and $\forall \eta > 0, \exists x, y \in (b, b+\eta]$ such that $f(x) > 0$ and $f(y) = 0$.
   Example: $(x-b)^{2k} \sin^2(\frac{1}{x-b})$.
5. f is negative or zero in a ball right of b, and is both negative and zero in any ball right of b. Formally: $b < u$ and $\exists \eta > 0, \forall x \in (b, b+\eta], f(x) \leq 0$ and $\forall \eta > 0, \exists x, y \in (b, b+\eta]$ such that $f(x) < 0$ and $f(y) = 0$.
   Example: $-(x-b)^{2k} \sin^2(\frac{1}{x-b})$.
6. $b = u \in \mathbb{R}$. This is equivalent to $\forall x \in [a; u], x = 0$.
7. $b = u = +\infty$.

The 7 cases for $x < a$ and the 7 cases $x > b$ give rise to 49 cases, most of them subdivided into a passing subcase and a staying subcase, bringing the total to 85. All those cases are summarized in Figure 1. Line G and column 7 do not have passing subcases, because it does not make sense to have $a = b$ with $a = -\infty$ or $b = +\infty$.

We now turn our interest to understanding the different possibilities for f when $x < a$ and $x > b$. We identify 7 different cases when $x < a$ (numbered A to G), as well as 7 different cases when $x > b$ (numbered 1 to 7). All cases are explained using examples in Appendix A and they are also summarized in Figure 1. In examples of this section, when a function is undefined at a point, by convention we extend it (by continuity) by zero at this point.

A.1 Possible Formal Definitions of a Zero-Crossing

Now that we have identified all possible cases, We are now ready to decide which cases should be zero-crossings, and which cases should not. But first we identify desirable properties of the definition.

Formal definitions of Zero-Crossings To respect Desirable Property 2 in Section 3, we eliminate columns 2, 5, 6 and 7; as well as lines B, E, F and G. We believe that A1, A4, D1 and D4 should clearly be zero-crossings. We believe that column 3 and line C are more controversial as to whether they should have a zero-crossing, or not, or throw an error.

Given those ideas, if we limit the zero-crossings to cases A1, A4, D1 and D4, we can come up with the following definition of a zero-crossing:

Definition 6. *$z \in \mathbb{R}$ is a zero-crossing for function f if and only if there exist $a \in \mathbb{R}$ and $b \in \mathbb{R}$ with $a \leq b$ and $z = b$ such that:*

1. $\forall x \in [a, b], f(x) = 0$;
2. $\forall \epsilon > 0, \exists x \in [a - \epsilon; a), f(x) < 0$;
3. $\forall \epsilon > 0, \exists x \in (b; b + \epsilon], f(x) > 0$;
4. $\exists \eta > 0, \forall x \in [a - \eta; a), f(x) \leq 0$;
5. $\exists \eta > 0, \forall x \in (b; b + \eta], f(x) \geq 0$.

Now, if we consider that beyond cases A1, A4, D1 and D4, cases A3, C1, C3, C4 and D3 should also be considered zero-crossings, we can come up with a looser definition of a zero-crossing, only including conditions (i), (ii) and (iii), which results in the definition that appears in the main paper:

Definition 7. *$z \in \mathbb{R}$ is a zero-crossing for function f if and only if there exist $a \in \mathbb{R}$ and $b \in \mathbb{R}$ with $a \leq b$ and $z = b$ such that:*

1. $\forall x \in [a, b], f(x) = 0$;
2. $\forall \epsilon > 0, \exists x \in [a - \epsilon; a), f(x) < 0$;
3. $\forall \epsilon > 0, \exists x \in (b; b + \epsilon], f(x) > 0$.

Both definitions respect Desirable Properties 1, 2 and 3. We can also imagine hybrid definitions that include conditions (i), (ii), (iii) and (iv); or (i), (ii), (iii) and (v) of Definition 1. In the rest of this paper we will consider Definition 2 in our semantics for zero-crossings.

Another decision is which point should be the zero-crossing in the staying case where $a < b$. We would typically want the zero-crossing to be a or b, but which one is almost purely a matter of convention. For this paper, we will pick b (but the subsequent developments can easily be adapted if picking a).

B Proof of Lemma 1

Proof. Define $g_j(t) \triangleq u_j(\rho(t)(x))$. Each g_j is continuous in t (because ρ is continuous and u_j is a continuous expression). Fix any j where j is the index of an active guard. We prove $\forall t \in [0, t_r],\ g_j(t) \leq 0$ by contradiction. Assume there exists $t_1 \in [0, t_r]$ such that $g_j(t_1) > 0$.

First, we show that g_j is strictly negative at some time strictly after 0:

- If $g_j(0) < 0$, then g_j is negative at time 0.
- If $g_j(0) = 0$ and $\dot{g}_j(0) < 0$, then by differentiability at 0 there exists $\delta > 0$ such that for all $t \in (0, \delta]$, $g_j(t) < 0$.

In either case, there exists some $t_- \in [0, t_1)$ with $g_j(t_-) < 0$.

Assume towards a contradiction that g_j becomes positive at or before t_r. Then there exists $t_1 \leq t_r$ such that $g_j(t_1) > 0$. Let

$$\tau \triangleq \inf\{t \in [0, t_1] \mid g_j(t) > 0\}.$$

By continuity, $g_j(\tau) = 0$, and by definition of τ there are arbitrarily close times after τ where g_j is positive. Also, since $g_j(t_-) < 0$ for some $t_- < \tau$, there are negative values somewhere before τ.

Now consider the maximal interval on which g_j is 0 ending at τ: choose any $a \leq \tau$ such that $g_j(t) = 0$ for all $t \in [a, \tau]$ and a is minimal with this property. Then:

- $\forall t \in [a, \tau]$, $g_j(t) = 0$ by construction;
- for every $\varepsilon > 0$, there exists $t \in [a - \varepsilon, a)$ with $g_j(t) < 0$ (by minimality of a);
- for every $\varepsilon > 0$, there exists $t \in (\tau, \tau + \varepsilon]$ with $g_j(t) > 0$ (by definition of τ).

These are exactly the conditions of Def. 2, so τ is an upward zero-crossing time of g_j. Finally, because $g_j(t_1) > 0$ occurs at some $t_1 \leq t_r$, we have $\tau < t_r$, so we have found an upward zero-crossing strictly before t_r. This contradicts that t_r is the *first* upward zero-crossing among the monitored signals (Def. 4). Hence $g_j(t) \leq 0$ for all $t \in [0, t_r]$.

Since j was arbitrary among the active guards, all conjuncts produced by $\text{Active}(\rho(0)(x), \{u_1, \ldots, u_n\})$ are preserved up to t_r.

Moreover, since t_r is an upward zero-crossing time of g_i (Def. 2), in particular $g_i(t_r) = 0$, i.e. $u_i(\rho(t_r)(x)) = 0$.

Remark (Non-active guards may trigger first). Note that Lemma 1 does *not* require that the first triggered event comes from an active guard. Even if some initially non-active guard triggers the first up-crossing at time t_r, the proof above shows that *every* guard that *was* active at time 0 still cannot become positive before t_r (otherwise it would generate an earlier up-crossing), matching the stated intuition.

C Proof of Lemma 2

Proof. Fix an active guard index j and let $g_j(t) = u_j(\rho(t)(x))$ as in B. As in Lemma 1, activity at time 0 implies g_j is negative at some time $t_- \geq 0$.

Assume for contradiction that $g_j(t_1) > 0$ for some $t_1 \geq 0$. Define $\tau = \inf\{t \in [0, t_1] \mid g_j(t) > 0\}$. The same continuity argument as before shows that τ satisfies Def. 2, hence τ is an upward zero-crossing time of g_j.

But this contradicts the assumption that the flow segment has no monitored upward zero-crossing at any finite time (the $+\infty$ outcome of flow). Therefore $g_j(t) \leq 0$ for all $t \geq 0$.

D Proof of Continuous Type Preservation

Proof. By structural induction on the operational semantics derivation of $S; \sigma \vdash e \overset{(\rho, t_r)}{\hookrightarrow} e'$.

Case S-LETREC-DER-FIN. Assume $\Gamma \vdash e : \{v : b \mid \Box\, \phi(v)\}$ and $S; \sigma \vdash e \overset{(v(t), t_r); v_r}{\hookrightarrow} e'$ is derived by S-LETREC-DER-FIN, where

1. $flow(\mathsf{der}\ x = e_f,\ v_0,\ \{u_1(x), \ldots, u_n(x)\}) \Downarrow (\rho, t_r)$ with $t_r < +\infty$,
2. there exists an index j whose guard triggers at t_r (first upward zero-crossing),
3. and the reset branch steps in the discrete fragment under the post-flow store $S; \sigma$, $[x \mapsto \rho(t_r)(x) :: \mathsf{nil}]$, producing a rewritten reset expression $de'_{r,j}$, and the equation residual e' is obtained by replacing the initial value with the post-reset state and the reset branch to $de'_{r,j}$, where $v_{r,j}$ is emitted.

(1) Safety of the finite continuous trajectory. T-DER-FIN must be the last (non-subtyping) rule used. By inversion, we obtain its premises.

(T1) There exists an auxiliary invariant φ_I such that typing e_0 establishes: (i) the safety predicate φ at time 0, (ii) the differential-invariant side condition needed by Platzer's rule (dI) for φ_I, and (iii) a bridge implication from φ_I together with the Active$(v, u_1(x)...u_n(x))$ to φ.

From the flow premise and Lemma 1 due to a finite t_r, we have:

$$\forall t \in [0, t_r].\ \text{Active}(\rho(t)(x)) \qquad \text{and} \qquad u_i(\rho(t_r)(x)) = 0.$$

Now apply the soundness of Platzer's differential invariant rule (dI) [27]: premise (T1) provides the initial establishment of φ_I together with its differential side condition, and Lemma 1 supplies Active throughout $[0, t_r]$. Therefore φ_I holds at all times in $[0, t_r]$. Finally by assumption from (T1),

$$\Box_{[0,t_r]}\phi(v(t))$$

We then show $\Gamma_c \vdash e' : \{v : b \mid \Box\phi(v)\}$ by re-applying T-Der-Fin to the rewritten program.

(Body premise.) The typing premise for the body (call it (T3)) is unchanged: the binder for x in the residual remains $x : \{v : b \mid \phi(v)\}$, hence the same derivation establishes the body typing.

(Reset premise and updated initializer.) Since branch j triggers at time t_r, then we have $u_j(x_r) = 0$ from Lemma 1. Together with (1) we have $x_r \models \phi$. Therefore, the runtime store used to evaluate the reset expression, $\sigma[x \mapsto x_r]$, satisfies the assumptions of the typing premise (T2) for branch i, which types $de_{r,j}$ under

$$x : \{v : b \mid \phi(v) \wedge u_i(v) = 0\}.$$

Since the predicate syntax of the hybrid type system is a subset of those in the purely discrete language, correspondence of the typing and term environments in the hybrid context implies correspondence in the discrete context as well. By the assumed discrete preservation for the non-continuous fragment, from $S; \sigma, [x^m \mapsto \rho(t_r)(x^m) :: \mathsf{nil}] \vdash de_{r,j} \overset{v_{r,j}}{\hookrightarrow} de'_{r,j}$ we obtain that the result $v_{r,j}$ satisfies the refinement postcondition stated by (T2). This also uses the fact that $\mathsf{tl}(\Box p) \equiv \Box p$. In particular, since (T2) bundles the conjunct $\phi(v)$ in the type of $de_{r,i}$, we conclude

$$v_{r,j} \models \phi.$$

All other reset expressions retain their previous typing judgments due to them not stepping. Finally, S-Letrec-Der-Fin updates the initializer of the residual program to be this post-reset value $v_{r,j}$. Hence, the initializer premise required to re-apply T-Der-Fin holds for ce'.

Combining the unchanged body premise with the updated initializer and the preserved reset-branch typing, all premises of T-Der-Fin hold for ce', and therefore

$$\Gamma_c \vdash v_{r,j} : \{v : b \mid \Box\phi(v)\} \qquad \textit{and} \qquad \Gamma_c \vdash ce' : \{v : b \mid \Box\phi(v)\}.$$

This completes the S-LETREC-DER-FIN case.

Case S-LETREC-DER-INF. The last operational rule is S-Der-Inf. Thus

$$\mathit{flow}(\mathsf{der}\ x = e_f,\ x_0,\ \{u_1, \ldots, u_n\}) \Downarrow (\rho, +\infty),$$

and the residual is the distinguished infinite marker $[\infty]$.

We must show:

1. $[\![v]\!]\rho \vDash \Box_{[0,+\infty)}\phi(v)$, i.e. $\forall t \geq 0.\ \rho(t)(x) \models \phi$;
2. $\Gamma_c \vdash [\infty] : \{v : b \mid \Box\phi(v)\}$.

(1) Safety of the infinite continuous trajectory. By inversion, the last typing rule is again T-Der-Fin (or the corresponding continuous typing rule for derivatives), yielding premise (T1) with an auxiliary invariant φ_I and a bridge. From Lemma 2 we have domain preservation for all time:

$$\forall t \geq 0.\ \mathrm{Active}(\rho(t)(x)).$$

By soundness of (dI), the differential side condition from (T1) implies that φ_I is preserved along ρ for all $t \geq 0$ (since Active holds for all $t \geq 0$). Applying the bridge pointwise yields:

$$\forall t \geq 0.\ \rho(t)(x) \models \varphi,$$

i.e. $\rho \models \Box_{[0,+\infty)}\varphi$.

(2) Residual typing. The residual is $[\infty]$ (or equivalent). By the typing rule for the inert infinite residual, we derive directly:

$$\Gamma_c \vdash [\infty] : \{v : b \mid \Box\varphi(v)\}.$$

This concludes both cases.

E Detailed Proofs of Hybrid Examples

We provide detailed proofs of the two examples presented in the text, and introduce two additional examples.

E.1 Proof of the Water Tank Example

$$\begin{aligned}\mathrm{T3} \equiv \Gamma,\ &(\mathit{level}, \mathit{flow}) : \{(l,f) : \mathit{fl} \times \mathit{fl} \mid \Box(1 \leq l \leq 9)\} \ \vdash \\ &\mathit{level} : \{l : \mathit{fl} \mid \Box(1 \leq l \leq 9)\}\end{aligned}$$

$$\begin{aligned}\mathrm{T1} \equiv \Gamma,\ &\mathit{level} : \{v : \mathit{fl} \mid \Box(1 \leq v \leq 9)\},\ \dot{\mathit{level}} : \{v : \mathit{fl} \mid v = \Box\mathit{flow}\} \ \vdash \\ &(5,5) : \Big\{(l,f) : \mathit{fl} \times \mathit{fl} \mid \Box(1 \leq l \leq 9) \wedge \\ &\dot{\mathit{True}} \wedge \big((\mathit{True} \wedge \Box(1 \leq l \leq 9)) \Rightarrow \Box(1 \leq \mathit{level} \leq 9)\big)\Big\}\end{aligned}$$

$$\begin{aligned}\mathrm{T2}_1 \equiv \Gamma,\ &\mathit{level} : \{\Box(v \mid 1 \leq v \leq 9) \wedge v = 9\},\ \dot{\mathit{level}} : \{v \mid v = \Box\mathit{flow}\} \ \vdash \\ &(9,-5) : \Big\{(l,f) : \mathit{fl} \times \mathit{fl} \mid \Box(1 \leq l \leq 9) \wedge \Box(\dot{\mathit{level}} \leq 9) \wedge \\ &\big((\Box(\mathit{level} \leq 9) \wedge \Box(1 \leq l \leq 9)) \Rightarrow \Box(1 \leq \mathit{level} \leq 9)\big)\Big\}\end{aligned}$$

$$\mathrm{T2}_2 \equiv \Gamma,\ level : \{v \mid \Box(1 \le v \le 9) \wedge v = 1\},\ \dot{level} : \{v \mid v = \Box flow\} \vdash$$
$$(1,5) : \Big\{(l,f) : fl \times fl \mid \Box(1 \le l \le 9) \wedge \Box(1 \le \dot{l}evel) \wedge$$
$$\big((\Box(1 \le level) \wedge \Box(1 \le l \le 9)) \Rightarrow \Box(1 \le level \le 9)\big)\Big\}$$

$$\text{T-DER-FIN}\ \frac{\mathrm{T1} \quad \mathrm{T2}_1 \quad \mathrm{T2}_2 \quad \mathrm{T3}}{\begin{array}{c}\Gamma \vdash \text{let rec der (level, flow)} : \{(l,f) : fl \times fl \mid \Box(1 \le l \le 9)\} \\ = \text{(flow, 0) init (5 , 5) reset}\left|\begin{array}{l}\text{up(level-9)}\rightarrow\text{(last level,}-5\text{)} \\ \text{up(-level+1)}\rightarrow\text{(last level,5)}\end{array}\right. \\ \text{in level} : \{l : fl \mid \Box(1 \le l \le 9)\}\end{array}}$$

E.2 Proof of the Event-Triggered Automatic Braking Example

Let *prog* =

```
1  let x_obs = 5.; brake = -0.2
2  let rec der (x,v,a)  =
3      (v init 0.,
4       a init 1.
5       0. init 1.) reset
6    | up(x -. (v *. v /. (2. *. brake)) +. 0.1 -. x_obs) ->
7        (last x, last v, brake)
8    | up(-. v ) -> (last x, last v, 0.)
9  in (x, v, a)
```

The program initially starts with both guards $x - \frac{v^2}{2\times brake} + 0.1 - x_{obs}$ and $-v$ negative; thus the differential invariant is not necessary to prove safety since these alone imply the main invariant.

Upon a zero-crossing on $x - \frac{v^2}{2\times brake} + 0.1 - x_{obs}$, the acceleration is set to $brake$. However, since x and v retain their previous values and the derivative of this guard is zero, this guard is no longer active. Therefore, use the differential invariant to prove safety. This is done by observing that, since $brake$ is negative, $v - \frac{va}{brake} \le 0$ only if a is at least as negative as $brake$. This is obviously true so the differential invariant is provable.

Upon a zero-crossing on $-v$, the acceleration is set to 0. As before, neither this guard nor the other can be proven to be active after reset, so the differential invariant is used again. Since $v = 0$, $v - \frac{va}{brake} \le 0$ trivially and furthermore it is provable to stay at zero because acceleration is also zero, thus guaranteeing safety.

Applying (T-DER-FIN):

$$T3 \equiv \quad \Gamma,\ (x,v,a) : \{(v_x,v_v,v_a) : fl \times fl \times fl \mid \Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs})\}$$
$$\vdash\ (x,v,a) : \{(v_x,v_v,v_a) : fl \times fl \times fl \mid \Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs})\}$$

For $e_0 = (0, 1, 1)$, select $\phi_I = True$

T1≡

$$\Gamma,\ (x, v, a) : \{(v_x, v_v, v_a) : fl \times fl \times fl \mid \Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs})\},$$
$$(\dot{x}, \dot{v}, \dot{a}) : \{(w_x, w_v, w_a) : fl \times fl \times fl \mid \Box(w_x, w_v, w_a) = (v, a, 0)\} \vdash$$
$$(0, 1, 1) : \Big\{(v_x, v_v, v_a) : fl \times fl \times fl \mid \Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs}) \wedge$$
$$\dot{True} \wedge \big((True \wedge \Box(v_v \geq 0 \wedge v_x - \frac{v_v^2}{2 \times brake} + 0.1 \leq x_{obs})) \Rightarrow$$
$$\Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs})\big)\Big\}$$

For this reset, choose $\phi_I \equiv (v_x - \frac{v_v^2}{2 \times brake} < x_{obs})$

$\mathrm{T2}_1 \equiv$

$$\Gamma,\ (x, v, a) : \{(v_x, v_v, v_a) : fl \times fl \times fl \mid \Box((v_x - \frac{v_v^2}{2 \times brake} < x_{obs}) \wedge$$
$$(v_x - \frac{v_v^2}{2 \times brake} + 0.1 = x_{obs}))\},$$
$$(\dot{x}, \dot{v}, \dot{a}) : \{(w_x, w_v, w_a) : fl \times fl \times fl \mid \Box(w_x, w_v, w_a) = (v, a, 0)\} \vdash$$
$$(\mathsf{last}\ x, \mathsf{last}\ v, brake) : \Big\{(v_x, v_v, v_a) : fl \times fl \times fl \mid \Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs}) \wedge$$
$$\Box v_v - \frac{v_v v_a}{b} \leq 0 \wedge \big((\Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs}) \wedge True) \Rightarrow$$
$$\Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs})\big)\Big\}$$

Choose the same ϕ_I for this reset

$\mathrm{T2}_2 \equiv$

$$\Gamma,\ (x, v, a) : \{(v_x, v_v, v_a) : fl \times fl \times fl \mid$$
$$\Box((v_x - \frac{v_v^2}{2 \times brake} < x_{obs}) \wedge (v_v = 0))\},$$
$$(\dot{x}, \dot{v}, \dot{a}) : \{(w_x, w_v, w_a) : fl \times fl \times fl \mid \Box(w_x, w_v, w_a) = (v, a, 0)\} \vdash$$
$$(\mathsf{last}\ x, \mathsf{last}\ v, 0) : \Big\{(v_x, v_v, v_a) : fl \times fl \times fl \mid \Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs}) \wedge$$
$$\Box v_v - \frac{v_v v_a}{b} \leq 0 \wedge \big((\Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs}) \wedge True) \Rightarrow$$
$$\Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs})\big)\Big\}$$

T-DER-FIN

$$\frac{\mathrm{T1} \qquad \mathrm{T2}_1 \qquad \mathrm{T2}_2 \qquad \mathrm{T3}}{\Gamma \vdash prog : \{(v_x, v_v, v_a) : fl \times fl \times fl \mid \Box(v_x - \frac{v_v^2}{2 \times brake} < x_{obs})\}}$$

E.3 Additional Examples and Proofs

Decreasing Sawtooth Function We begin with a program which produces a "decreasing sawtooth" signal. The signal's level increases continuously in a linear fashion until it reaches an upper limit ($x = 0$), after which it is reset to a lower value which is lower than the upper limit and decreases with each reset. A useful property to prove is that the signal's level never exceeds 0.

```
1  let rec der x = 1. init -1 reset up(x) ->
2     (let rec xr = -2. fby (xr -. 1.) in xr)
3  in x
```

Ensure: $x \leq 0$ at all times
The invariant, $x \leq 0$ can be proven directly by using the fact that the zero-crossing detector $\mathsf{up}(x)$ is active if the continuous segment begins with x at any value less than 0. This is clearly true for the initial condition, a constant -1. The reset expression can be proven to always be negative using discrete typing rules. Specifically, the (T-LETREC) and (T-FBY) rules from [12] can be used to show that, since x_r was initialized to -2, then subtracting 1 from x_r will always yield a negative number thus allowing us to show $\Box(x_r < 0)$ and thus not only the invariant is satisfied for the reset but that the guard remains active after the reset (since $x < 0$).

Proof of the Sawtooth Example

$$
\begin{aligned}
&\text{T1} \equiv \\
&\Gamma,\ x : \{v : \mathsf{fl} \mid \Box v \leq 0\},\ \dot{x} : \{v : \mathsf{fl} \mid \Box v = 1.0\} \ \vdash \\
&-1.0 : \left\{ v : \mathsf{fl} \;\middle|\; \Box(v \leq 0) \wedge \dot{True} \wedge \big((True \wedge \Box(v \leq 0)) \Rightarrow \Box(x \leq 0)\big) \right\}
\end{aligned}
$$

$$
\begin{aligned}
&\text{T2} \equiv \\
&\Gamma,\ x : \{v : \mathsf{fl} \mid \Box v \leq 0\},\ \dot{x} : \{v : \mathsf{fl} \mid \Box v = 1.0\},\ x_r : \{v | \Box v < 0\} \ \vdash \\
&x_r : \left\{ v : \mathsf{fl} \;\middle|\; \Box(v \leq 0) \wedge \dot{True} \wedge \big((True \wedge \Box(v \leq 0)) \Rightarrow \Box(x \leq 0)\big) \right\}
\end{aligned}
$$

$$
\begin{aligned}
&\text{T3} \equiv \\
&\Gamma,\ x : \{v : \mathsf{fl} \mid \Box\, v \leq 0\} \ \vdash \ x : \{v : \mathsf{fl} \mid \Box\, v \leq 0\}
\end{aligned}
$$

T-DER-FIN

$$
\frac{\text{T1} \qquad \text{T2} \qquad \text{T3}}{\begin{array}{c}\Gamma \vdash \text{let rec der } x : \{v : \mathsf{fl} \mid \Box\, v \leq 0\} = 1.0 \text{ init -1.0 reset} \\ \mid\ \text{up}(x) \rightarrow (\text{let rec } x_r : \{v | v < 0\} = \text{-2. fby } (x_r \text{ -. } 1.) \text{ in } x_r) \\ \texttt{in}\ \ x : \{v : \mathsf{fl} \mid \Box\, v \leq 0\}\end{array}}
$$

Proof that $\square(x_r < 0)$: Define $\tau \equiv \{v : \mathsf{float} \mid \square(v < 0)\}$ for conciseness.
$T_{2,4} \equiv \Gamma, x_r : \tau \vdash (x_r - . 1.) : \{v : \mathsf{float} \mid \square(v = x_r - 1)\}$
(Proven trivially via type synthesis of basic arithmetic operators)

$T_{2,3} \equiv$

$$\frac{\dfrac{}{\Gamma, x_r : \tau \vdash -2. : \{v : \mathsf{float} \mid \square(v = -2)\}}\ (\text{T-CONST}) \qquad T_{2,4}}{\Gamma, x_r : \tau \vdash -2.\ \mathsf{fby}\ (x_r - . 1.) : \{v : \mathsf{float} \mid \mathsf{hd}(v = -2) \wedge \bigcirc\square(v = x_r - 1)\}}\ (\text{T-FBY})$$

(Trivial application of subtyping applied for $\{v : \mathsf{float} \mid \square(v = -2)\} \preceq \{v : \mathsf{float} \mid (v = -2)\}$
$T_{2,1} \equiv$

$$\frac{T_{2,3} \qquad \{v : \mathsf{float} \mid \mathsf{hd}(v = -2) \wedge \bigcirc\square(v = x_r - 1)\} \preceq \tau \qquad \Gamma, x_r : \tau \vdash \tau}{\Gamma, x_r : \tau \vdash -2.\ \mathsf{fby}\ (x_r - . 1.) : \tau}\ (\text{T-SUB})$$

(Subtyping obligation is proven by noting that $\forall x_r : \mathsf{float}\ .\ x_r < 0 \implies (x_r - 1) < 0$, and that $2 < 0$).
$T_{2,2} \equiv$

$$\frac{(\Gamma, x_r : \tau)(x_r) = \tau}{\Gamma, x_r : \tau \vdash x_r : \tau}\ (\text{T-VAR})$$

$$\frac{\Gamma, x_r : \tau \vdash \tau \qquad T_{2,1} \qquad T_{2,2}}{\Gamma \vdash \text{let rec } x_r : \tau = -2.\ \mathsf{fby}\ (x_r - . 1.) \text{ in } x_r : \tau}\ (\text{T-LETREC})$$

Bouncing Ball The bouncing ball is a classic hybrid system which, while not necessarily controlled by a software component, nevertheless demonstrates a system containing both continuous dynamics and a discrete event which resets the system's state. Here, the ball is released from a positive height and allowed to free-fall until it reaches the ground, at which point it bounces up with some energy lost during the bounce. We verify that the ball never travels through the ground, and that, obeying the laws of physics, does not bounce higher than its initial height. Safety of this system is proven by using both the reset condition and differential reasoning similarly to [26] to ensure the ball obeys conservation of energy.

```
1  let y0 = 10.0; v0 = 0.0; g = 9.81
2  let rec der (y, v) =
3      (v init y0,
4      -.g init v0) reset up(-.y) -> (last y, -0.8 *. last v)
5   in (y, v)
```

Ensure: $0 \leq y \leq y_0$
The proof applies rule (T-DER-FIN) to show that the ball's height never becomes negative and remains within the specified bounds. The bouncing ball can be proven safe by splitting the invariant; $y \geq 0$ can be proven entirely using the fact that $\mathsf{up}(-y)$ is always active. Similarly to [26], $y \leq y_0$ can be proven using the invariant $2gy \leq 2gy_0 - v^2$ which is just a reformulation of the potential and kinetic energies of the ball. We

verify the invariant $y \geq 0 \wedge 2gy \leq 2gy_0 - v^2$. The invariant combines a geometric safety condition on the height with an energy bound relating height and velocity. Since $Active(y_0, \{y\})$ lets us directly prove $y \geq 0$, we select $\phi_I \equiv 2gy \leq 2gy_0 - v^2$, and subsequently note that $\phi_i' \equiv -2gv \leq -2gv$ During continuous evolution under gravity, the energy component of the invariant is preserved by the trivially true differential invariant ϕ_i', while the zero-crossing guard prevents the height from crossing below zero. When the guard triggers at ground contact, the reset maps the state to a strictly lower-energy configuration with upward velocity (thus $(-0.8v)^2 \leq v^2 \leq 2gy_0$), immediately re-establishing the invariant and providing a safe initial state for the next segment. By induction over alternating continuous evolutions and resets, the invariant holds globally.

Proof of the Bouncing Ball Example

$$
\begin{aligned}
&T3 \equiv \\
&\Gamma,\ (y,v) : \{(u,w) : fl \times fl \mid \Box(0.0 \leq u \leq 10.0 \wedge 2gy \leq 2gy_0 - v^2)\} \vdash \\
&y : \{u : fl \mid \Box(0.0 \leq u \leq 10.0)\}
\end{aligned}
$$

$$
\begin{aligned}
&T1 \equiv \\
&\Gamma,\ y : \{u \mid \Box u \geq 0\},\ \dot{y} : \{u \mid \Box u = v\},\ \dot{v} : \{w \mid \Box w = -9.81\} \vdash \\
&(10.0, 0.0) : \Big\{(u,w) \mid \Box(0.0 \leq u \leq 10.0 \wedge 2gu \leq 2gy_0 - w^2) \\
&\wedge \Box((y \leq 10.0) \wedge (2gy \leq 2gy_0 - v^2))' \wedge \\
&\big(\Box(((y \leq 10.0) \wedge (2gy \leq 2gy_0 - v^2)) \wedge \Box(u \geq 0.0)) \Rightarrow \Box(0.0 \leq y \leq 10.0)\big)\Big\}
\end{aligned}
$$

$$
\begin{aligned}
&\text{T2} \equiv \\
&\Gamma,\ y : \{u \mid \Box(u \geq 0) \wedge u = 0\},\ \dot{y} : \{u \mid \Box u = v\} \vdash \\
&(0.0, -0.8 \cdot v) : \Big\{(u,w) \mid \Box(0.0 \leq u \leq 10.0 \wedge 2gu \leq 2gy_0 - w^2) \\
&\wedge \Box((0.0 \leq y) \wedge (y \leq 10.0) \wedge (2gy \leq 2gy_0 - v^2))' \\
&\wedge \big(\Box(((0.0 \leq y) \wedge (y \leq 10.0) \wedge (2gy \leq 2gy_0 - v^2)) \wedge True) \Rightarrow \\
&\Box(0.0 \leq u \leq 10.0 \wedge 2gy \leq 2gy_0 - v^2)\big)\Big\}
\end{aligned}
$$

T-DER-FIN

$$
\frac{\text{T1} \qquad \text{T2} \qquad \text{T3}}{\begin{array}{c}\Gamma \vdash \text{let rec der (y,v)} : \{(u,w) : fl \times fl \mid \Box(0.0 \leq u \leq 10.0 \wedge 2gy \leq 2gy_0 - v^2)\} \\ = (v, -9.81) \text{ init } (10.0, 0.0) \\ \text{up(-y)} \rightarrow \text{(last y, -0.8 * last v) } \texttt{in}\ y : \{u : fl \mid \Box(0.0 \leq u \leq 10.0)\}\end{array}}
$$