A Type-based Analysis of Causality Loops in Hybrid Systems Modelers

Albert Benveniste, Timothy Bourke, Benoît Caillaud, Bruno Pagano, Marc Pouzet

A Type-based Analysis of Causality Loops in Hybrid Systems Modelers

Albert Benveniste[a], Timothy Bourke[b,c], Benoit Caillaud[a], Bruno Pagano[e], Marc Pouzet[d,c,b,*]

[a] *Inria Rennes-Bretagne Atlantique*
[b] *Inria Paris*
[c] *Département d'Informatique, Ecole normale supérieure*
[d] *Université Pierre et Marie Curie*
[e] *ANSYS/Esterel Technologies*

Abstract

Explicit hybrid systems modelers like *Simulink/Stateflow* allow for programming both discrete- and continuous-time behaviors with complex interactions between them. An important step in their compilation is the static detection of algebraic or *causality* loops. Such loops can cause simulations to deadlock and prevent the generation of statically scheduled code.

This paper addresses this issue for a hybrid modeling language that combines synchronous data-flow equations with Ordinary Differential Equations (ODEs). We introduce the operator `last` x for the left-limit of a signal x. The `last` x operator is used to break causality loops and permits a uniform treatment of discrete and continuous state variables. The semantics of the language relies on non-standard analysis, defining an execution as a sequence of infinitesimally small steps. A signal is deemed *causally correct* when it can be computed sequentially and only changes infinitesimally outside of announced discrete events like zero-crossings. The causality analysis takes the form of a type system that expresses dependencies between signals. In well-typed programs, signals are *provably continuous during integration* provided that imported external functions are also continuous, and *sequential code can be generated*.

The effectiveness of the system is illustrated with several examples written in Zélus, a Lustre-like synchronous language extended with ODEs.

Keywords: Hybrid systems, Synchronous programming languages, Type systems

*Corresponding author
Email addresses: `Albert.Benveniste@inria.fr` (Albert Benveniste), `Timothy.Bourke@inria.fr` (Timothy Bourke), `Benoit.Caillaud@inria.fr` (Benoit Caillaud), `Bruno.Pagano@esterel-technologies.com` (Bruno Pagano), `Marc.Pouzet@ens.fr` (Marc Pouzet)

1. Causality and Scheduling

Tools for modeling hybrid systems [1] such as MODELICA,[1] LABVIEW,[2] and SIMULINK/STATEFLOW,[3] are now rightly understood and studied as programming languages. Indeed, models are used not only for simulation, but also for test-case generation, formal verification and translation to embedded code. This explains the need for formal operational semantics for specifying their implementations [2].

The underlying mathematical model is the synchronous parallel composition of stream equations, Ordinary Differential Equations (ODEs), hierarchical automata, and imperative features. While each of these taken separately is precisely understood, real languages allow them to be combined in sophisticated ways. One major difficulty in such languages is the treatment of causality loops.

Causality or *algebraic* loops [3, 2-34] pose problems of well-definedness and compilation. They can lead to mathematically unsound models, prevent simulators from statically ensuring the existence and unicity of a least fixed point, and compilers from generating statically scheduled code (typically sequential code written in C). Statically scheduled code is the usual target for embedded software. But it is also key to get efficient simulations of the whole system where continous-time trajectories are approximated by a numerical solver. The static detection of causality loops, termed *causality analysis*, has been studied and implemented since the mid-1980s in synchronous language compilers [4, 5, 6, 7]. The classical and simplest solution is to reject instantaneous cycles or feed-back loops, which do not cross a unit delay: at every instant, the value of a signal x only depend on the current value of inputs and possibly some internal state, but not of x itself. For instance, the LUSTRE-like equations:[4][5]

```
x = 0.0 -> pre y    and    y = if c then x + 1.0 else x
```

define the two sequences $(x_n)_{n\in\mathbb{N}}$ and $(y_n)_{n\in\mathbb{N}}$ such that:

$$x(0) = 0 \qquad\qquad y(n) = \texttt{if}\, c(n)\, \texttt{then}\, x(n) + 1\, \texttt{else}\, x(n)$$
$$x(n) = y(n-1)$$

They are causally correct since the feedback loop for `x` contains a unit delay `pre y` ('previous'). Replacing `pre y` with `y` gives two non-causal equations that the LUSTRE detects and rejects. Causally correct equations are statically scheduled to produce a sequential, loop-free *step* function. Below is an excerpt of the C code generated by the HEPTAGON compiler [8] of LUSTRE:

---

[1] http://www.modelica.org

[2] http://www.ni.com/labview

[3] http://www.mathworks.com/products/simulink

[4] The unit delay $0\,\texttt{->}\,\texttt{pre}(\cdot)$, initialized to 0, is sometimes written as $0\ \texttt{fby}\ \cdot$ ('0 followed by'), or in SIMULINK: $\frac{1}{z}$.

[5] Examples given in this paper together with more detailled ones are available at http://zelus.di.ens.fr/nahs2016/. Follow ♣ links.

```
if (self->v_1) {x = 0;} else {x = self->v_2;};
if (c) {y = x+1;} else {y = x;};
self->v_2 = y; self->v_1 = false;
```

It computes current values of `x` and `y` from that of `c`. The internal memory of function *step* is in `self`, with `self->v_1` initialized to true and set to false (to encode the LUSTRE operator `->`) and `self->v_2` storing the value of `pre y`.

ODEs with resets: Consider now the situation of a program defining continuous-time signals only, made of ODEs and equations. For example:

```
der y = z init 4.0 and z = 10.0 - 0.1 * y and k = y + 1.0
```

defines signals y, z and k, where for all $t \in \mathbb{R}^+$, $\frac{dy}{dt}(t) = z(t)$, $y(0) = 4$, $z(t) = 10 - 0.1 \cdot y(t)$, and $k(t) = y(t) + 1$.[6] This program is causally correct since it is possible to generate a sequential function $\mathit{derivative}(y) = \texttt{let}\, z = 10 - 0.1 * y\, \texttt{in}\, z$ that returns the current derivative of y and an initial value 4 for y from which a numeric solver [9] can compute a sequence of approximations $y(t_n)$ for increasing values of time $t_n \in \mathbb{R}^+$ and $n \in \mathbb{N}$. Given a set of mutually recursive equations $\{x_i = e_i\}_{i \in [1..k]}$ and $\{\dot{y_j} = e'_j\}_{j \in [1..m]}$, the compiler has to produce the *derivative* function that defines the current value of $(\dot{y_j})_{j \in [1..m]}$ from current inputs, discrete state variables and continuous state variables $(y_j)_{j \in [1..m]}$. Thus, for equations between continuous-time signals, integrators break algebraic loops just as delays do for equations over discrete-time signals.

Can we reuse the simple justification we used for data-flow equations to justify that the above program is causal? Consider the value that y would have if computed by an ideal solver taking an infinitesimal step of duration ∂ [10]. Writing ${}^\star y(n)$, ${}^\star z(n)$ and ${}^\star k(n)$ for the values of y, z and k at instant $n\partial$, with $n \in {}^\star\mathbb{N}$ a non-standard integer, we have:

$$\begin{aligned} {}^\star y(0) &= 4 & {}^\star z(n) &= 10 - 0.1 \cdot {}^\star y(n) \\ {}^\star y(n+1) &= {}^\star y(n) + {}^\star z(n) \cdot \partial & {}^\star k(n) &= {}^\star y(n) + 1 \end{aligned}$$

where ${}^\star y(n)$ is defined sequentially from past values and ${}^\star y(n)$ and ${}^\star y(n+1)$ are infinitesimally close, for all $n \in {}^\star\mathbb{N}$, yielding a unique solution for y, z and k. The equations are thus causally correct.

Troubles arise when ODEs interact with discrete-time constructs, for example when a reset occurs at every occurrence of an event. For example, consider the sawtooth signal $y : \mathbb{R}^+ \mapsto \mathbb{R}^+$ where $\frac{dy}{dt}(t) = 1$ and $y(t) = 0$ when $t \in \mathbb{N}$. One may try with an ODE and a reset

```
der y = 1.0 init 0.0 reset up(y - 1.0) -> 0.0
```

where `y` is initialized with 0.0, has derivative 1.0, and is reset to 0.0 every time the zero-crossing `up(y - 1.0)` is true, that is, whenever `y - 1.0` crosses 0.0

[6] $\texttt{der}\, y = e\, \texttt{init}\, v_0$ stands for $y = \frac{1}{s}(e)$ inititialized to v_0 in SIMULINK.

from negative to positive. But is this program causal? Again, consider the value y would have were it calculated by an ideal solver taking infinitesimal steps of length ∂. The value of ${}^\star y(n)$ at instant $n\partial$, for all $n \in {}^\star\mathbb{N}$ would be:

$$\begin{array}{ll} {}^\star y(0) = 0 & {}^\star y(n) = \texttt{if}\ {}^\star z(n)\ \texttt{then}\ 0.0\ \texttt{else}\ {}^\star ly(n) \\ {}^\star ly(n) = {}^\star y(n-1) + \partial & {}^\star c(n) = ({}^\star y(n) - 1) \geq 0 \\ {}^\star z(0) = \texttt{false} & {}^\star z(n) = {}^\star c(n) \wedge \neg {}^\star c(n-1) \end{array}$$

With the above interpretation, this set of equations is not causally correct: the value of ${}^\star y(n)$ depends instantaneously on ${}^\star z(n)$ which itself depends on ${}^\star y(n)$. There are two ways to break this cycle: *(a)* consider that the effect of the zero-crossing is delayed by one cycle, that is, the test is made on ${}^\star z(n-1)$ instead of on $z(n)$ — equivalently, that the test is made on ${}^\star z(n)$ but the effect is on $z(n+1)$, that is, one step later — or, *(b)* distinguish the current value of ${}^\star y(n)$ from the value it would have had were it not reset, namely ${}^\star ly(n)$. Testing a zero-crossing of ly (instead of y), that is,

$${}^\star c(n) = ({}^\star ly(n) - 1) \geq 0,$$

gives a program that is causal since ${}^\star y(n)$ no longer depends instantaneously on itself. We propose writing this ♣[7]:

```
der y = 1.0 init 0.0 reset up(last y - 1.0) -> 0.0
```

where `last` y stands for ly, that is, the *left-limit* of y. In non-standard semantics [10], it is infinitely close to the previous value of y, and written ${}^\star ly(n) \approx {}^\star y(n-1)$. The modeling of a bouncing ball is another prototypical example where the left-limit is needed. Suppose that y_0, y'_0 and g are given constants. The signal y is the height of the ball, with initial position y_0 and speed y'_0.

```
    der y = y' init y0
and der y' = -g init y'0 reset up(-y) -> -0.8 * last y'
```

At the instant of the zero-crossing, ${}^\star y'(n)$ is reset with value $-0.8 \cdot {}^\star y'(n-1)$. Replacing `last` y' by y' would lead to a causality loop on y': at the instant where condition `up`$(-y)$ is taken, the equation defining y' would be $y' = -0.8 * y'$, so with an instantaneous loop.

To solve this, a simple convention is that at a reset instant, y' on the right-hand-side of an equation implicitely denotes the left limit of y', that is, $y' = -0.8 * y'$ stands for $y' = -0.8 * \texttt{last}\, y'$. The other is to replace equations by assignements, e.g., write $y' := -0.8 * y'$ and order the changes sequentially when several have to be made. The later is precisely what is done in hybrid automata [11]. These conventions may lead to ambiguities when systems are composed in parallel: the two parallel equations $y = -0.8 * x$ and $x = y$ (or

[7] The ♣'s link to http://zelus.di.ens.fr/nahs2016/.

$y := -0.8{*}x$ and $x := y$) would be interpreted as $y = -0.8{*}last x$ and $x = \texttt{last}\, y$ but what if the designer has in mind $y = -0.8 * \texttt{last}\, x$ and $x = y$ whose result differs? Instead, we propose to make the use of `last` . explicit and rely on a static checking that there is no instantaneous feedback and that the code can indeed be statically scheduled.

When a variable y is defined by its derivative, $\texttt{last}\, y$ corresponds to the so-called 'state port' of the integrator block $\frac{1}{s}$ of SIMULINK, which is introduced expressly to break causality loops like the ones above ♣.[8] According to the documentation [12, 2-685]:

> "The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset."

SIMULINK restricts the use of the state port. It is only defined for the integrator block and cannot be returned as a subsystem output: it may only be referred to in the same context as its integrator block and used to break algebraic loops. The use of the state port reveals subtle bugs in the SIMULINK compiler. Consider the SIMULINK model shown in Figure 1a with the simulation results given by the tool for `x` and `y` in Figure 1b. The model contains two integrators. The one at left, named 'Integrator0' and producing x, integrates the constant 1. The one at right, named 'Integrator1' and producing y, integrates x; its state port is fed back through a bias block to reset both integrators, and through a gain of -3 to provide a new value for Integrator0. The new value for Integrator1 comes from the state port of Integrator0 multiplied by a gain of -4. In our syntax ♣:

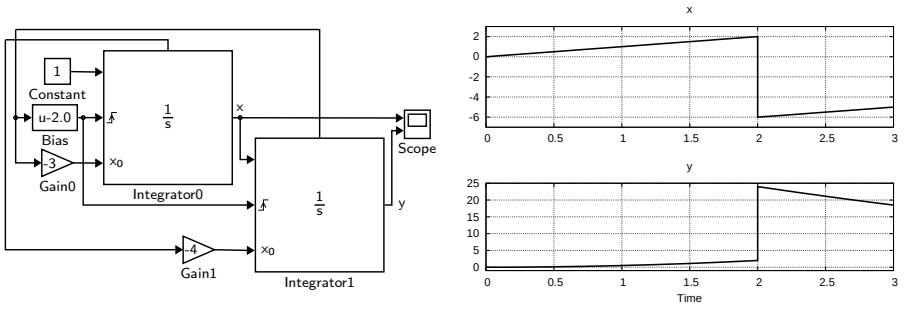
```
    der x = 1.0 init 0.0 reset z -> -3.0 * last y
and der y = x init 0.0 reset z -> -4.0 * last x
and z = up(last y - 2.0)
```

In the non-standard interpretation of signals, the equations above are perfectly causal: the current values of ${}^\star x(n)$ and ${}^\star y(n)$ only depend on previous values, that is:

$$\begin{aligned} {}^\star x(n) &= \texttt{if}\ {}^\star z(n)\ \texttt{then}\ -3 \cdot {}^\star y(n-1)\ \texttt{else}\ {}^\star x(n-1) + \partial \\ {}^\star y(n) &= \texttt{if}\ {}^\star z(n)\ \texttt{then}\ -4 \cdot {}^\star x(n-1) \\ &\quad\ \texttt{else}\ {}^\star y(n-1) + \partial \cdot {}^\star x(n-1) \end{aligned}$$

$$\begin{array}{llll} {}^\star x(0) = 0 & {}^\star z(0) = \texttt{false} & & {}^\star z(n) = {}^\star c(n) \wedge \neg {}^\star c(n-1) \\ {}^\star y(0) = 0 & {}^\star c(n) = ({}^\star y(n-1) - 2) \geq 0 & & \end{array}$$

[8] The SIMULINK integrator block outputs both an integrated signal and a state port. We write $(x, lx) = \frac{1}{s}(x_0, \texttt{up}(z), x')$ for the integral of x', reset with value x_0 every time z crosses zero from negative to positive, with output x and state port lx. The first example would thus be written: $(y, ly) = \frac{1}{s}(0.0, \texttt{up}(ly - 1.0), 1.0)$.



(a) Simulink model (b) Simulation results

Figure 1: A miscompiled Simulink model (R2009b) ♣

```
// P_0 = -2.0 P_1 = -3.0 P_2 = -4.0 P_3 = 1.0

static void mdlOutputs(SimStruct * S, int_T tid)
{ _rtX = (ssGetContStates(S));
  ...
  _rtB = (_ssGetBlockIO(S));

  _rtB->B_0_0_0 = _rtX->Integrator1_CSTATE + _rtP->P_0;
  _rtB->B_0_1_0 = _rtP->P_1 * _rtX->Integrator1_CSTATE;

  if (ssIsMajorTimeStep (S)) { ...
      if (zcEvent || ...)
        { (ssGetContStates (S))->Integrator0_CSTATE =
            _ssGetBlockIO (S)->B_0_1_0; } ...
  }

  (_ssGetBlockIO (S))->B_0_2_0 =
    (ssGetContStates (S))->Integrator0_CSTATE;
    _rtB->B_0_3_0 = _rtP->P_2 * _rtX->Integrator0_CSTATE;

  if (ssIsMajorTimeStep (S)) { ...
      if (zcEvent || ...)
       { (ssGetContStates (S))-> Integrator1_CSTATE =
            (ssGetBlockIO (S))->B_0_3_0; } ...
  } ...
}
```

Figure 2: Excerpt of C code produced by RTW (R2009b)

Yet, can you guess the behavior of the model and explain why the trajectories computed by SIMULINK are wrong?

Initially, both x and y are 0. At time $t = 2$, the state port of Integrator1 becomes equal to 2 triggering resets at each integrator as the output of block $u - 2.0$ crosses zero. The results show that Integrator0 is reset to -6 $(= 2 \cdot -3)$ and that Integrator1 is reset to 24 $(= -6 \cdot -4)$. The latter result is surprising since, at this instant, the state port of Integrator0 should also be equal to 2, and we would thus expect Integrator1 to be reset to -8 $(= 2 \cdot -4)$!

The SIMULINK implementation does not satisfy its documented behavior. Inspecting the C function which computes the current outputs, `mdlOutput` in Figure 2, reveals that the sequence of updates for the two integrators is incor-

```
model scheduling
  Real x(start = 0);
  Real y(start = 0);
equation

  der(x) = 1;
  der(y) = x;

  when y >= 2 then
    reinit(x, −3 ∗ pre(y));
  end when;

  when y >= 2 then
    reinit(y, −4 ∗ pre(x));
  end when;

end scheduling;
```

(a) with `pre`

```
model scheduling
  Real x(start = 0);
  Real y(start = 0);
equation

  der(x) = 1;
  der(y) = x;

  when y >= 2 then
    reinit(x, −3 ∗ y);
  end when;

  when y >= 2 then
    reinit(y, −4 ∗ x);
  end when;

end scheduling;
```

(b) `x` before `y`

```
model scheduling
  Real x(start = 0);
  Real y(start = 0);
equation

  der(x) = 1;
  der(y) = x;

  when y >= 2 then
    reinit(y, −4 ∗ x);
  end when;

  when y >= 2 then
    reinit(x, −3 ∗ y);
  end when;

end scheduling;
```

(c) `y` before `x`

```
model scheduling
  Real x(start = 0);
  Real y(start = 0);
equation

  der(x) = 1;
  der(y) = x;

  when y >= 2 then
    reinit(x, −3 ∗ y);
    reinit(y, −4 ∗ x);
  end when;

end scheduling;
```

(d) single `when`

Figure 3: Hybrid causality in Modelica

rectly scheduled.[9] At the instant when the zero-crossing is detected (conditions `ssIsMajorTimeStep(S)` and `zcEvent` are true), the state port of Integrator0 (stored in `sGetContStates(S)->Integrator0_CSTATE`) is reset using the state port value of Integrator1. Thus, Integrator1 does not read the value of Integrator0's state port (that is ${}^\star x(n-1)$) but rather the current value (${}^\star x(n)$) given an incorrect output. The SIMULINK model is not correctly compiled—another variable is required to store the value of ${}^\star x(n-1)$, just as a third variable is normally needed to swap the values of two others. We argue that such a program should either be scheduled correctly or trigger an error message. Providing a means to statically detect and explain causality issues is a key motivation of this paper. The static analysis presented in this paper accepts the SIMULINK example and it is correctly translated to sequential code. The example is rejected if either `last y` or `last x` are replaced, respectively, by `y` or `x`.

Reprogramming the model of Figure 1a in MODELICA and simulating it in OpenModelica or Dymola emphasizes the general nature and fundamental importance of the treatment of causality in hybrid modelers. A direct translation (Figure 3a) is accepted and correctly simulated by both tools.[10] Perhaps surprisingly, both compilers accept the program with its `pre` operators removed (Figure 3b). Simulation then gives the same result as in Figure 1b, but changing the order of the reinitializations (Figure 3c) gives a different result (`y` reset to -8 and `x` to 24).[11] The same behaviors are observed when both reinitializations are placed under a single trigger condition (Figure 3d). Causal loops are normal in MODELICA, but it is questionable to allow them during a discrete step

[9] The same issue exists in release R2015a.

[10] In Modelica, `pre x` is the left-limit of `x` whereas it is the unit delay of LUSTRE, which is only defined for streams (discrete-time signals). ZÉLUS keeps the LUSTRE interpretation for `pre x` and uses `last x` for the left limit.

[11] In Modelica, the `;` denotes the parallel composition of two equations.

if simulation results then depend on the order, or layout, of conceptually simultaneous constraint declarations. In this paper, though, we do not address the richer setting of Differential Algebraic Equations (DAEs), but limit ourselves to programs that mix synchronous programming constructs and ODEs.

Any loop in SIMULINK, whether of discrete- or continuous-time signals, can be broken by inserting the so-called memory block [12, 2-831].[12] If x is a signal, $\texttt{mem}(x)$ is a piecewise constant signal which refers to the value of x at the previous integration step (or *major* step). If those steps are taken at increasing instants $t_i \in \mathbb{R}$, $\texttt{mem}(x)(t_0) = x_0$ where $t_0 = 0$ and x_0 is an explicitly defined initial value, $\texttt{mem}(x)(t_i) = x(t_{i-1})$ for $i > 0$ and $\texttt{mem}(x)(t_i + \delta) = x(t_{i-1})$ for $0 \leq \delta < t_{i+1} - t_i$. As integration is performed globally, $\texttt{mem}(y)$ may cause strange behaviors as the previous value of a continuously changing signal x depends precisely on when the solver decides to stop! ♣ Writing $\texttt{mem}(y)$ is thus unsafe in general [13].[13] There is nonetheless a situation where the use of the memory block is mandatory and still safe: *the program only refers to the previous integration step during a discrete step.* This situation is not unusual. Consider, for example, a system that produces a signal x through two continuous modes M_1 and M_2, where on mode changes, the new mode restarts with the value computed previously by the solver in the previous mode and passed via a $\texttt{mem}(x)$ block ♣. Instead of using the unsafe operator $\texttt{mem}(x)$, it would be better to refer to the left limit of x, writing it again as $\texttt{last}\, x$. The unrestricted use of this operation may, however, introduce a new kind of causality loop which has to be statically rejected. Consider the following equation activated on a continuous time base:

```
y = -1.0 * (last y) and init y = 1.0
```

which defines, for all $n \in {}^\star\mathbb{N}$, the sequence ${}^\star y(n)$ such that:

$$ {}^\star y(n) = - {}^\star y(n-1) \quad {}^\star y(0) = 1 $$

Indeed, this differs little from the equation `y = -1.0 * y`. ${}^\star y(n)$ can be computed sequentially from ${}^\star y(n-1)$ but its value does not increase infinitesimally at each step, that is, y is not left continuous even though no zero-crossing occurs. For any time $t \in \mathbb{R}$, the set $\{n\partial \mid n \in {}^\star\mathbb{N} \wedge n\partial \approx t \wedge {}^\star y(n) \not\approx {}^\star y(n+1)\}$ is infinite. Thus, the value of $y(t)$ at any standard instant $t \in \mathbb{R}$ is undefined. This is an example of *chattering zeno* [2] and would cause the simulation to loop infinitely. In the analysis presented in the paper, we have chosen to statically reject this program. More than simply require that all computations be statically schedulable, we impose that every state jump and discontinuity must be aligned on a discrete event.

[12] In contrast, the application of a unit delay $\frac{1}{z}$ to a continuous-time signal is statically detected and results in a warning.

[13] The SIMULINK manual (http://www.mathworks.com/help/simulink/slref/memory.html) states, "Avoid using the Memory block when both these conditions are true: - Your model uses the variable-step solver ode15s or ode113. - The input to the block changes during simulation."

Contribution and organization of the paper: This paper presents the causality problem for a core language that combines LUSTRE-like stream equations, ODEs with reset and a basic control structure. The operator $\texttt{last}\, x$ stands for the previous value of x in non-standard semantics and coincides with its left-limit when x is left-continuous. This operation plays the role of a delay but is safer than the memory block $\texttt{mem}(x)$ as its semantics does not depend on when a particular solver decides to stop. When x is a continuous-state variable, it coincides with the so-called SIMULINK *state port*. We develop a non-standard semantics following [10] and a compile-time *causality analysis* in order to detect possible instantaneous loops. The static analysis takes the form of a type system, reminiscent of the simple Hindley-Milner type system for core ML [14]. A type signature for a function expresses the instantaneous dependencies between its inputs and outputs. We prove that well typed programs only progress by infinitely small steps outside of zero-crossing events, that is, signals are continuous during integration provided imported operations applied point-wise are continuous. We are not aware of such a correctness theorem based on a static typing discipline for hybrid modelers.

The presented material is implemented in ZÉLUS [15], a synchronous LUSTRE-like language extended with ODEs. Moreover, all examples in the paper are written in ZÉLUS.

The paper is organized as follows. Section 2 introduces a core synchronous language with ODEs. Section 3 presents its semantics based on non-standard analysis. Section 4 presents a type system for causality and Section 5 a major property: any well-typed program is proved not to have any discontinuities during integration. Section 6 discusses related work and we conclude in Section 7.

2. A Synchronous Lustre-like Language with ODEs

We now introduce a minimal kernel language with data-flow equations, ordinary differential equations and some simple control structures. These are the core elements of richer hybrid modeling languages, like, for instance, the Zélus programming language [15], which contains additional control structures needed for practical programming, including, notably, hierarchical automata. Working with a minimal language allows us to precisely define a semantic model in Section 3 and a causality analysis in Section 4. Its syntax is:

$$
\begin{aligned}
d &::= \texttt{let}\, x = e \mid \texttt{let}\, k\, f(p) = e\, \texttt{where}\, E \mid d; d \\
e &::= x \mid v \mid op(e) \mid e\, \texttt{fby}\, e \mid \texttt{last}\, x \mid f(e) \mid (e, e) \mid \texttt{up}(e) \\
p &::= (p, p) \mid x \\
E &::= () \mid x = e \mid \texttt{init}\, x = e \mid \texttt{der}\, x = e \\
&\quad \mid E\ \texttt{and}\ E \mid \texttt{local}\, x\, \texttt{in}\, E \mid \texttt{if}\, e\, \texttt{then}\, E\, \texttt{else}\, E \\
&\quad \mid \texttt{present}\, e\, \texttt{then}\, E\, \texttt{else}\, E \\
k &::= \texttt{D} \mid \texttt{C} \mid \texttt{A}
\end{aligned}
$$

A program is a sequence of definitions (d), that either bind the value of expression e to x ($\texttt{let}\, x = e$) or define a function ($\texttt{let}\, k\, f(p) = e\, \texttt{where}\, E$). In a function definition, k is the kind of the function f, p denotes formal parameters, and the result is the value of an expression e which may contain variables defined in the auxiliary equations E. There are three kinds of function: $k = \texttt{A}$ means that f is a *combinational* function (typically a function imported from the host language, for example, addition); $k = \texttt{D}$ means that f is a *sequential* function that must be activated at discrete instants (typically a LUSTRE function with an internal discrete state); $k = \texttt{C}$ denotes a hybrid function that may contain ODEs and which must be activated continuously.

An expression e can be a variable (x), an immediate value (v), for example, a boolean, integer or floating-point value, the point-wise application of an imported function ($op(e)$) such as $+$, $*$ or $\texttt{not}(\cdot)$, an initialized unit delay ($e_1\ \texttt{fby}\ e_2$), the left-limit of a signal ($\texttt{last}\, x$), a function application ($f(e)$), a pair ((e, e)) or a rising zero-crossing detection ($\texttt{up}(e)$), which, in this language kernel, is the only basic construct to produce an event from a continuous-time signal (e). A pattern p is a tree structure of identifiers (x). A set of equations E is either an empty equation ($()$); an equality stating that a pattern equals the value of an expression at every instant ($x = e$); the initialization of a state variable x with a value e ($\texttt{init}\ x = e$); or, the current value of the derivative of x ($\texttt{der}\, x = e$). An equation can also be the conjunction of two sets of equations ($E_1\ \texttt{and}\ E_2$); the declaration that a variable x is defined within, and local to, a set of equations ($\texttt{local}\ x\ \texttt{in}\ E$); a conditional that activates a branch according to the value of a boolean expression ($\texttt{if}\, e\, \texttt{then}\, E_1\, \texttt{else}\, E_2$), and a variant that operates on an event expression ($\texttt{present}\ e\ \texttt{then}\ E_1\ \texttt{else}\ E_2$).

Notational abbreviations:

(*a*) $\texttt{der}\, x = e\ \texttt{init}\ e_0 \stackrel{def}{=} \texttt{init}\ x = e_0\ \texttt{and}\ \texttt{der}\, x = e$

(*b*) $\texttt{der}\, x = e\ \texttt{init}\ e_0\ \texttt{reset}\ z \to e_1 \stackrel{def}{=}$
$\texttt{init}\ x = e_0\ \texttt{and}\ \texttt{present}\ z\ \texttt{then}\ x = e_1\ \texttt{else}\ \texttt{der}\, x = e$

$\texttt{init}\ x = e_0$ initialize the state variable $\texttt{last}\, x$ with the first value of e_0. When z is true, the current value of x is that of e_1, otherwise its current derivative is the current value of e. Equations must be in Static Single Assignment (SSA) form: every variable has a unique definition at every instant, e.g., the two equations $x = e_1\ \texttt{and}\ x = e_2$ are not valid.

2.1. Examples

This small language kernel is a subset of ZÉLUS and allows for writting synchronous LUSTRE-like programs. To start with, we give a few examples in the concrete syntax of ZÉLUS. [14]

[14] Tutorial and reference manual at: http://zelus.di.ens.fr/man/.

A combinatorial function with two inputs x and y which computes at every instant $n \in \mathbb{N}$, the minimum and maximum of the n-th values $x(n)$ and $y(n)$ is written: [15]

```
let fun min_max(x, y) = min(x, y), max(x, y)
```

The keyword `fun` is the concrete syntax for kind $k = \texttt{A}$. A function that counts the number of instants where an input signal *click* is true and resets the counter when *res* is true is written:

```
let node count_with_reset(x) = cpt where
  rec if x then do v = 1.0 done else do v = 0.0 done
  and cpt = (0.0 fby cpt) + v
```

The keyword `node` is the concrete syntax for kind $k = \texttt{D}$. It means that the function is *stateful*, i.e., its output at instant n may depend on previous inputs and computed values. The keyword `rec` means that the set of equations, separated by `and`s, are mutually recursive. 0 `fby` *cpt* is the unit delay initialized with 0. It is such that $(0$ `fby` $cpt)(0) = 0$ and for $n \geq 1, (0$ `fby` $cpt)(n) = cpt(n-1)$. The `do/done` plays the role of parenthesis. An other typical example of a node is the explicit forward Euler integration. Given x' and xi, it computes x such that $x(0) = xi(0)$ and $\forall n \geq 1. x(n) = x(n-1) + x'(n-1) * step(n-1)$. The node `heat` computes an approximation of $\dot{temp} = gain_1 - gain_2 * temp$ with *temp* initialized to $temp_0(0)$. [16]

```
let node integr(xi, x') = x where
  rec x = xi fby (x + x' * step)

let node heat(temp0, gain1, gain2) = temp where
  rec temp = integr(temp0, gain1 - gain2 * temp)
```

A continuous-time model for the heater is written:

```
let hybrid heat(temp0, gain1, gain2) = temp where
  rec der temp = gain1 - gain2 * temp init temp0
```

The keyword `hybrid` is the concrete syntax for kind $k = \texttt{C}$. It means that the function is continuous time, which will be interpreted as a function taking inputs and output signals defined on a non standard base clock.

3. Non-standard Semantics

We now define the semantics of the kernel language introduced in the previous section. These details define the real object of study and underlie the

[15] In Zélus, the keyword `fun` is optional.

[16] In all the remaining examples, we write `+/-/*` for operations on floating point numbers. In Zélus, those operators are normally written `+./-./*.`.

causality analysis introduced in the next section, and the properties demonstrated in Section 5. The semantics of a discrete synchronous language—that is, one with data-flow equations and control structures but not ODEs—can be modeled by infinite sequences of values synchronized on a 'base clock' indexed by $\mathbb{N}$. The basic idea [10] applied in this section is to introduce an infinitesimal base clock indexed by ${}^\star\mathbb{N}$, the non-standard extension of $\mathbb{N}$. The standard definitions of discrete operators, like the unit delay `fby`, are lifted to the richer context and it becomes possible to define differential equations, zero-crossings, and the left-limits of signals.

3.1. Semantics

Let ${}^\star\mathbb{R}$ and ${}^\star\mathbb{N}$ be the non-standard extensions of $\mathbb{R}$ and $\mathbb{N}$. ${}^\star\mathbb{N}$ is totally ordered and every set bounded from above (respectively below) has a unique maximal (respectively minimal) element. Let $\partial \in {}^\star\mathbb{R}$ be an arbitrary but fixed infinitesimal value, that is, $\partial > 0$ and $\partial \approx 0$. We show later (Invariant 2 in Section 3.3) that the precise choice of infinitesimal is unimportant. Let the global time base or *base clock* be the infinite set of instants:

$$\mathbb{T}_\partial = \{t_n = n\partial \mid n \in {}^\star\mathbb{N}\}$$

$\mathbb{T}_\partial$ inherits a total order from ${}^\star\mathbb{N}$; in addition, for each element of $\mathbb{R}_+$ there exists an infinitesimally close element of $\mathbb{T}_\partial$. Whenever possible we leave ∂ implicit and write $\mathbb{T}$ instead of $\mathbb{T}_\partial$. Let $T = \{t'_n \mid n \in {}^\star\mathbb{N}\} \subseteq \mathbb{T}$. $T(i)$ stands for t'_i, the i-th element of T. In the sequel, we only consider subsets of the time base $\mathbb{T}$ obtained by sampling a time base on a boolean condition or a zero-crossing event. Any element of a time base is thus of the form $k\partial$ where $k \in {}^\star\mathbb{N}$. If $T \subseteq \mathbb{T}$, we write ${}^\bullet T(t)$ for the immediate predecessor of t in T and $T^\bullet(t)$ for the immediate successor of t in T. For an instant t, we write its immediate predecessor and successor as, respectively, ${}^\bullet t$ and $t^\bullet$, rather than as ${}^\bullet\mathbb{T}(t)$ and $\mathbb{T}^\bullet(t)$. For $t \in T \subseteq \mathbb{T}$, neither ${}^\bullet t$ nor $t^\bullet$ necessarily belong to T. $min(T)$ is the minimal element of T and $t \leq_T t'$ means that t is a predecessor of t' in T.

Definition 1 (Signals). *Let $V_\perp = V + \{\perp\}$ where V is a set. The set of signals, Signals(V), is the set of functions from $\mathbb{T}$ to $V_\perp$, that is $\mathbb{T} \mapsto V_\perp$. A signal $x : T \mapsto V_\perp$ is a total function from a time base $T \subseteq \mathbb{T}$ to $V_\perp$. Moreover, for all $t \notin T, x(t) = \perp$. If T is a time base, $x(T(n))$ and $x(t_n)$ are the value of x at instant t_n where $n \in {}^\star\mathbb{N}$ is the n-th element of T. The clock of a signal x is clock$(x) = \{t \in \mathbb{T} \mid x(t) \neq \perp\}$.*

Sampling: Let $\texttt{bool} = \{\texttt{false}, \texttt{true}\}$ and $x : T \mapsto \texttt{bool}_\perp$. The *sampling of T according to x*, written T `on` x, is the subset of instants defined by:

$$T \texttt{ on } x = \{t \mid t \in T \wedge x(t) = \texttt{true}\}$$

Note that as T `on` $x \subseteq T$, it is also totally ordered.

The zero-crossing of $x : T \mapsto {}^\star\mathbb{R}_\perp$ is $up(x) : T \mapsto \texttt{bool}_\perp$. To emphasize that $up(x)$ is defined only for $t \in T$, we write its value at time t as $up(x)(T)(t)$. For $t \notin T$, $up(x)(T)(t) = \perp$. In the definition below $<$ is the total order on ${}^\star\mathbb{R}$.

$$\begin{array}{rcll} up(x)(T)(t_0) & = & \texttt{false} \text{ where } t_0 = min(T) \text{ and, otherwise,} & \\ up(x)(T)(t) & = & \texttt{true} \text{ if } \exists n \in \mathbb{N}, n{\geq}1. & \wedge (x(t{-}n\partial) < 0) \\ & & & \wedge (x(t{-}(n{-}1)\partial) = 0) \\ & & & \wedge \cdots \wedge\ (x(t{-}\partial) = 0) \\ & & & \wedge (x(t) > 0) \\ up(x)(T)(t) & = & \texttt{false} \text{ otherwise} & \end{array} \tag{1}$$

The above definition means that a zero-crossing on x occurs when x goes from a strictly negative to a strictly positive value, possibly with finitely many intermediate values equal to 0. With this definition, the output of $up(x)(T)(t)$ depends instantaneously on $x(t)$. An alternative definition delays the effect of a zero-crossing by one instant so that $up(x)(T)(t)$ does not depend instantaneously on $x(t)$:

$$\begin{array}{rcll} up(x)(T)(t) & = & \texttt{true} \text{ if } \exists n \in \mathbb{N}, n{\geq}2. & \wedge (x(t{-}n\partial) < 0) \\ & & & \wedge (x(t{-}(n{-}1)\partial) = 0) \\ & & & \wedge \cdots \wedge\ (x(t{-}2\partial) = 0) \\ & & & \wedge (x(t{-}\partial)) > 0) \end{array} \tag{2}$$

In Zélus [15], the expression $\texttt{up}(e)$ detects a *zero-crossing event* and is given the special type `zero`. Only a dedicated set of primitives produce a value of that this type. They are used at simulation time to stop simulation [16]. In the statement `present` y `then` E_1 `else` E_2, y must be of type `zero`; E_1 being executed when the event occurs; E_2 being executed otherwise. Hence, `present` $\texttt{up}(e)$ `then` E_1 `else` E_2 executes equations in E_1 at the instant where e crosses zero. In `if` y `then` E_1 `else` E_2, y must be a boolean: E_1 is activated when y is true; E_2 when y is false. Moreover, the language adopts the following interpretation for $\texttt{up}(\cdot)$ which differs slightly from definitions (1) and (2): an $\texttt{up}(x)$ only detects zero-crossings that occur during integration. In non-standard semantics, this means that $x(t{-}n\partial) \approx x(t)$. Also, the effect of $\texttt{up}(x)$ is delayed, following the interpretation of (2). Another primitive $\texttt{disc}(x)$ is in charge of detecting when a discrete change on x occurs, with $x(t - \partial) \not\approx x(t)$. The result at instant t depends instantaneously on $x(t)$.

Environments: Environments are functions associating names to values. They are defined and used for the semantics of equations and expressions.

Let V be a set of values closed under product and sum, ${}^\star V$ be its non-standard extension such that ${}^\star(V_1 \times V_2) = {}^\star V_1 \times {}^\star V_2$. ${}^\star V = V$ when V is finite. We define ${}^\star V_\perp = {}^\star V + \{\perp\}$. Let $L = \{x_1, \texttt{last}\, x_1, ..., x_n, \texttt{last}\, x_n, ...\}$ be a set of variable names; $\texttt{last}\, x$ denotes a name that is distinct from x. $L_g = \{f_1, ..., f_n, ...\}$ stands for the set of identifiers for functions. A local environment

ρ and a global environment G map names, respectively, to signals and signal functions:

$$\rho : L \mapsto \mathit{Signals}(^\star V) \qquad G : L_g \mapsto (\mathit{Signals}(^\star V) \mapsto \mathit{Signals}(^\star V))$$

If ρ is an environment, $\rho(x)$ returns its value in ρ if $x \in \mathit{Dom}(\rho)$, and $\bot$ otherwise. If ρ is an environment, $\rho + [s/x]$ is its extension such that $(\rho + [s/x])(y) = s$ if $x = y$ and $\rho(y)$ otherwise. $(\rho + [s/\texttt{last}\, x])(\texttt{last}\, x) = s$ and $(\rho + [s/\texttt{last}\, y])(\texttt{last}\, x) = \rho(\texttt{last}\, x)$ if $x \neq y$, and $\bot$ otherwise.

Given two local environments ρ_1 and ρ_2, we define their *(exclusive) composition* that is commutative and associative $(\rho_1 + \rho_2)(x)$ as $\rho_1(x)$ if $x \notin \mathit{Dom}(\rho_2)$, $\rho_2(x)$ if $x \notin \mathit{Dom}(\rho_1)$, and $\bot$ otherwise. The composition of environments is used for defining the semantics of two parallel equations. If an equation $x = e_1$ defines the environment $[s_1/x]$ and equation $y = e_2$ the environment $[s_2/y]$, then $x = e_1$ `and` $y = e_2$ defines $[s_1/x] + [s_2/y]$.

The merge of two environments according to a signal $s \in \mathit{Signals}(\texttt{bool})$, written $\rho = \mathit{merge}\,(T)\,(s)\,(\rho_1)\,(\rho_2)$, is defined by:

$$\rho(x)(t) = \begin{cases} \rho_1(x)(t) & \text{if } s(t) = \texttt{true} \text{ and } x \in \mathit{Dom}(\rho_1) \\ \rho(\texttt{last}\, x)(t) & \text{if } s(t) = \texttt{true} \text{ and } x \in \mathit{Dom}(\rho_2) \setminus \mathit{Dom}(\rho_1) \\ \rho_2(x)(t) & \text{if } s(t) = \texttt{false} \text{ and } x \in \mathit{Dom}(\rho_2) \\ \rho(\texttt{last}\, x)(t) & \text{if } s(t) = \texttt{false} \text{ and } x \in \mathit{Dom}(\rho_1) \setminus \mathit{Dom}(\rho_2) \\ \bot & \text{otherwise.} \end{cases}$$

The second and fourth cases states that signals implicitly maintain their values when not explicitly defined in a branch. If a variable x is defined in ρ_1 but not in ρ_2, we implicitly add the equation $x = \texttt{last}\, x$ to the latter branch. For example, consider $\rho' = \rho + \mathit{merge}\,(T)\,(s)\,([s_1/x])\,([s_2/y])$ with $s(t) = \texttt{false}$. Then, $\rho'(x)(t) = \rho(\texttt{last}\, x)(t)$. The merge of two environments is used for defining the semantics of $\texttt{if}\, z\, \texttt{then}\, x = e_1\, \texttt{else}\, y = e_2$. When z is true, no equation for x is given; when z is false, no equation for y is given. We consider that when no equation is given for a variable, its implicitly keep its previous value, that is, the behavior is that of $\texttt{if}\, z\, \texttt{then}\, x = e_1$ `and` $y = \texttt{last}\, y\, \texttt{else}\, y = e_2$ `and` $x = \texttt{last}\, x$. Said differently, x is constant between two instants where z is true. $\texttt{last}\, x$ always contain the *last computed value of signal* x. The implicit completion with *last* is followed by SCADE 6 for all control structures, including activate conditions and hierarchical automata and ZÉLUS.

Expressions: The value of an expression is a signals (or tuple of signals) whereas node definitions define functions from signals to signals. For an expression e, ${}^\star[\![e]\!]^\rho_G(T)(t)$ defines its semantics. It defines at instant $t \in T$ both the value of e and a Boolean value, true if e raises a zero-crossing event, false otherwise. The definition is given in Figure 4 and explained below.

Considering each clause from the top: the value of expression e is considered undefined outside of T. The current value of an immediate constant v is v and no zero-crossing event is raised. The current value of x is the one stored in the

$$
\begin{array}{lcl}
{}^{\star}[\![e]\!]^{\rho}_{G}(T)(t) & = & \bot, \bot \text{ if } t \notin T, \text{ and otherwise:} \\
{}^{\star}[\![v]\!]^{\rho}_{G}(T)(t) & = & v, \texttt{false} \\
{}^{\star}[\![x]\!]^{\rho}_{G}(T)(t) & = & \rho(x)(t), \texttt{false} \\
{}^{\star}[\![\texttt{last}\, x]\!]^{\rho}_{G}(T)(t) & = & \rho(\texttt{last}\, x)(t), \texttt{false} \text{ if } \texttt{last}\, x \in \mathit{Dom}(\rho) \\
{}^{\star}[\![\texttt{last}\, x]\!]^{\rho}_{G}(T)(t) & = & \rho(x)({}^{\bullet}\mathit{clock}(x)(t)), \texttt{false} \text{ otherwise} \\
{}^{\star}[\![\mathit{op}(e)]\!]^{\rho}_{G}(T)(t) & = & \text{let } v, z = {}^{\star}[\![e]\!]^{\rho}_{G}(T)(t) \text{ in } \mathit{op}(v), z \\
{}^{\star}[\![(e_1, e_2)]\!]^{\rho}_{G}(T)(t) & = & \text{let } v_1, z_1 = {}^{\star}[\![e_1]\!]^{\rho}_{G}(T)(t) \text{ in} \\
 & & \text{let } v_2, z_2 = {}^{\star}[\![e_2]\!]^{\rho}_{G}(T)(t) \text{ in } (v_1, v_2), (z_1 \vee z_2) \\
{}^{\star}[\![e_1\ \texttt{fby}\ e_2]\!]^{\rho}_{G}(T)(t_0) & = & {}^{\star}[\![e_1]\!]^{\rho}_{G}(T)(t_0) \text{ if } t_0 = \mathit{min}(T) \\
{}^{\star}[\![e_1\ \texttt{fby}\ e_2]\!]^{\rho}_{G}(T)(t) & = & {}^{\star}[\![e_2]\!]^{\rho}_{G}(T)({}^{\bullet}T(t)) \text{ otherwise} \\
{}^{\star}[\![f(e)]\!]^{\rho}_{G}(T)(t) & = & \text{let } s(t'), z(t') = {}^{\star}[\![e]\!]^{\rho}_{G}(T)(t') \text{ for all } t' \leq_T t \text{ in} \\
 & & \text{let } v', z' = G(f)(s)(t) \text{ in } v', z(t) \vee z' \\
{}^{\star}[\![\texttt{up}(e)]\!]^{\rho}_{G}(T)(t) & = & \text{let } s(t'), z(t') = {}^{\star}[\![e]\!]^{\rho}_{G}(T)(t') \text{ for all } t' \leq_T t \text{ in} \\
 & & \text{let } v' = \mathit{up}(s)(T)(t) \text{ in } v', z(t) \vee v'
\end{array}
$$

Figure 4: The non-standard semantics of expressions

environment $\rho(x)$ and no event is raised. The value of $\texttt{last}\, x$ is either the value associated to the entry $\texttt{last}\, x$ in ρ, if it exists, or the last computed value of x. Remind that $\mathit{clock}(x)$ define the sequence of instants where x is defined. Then, $\rho(x)({}^{\bullet}\mathit{clock}(x)(t))$ is the previous value of x, on the clock where x is defined. The semantics of $\mathit{op}(e)$ is obtained by applying the operation op to e at every instant, an event is raised only if e raises one. An expression (e_1, e_2) returns a pair at every instant and raises an event if either e_1 or e_2 does. The initial value of a delay $e_1\ \texttt{fby}\ e_2$ is that of e_1. Afterward, it is the previous value of e_2 according to clock T. For example, the value of $0\ \texttt{fby}\ x$ on clock T is the value x had at the previous instant that T was active. This is not necessarily the previous value of x. On the contrary, $\texttt{last}\, x$ is the previous value of x the last time x was defined.

The detailed definitions of time bases allow us to precisely express this subtle but important difference: ${}^{\bullet}T(t)$ is the instant that precedes t on the clock T, which, due to an `if` or a `present`, may not be the previous instant of the base clock; ${}^{\bullet}\mathit{clock}(x)(t)$ is the instant that precedes t on the clock of variable x, that is, the last instant when x was defined. Furthermore, the use of an infinitesimal time base permits a definition of $\texttt{last}\, x$ that is valid in both discrete and continuous contexts. The semantics of $f(e)$ is the application of the function f to the signal value of e, which raises an event at an instant when either e or the body of f does. Note that f maps a complete input stream s (defined by quantifica-

$$
\begin{aligned}
&{}^{\star}[\![x = e]\!]^{\rho}_{G}(T) = [s/x], z && \textit{where } \forall t \in T,\ s(t), z(t) = {}^{\star}[\![e]\!]^{\rho}_{G}(T)(t)\\
&{}^{\star}[\![E_1 \texttt{ and } E_2]\!]^{\rho}_{G}(T) = \rho_1 + \rho_2, z_1 \textit{ or } z_2 && \textit{where } \rho_1, z_1 = {}^{\star}[\![E_1]\!]^{\rho}_{G}(T)\\
& && \quad \textit{and } \rho_2, z_2 = {}^{\star}[\![E_2]\!]^{\rho}_{G}(T)\\
&{}^{\star}[\![\texttt{present } e \texttt{ then } E_1 \texttt{ else } E_2]\!]^{\rho}_{G}(T) = \rho', z \textit{ or } z_1 \textit{ or } z_2 && \\
& && \textit{where } \forall t \in T,\ s(t), z(t) = {}^{\star}[\![e]\!]^{\rho}_{G}(T)(t)\\
& && \quad \textit{and } \rho_1, z_1 = {}^{\star}[\![E_1]\!]^{\rho}_{G}(T \texttt{ on } s)\\
& && \quad \textit{and } \rho_2, z_2 = {}^{\star}[\![E_2]\!]^{\rho}_{G}(T \texttt{ on } \mathit{not}(s))\\
& && \quad \textit{and } \rho' = \mathit{merge}\,(T)\,(s)\,(\rho_1)\,(\rho_2)\\
&{}^{\star}[\![\texttt{if } e \texttt{ then } E_1 \texttt{ else } E_2]\!]^{\rho}_{G}(T) = \rho', z \textit{ or } z_1 \textit{ or } z_2 \text{ defined as above.} && \\
&{}^{\star}[\![\texttt{init } x = e]\!]^{\rho}_{G}(T) = [s/\texttt{last } x], z && \textit{where } s(t_0), z(t_0) = {}^{\star}[\![e]\!]^{\rho}_{G}(T)(t_0)\\
& && \quad \textit{and } t_0 = \mathit{min}(T)\\
& && \quad \textit{and } \forall t \neq t_0,\ s(t) = \rho(x)({}^{\bullet}T(t))\\
& && \quad \textit{and } z(t) = \texttt{false}\\
&{}^{\star}[\![\texttt{der } x = e]\!]^{\rho}_{G}(T) = [s/x], z && \textit{where } \forall t \in T, s'(t), z(t) = {}^{\star}[\![e]\!]^{\rho}_{G}(T)(t)\\
& && \quad \textit{and } s(t) = s({}^{\bullet}t) + \partial \times s'({}^{\bullet}t)
\end{aligned}
$$

Figure 5: The non-standard semantics of equations

tion over t') to an output stream from which the value at instant t is extracted. Typically for a LUSTRE-like language, the output stream inherits the clock T of the input stream. Finally, the semantics of $\texttt{up}(e)$ is given by the operator $up(\cdot)$, which raises a zero-crossing event when either e does or $up(s)(T)(t)$ is true.

Equations: If E is an equation, G is a global environment, ρ is a local environment and T is a time base, ${}^{\star}[\![E]\!]^{\rho}_{G}(T) = \rho', z$ means that the evaluation of E on the time base T returns a local environment ρ' and a zero-crossing signal z. As for expressions, the value of E is undefined outside of T, that is, for all $t \notin T$, $\rho'(x)(t) = \bot$ and $z(t) = \bot$. For all $t \in T$, $z(t) = \texttt{true}$ signals that a zero-crossing occurs at instant t and $z(t) = \texttt{false}$ means that no zero-crossing occurs at that instant. The semantics of equations is given in Figure 5, where the following notation is used: if $z_1 : T \mapsto \texttt{bool}_{\bot}$ and $z_2 : T \mapsto \texttt{bool}_{\bot}$ then $z_1 \textit{ or } z_2 : T \mapsto \texttt{bool}_{\bot}$ and $\forall t \in T,\ (z_1 \textit{ or } z_2)(t) = z_1(t) \vee z_2(t)$ if $z_1(t) \neq \bot$ and $z_2(t) \neq \bot$, and otherwise, $(z_1 \textit{ or } z_2)(t) = \bot$.

Considering each clause from the top: a (basic) equation yields a singleton environment, where x is associated with the stream s defined by the expression e, and a boolean event stream. Combining equations amounts to combining their (disjoint) environments, which themselves are inductively defined. A `present` statement partitions a clock T into two sets according to the value of a signal s:

T `on` s and T `on` $not(s)$. The equations in a branch are inductively defined relative to one clock or the other and the resulting environments are then merged onto the original clock. An initialization equation `init` $x = e$ defines an environment $[s/\texttt{last}\, x]$ such that the first value of s (at time $t_0 = min(T)$) is the initial value of e. Then, the current value of s is the previous value of x. This definition works both for a continuous state variable — whose value is defined by an equation `der` $x = e$ — or a discrete state variable — whose value is defined by an equation $x = e$. For example, the following equations activated on a set of instants $T = \{t_0, t_1, ...\}$ define a counter x which is incremented by one at every step. The associated environment is $[s/x, s'/\texttt{last}\, x]$ with $s'(t_0) = 0$, $s'(t_{i+1}) = s(t_i)$, and $s(t_i) = s'(t_i) + 1$.

```
init x = 0.0 and x = last x + 1.0
```

Finally, `der` $x = e$ defines the derivative of x using an explicit Euler scheme [10].

Function definitions: Function definition is our final concern: we must show the existence of fixed points in the sense of Kahn process network semantics based on Scott domains.

The prefix order on signals $Signals(V)$ indexed by $\mathbb{T}$ is defined as: signal x is a *prefix* of signal y, written $x \leq_{Signals(V)} y$, if $x(t) \neq y(t)$ implies $x(t') = \bot$ for all t' such that $t \leq t'$. This order models that a signal must be computed from left to right. As soon as there exist an instant t where $x(t)$ and $y(t)$ differs, i.e, $x(t) = \bot$ and $y(t) \neq \bot$ (or the contrary), then $x(t)$ is undefined for the remaining instants. The minimum element is the undefined signal $\bot_{Signals(V)}$ for which $\forall t \in \mathbb{T}$, $\bot_{Signals(V)}(t) = \bot$. When possible, we write $\bot$ for $\bot_{Signals(V)}$ and $x \leq y$ for $x \leq_{Signals(V)} y$. The symbol $\bigvee$ denotes a supremum in the prefix order. The set $(Signals(V), \leq_{Signals(V)}, \bot)$ is a complete partial order. A function $f : Signals(^\star V) \mapsto Signals(^\star V)$ is continuous if $\bigvee_i f(x_i) = f(\bigvee_i x_i)$ for every increasing chain of signals, where increasing refers to the prefix order. If f is continuous, then equation $x = f(x)$ has a least solution denoted by $fix(f)$, and equal to $\bigvee_i f^i(\bot)$ (Kleene fix-point theorem). We name such continuity on the prefix order *Kahn continuity* [17]. Figure 6 illustrates this principle thanks to a simple program of one variable x, which semantics defines a signal $^\star x$ as the least fixed point of a continuous operator. The iteration of the operator computes the signal step by step, advancing time by ∂ at each iteration.

The prefix order is lifted to environments so that $\rho \leq \rho'$ iff for all $x \in Dom(\rho) \cup Dom(\rho')$, $\rho(x) \leq \rho'(x)$, and to pairs such that $(x, y) \leq (x', y')$ iff $x \leq x'$ and $y \leq y'$.

Property 1 (Kahn continuity). *Let $[s/p]$ be an environment, G a global environment of Kahn-continuous functions and T a clock. The function:*

$$F : (L \mapsto Signals(^\star V)) \times Signals(\texttt{bool}) \mapsto (L \mapsto Signals(^\star V)) \times Signals(\texttt{bool})$$

such that:

$$F(\rho, z) = \rho', (z \text{ or } z') \quad \text{where} \quad \rho', z' = {}^\star[\![E]\!]_G^{\rho+[s/p]}(T)$$

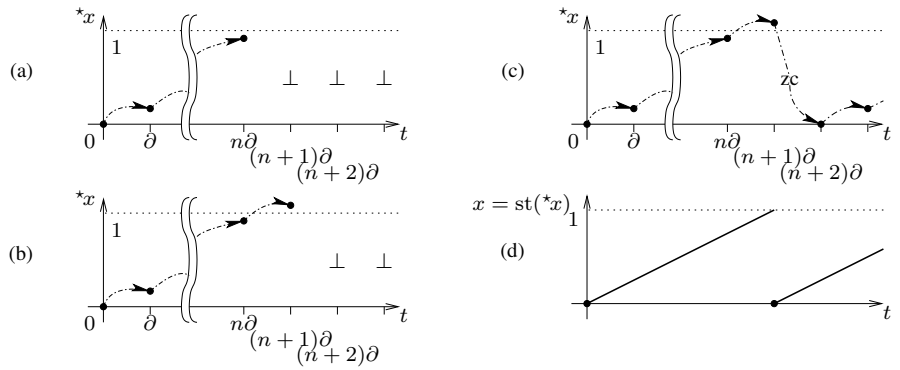


Figure 6: Semantics of program `der x = 1.0 init 0.0 reset up(x - 1.0) -> 0.0`; (a) Signal $^\star x$ is defined up to date $t = n\partial$, it takes value $\bot$ after; dashed arrows represent the causal dependencies between the last value of $^\star x$ and its current value; (b) using the definition of the semantics of equation `der x = ...` (last two lines of Figure 5), the signal is extended by one infinitesimal time-step, $^\star x((n+1)\partial) = {}^\star x(n\partial) + \partial$; (c) the least fixed point is a signal with no $\bot$; remark the occurrence of a zero-crossing that triggers the reset of the signal, as defined in Figure 5, lines 4–8; (d) standardization (see Section 3.2) of the signal yields a sawtooth function.

is Kahn continuous, that is, for any sequence $(\rho_i, z_i)_{i\geq 0}$*:*

$$F(\bigvee_{i\in I}(\rho_i, z_i)) = \bigvee_{i\in I}(F(\rho_i, z_i))$$

ρ (and thus ρ_i) denotes an environment, that is, a mapping from a set of names to signals. A set of names is of the form $L = \{x_1, \texttt{last}\, x_1, ...x_n, \texttt{last}\, x_n\}$. z (and thus z_i) denotes a boolean which is true when a zero-crossing occurs, false otherwise.

Proof: We only provide a sketch. We first need to prove the result for expressions listed in Figure 4. We only review the expressions involving the non-standard semantics in a nontrivial way, as the other cases are routine. Consider e_1 `fby` e_2 and `last` x. None of these expressions contributes to the second (zero-crossing) field of the semantics, so only the first field matters. In fact, the Kahn continuity of e_1 `fby` e_2 is proved exactly as for Lustre [18], since only the total ordering of the underlying time index matters and the argument lifts without change from $\mathbb{N}$ to $\mathbb{T}$. The same holds for `last` x, which corresponds to `pre`(x) in Lustre in the lifting from $\mathbb{N}$ to $\mathbb{T}$. The expression `up`(e) contributes to the second field of the semantics. Formula (1) defining $up(x)$ is causal and thereby Kahn continuous. We then consider the equations of Figure 5. We discuss only `der` $x = e$ since the other cases pass as in Lustre (including the composition of equations E_1 `and` E_2). Consider the first field of the semantics. If e returns the value $s'(t)$ at instant t, the value of x at instant t is $s(^\bullet(t)) + \partial \times s'(^\bullet t)$. As the sum of two Kahn continuous signals is continuous, x is also Kahn continuous. □

As a consequence, an equation $(\rho, z) = F(\rho, z)$ admits a least fixed point $fix(F) = \bigvee_i(F^i(\bot, \bot))$.

The declaration of ${}^\star[\![\texttt{let}\, k\, f(p) = e\, \texttt{where}\, E]\!]_G(T)$ defines a Kahn-continuous function ${}^\star f$ such that

$$ {}^\star[\![\texttt{let}\, k\, f(p) = e\, \texttt{where}\, E]\!]_G(T)(s)(t) = {}^\star f(T)(s)(t) $$

where

$$ \begin{aligned} {}^\star f(T)(s)(t) = \text{let } s'(t'), z(t') = {}^\star[\![e]\!]_G^{\rho'+[s/p]}(T)(t') \text{ in} \\ s'(t), z(t) \vee z'(t) \end{aligned} $$

and with

$$ (\rho', z') = \mathit{fix}((\rho, z) \mapsto {}^\star[\![E]\!]_G^{\rho+[s/p]}(T)) $$

Kahn-continuity of ${}^\star f$ does not mean that the function computes anything interesting. In particular, the semantics gives a meaning to functions that become 'stuck', like[17]

```
let hybrid f(x) = y where rec y = y + x
```

The semantics of `f` is ${}^\star f(x) = \bot$ since the minimal solution of equation $y = y + x$ is $\bot$. The purpose of the causality analysis is to statically reject this kind of program.

3.2. Standardization

We now relate the non-standard semantics to the usual super-dense semantics of hybrid systems. Following [11], the execution of a hybrid system alternates between integration steps and discrete steps. Signals are now interpreted as total functions from the time index $\mathbb{S} = \mathbb{R} \times \mathbb{N}$ to $V_\bot$. This time index is called *super-dense time* [11, 2] and is ordered lexically, $(t, n) <_{\mathbb{S}} (t', n')$ iff $t <_{\mathbb{R}} t'$, or $t = t'$ and $n <_{\mathbb{N}} n'$. Moreover, for any (t, n) and (t, n') where $n \leq_{\mathbb{N}} n'$, if $x(t, n') \neq \bot$ then $x(t, n) \neq \bot$.

A *timeline* for a signal x is a function $N_x : \mathbb{R}_+ \mapsto \mathbb{N}_\bot$. $N_x(t)$ is the number of instants of x that occur at a real date t and such timelines thus specify a subset of super-dense time $\mathbb{S}_{N_x} = \{(t, n) \in \mathbb{S} \mid n \leq_{\mathbb{N}} N_x(t)\}$. In particular, if N_x is always 0, then $\mathbb{S}_{N_x}$ is isomorphic to $\mathbb{R}_+$. For $t \in \mathbb{R}$ and $T \subseteq \mathbb{T}$, define:

$$ \mathit{set}(T)(t) \stackrel{def}{=} \{t' \in T \mid t' \approx t\} \subseteq \mathbb{T} $$

that is, the set of all instants infinitely close to t. T is totally ordered and hence so is $\mathit{set}(T)(t)$. Let $x : T \mapsto {}^\star V_\bot$.

We now proceed to the definition of the *timeline* N_x of x and the *standardization* of x, written

$$ \mathit{st}(x) : \mathbb{R} \times \mathbb{N} \mapsto V_\bot, $$

[17] Remember that the keyword `hybrid` stands for $k = \texttt{C}$ and `node` for $k = \texttt{D}$.

such that $st(x)(t,n) = \bot$ for $n > N_x(t)$.

Let $T' \stackrel{def}{=} set(T)(t)$ and consider

$$st(x(T')) \stackrel{def}{=} \{st(x(t')) \mid t' \in T'\}.$$

(a) If $st(x(T')) = \{v\}$ then, at instant t, x's timeline is $N_x(t) = 0$ and its standardization is $st(x)(t,0) = v$.

(b) If $st(x(T'))$ is not a singleton set, then let

$$Z \stackrel{def}{=} \{t' \mid t' \in T' \wedge x(t') \not\approx x(T'^{\bullet}(t'))\}$$

that is, Z collects the instants at which x experiences a non-infinitesimal change. Z is either finite or infinite:

(i) If $Z = \{t_{z_0}, \ldots, t_{z_m}\}$ is finite, timeline $N_x(t) = m$ and the standard value of signal x at time t is:

$$\forall n \in \{0, \ldots, m\},\ st(x)(t,n) = st(x(t_{z_n}))$$

(ii) If Z is infinite (it may even lack a minimum element), let

$$N_x(t) = \bot \ \text{ and } \ \forall n,\ st(x)(t,n) = \bot$$

which corresponds to a Zeno behavior.

Our approach differs slightly from [2], where the value of a signal is frozen for $n > N(t)$. We decide instead to set it to the value $\bot$. Each approach has its merits. For ours, parts of signals that do not experience jumps are simply indexed by $(t,0)$ which we identify with t. In turn, we squander the undefined value $\bot$ which is usually devoted to Scott-Kahn semantics and causality issues.

3.3. Key properties

We now define two main properties that "reasonable" programs should satisfy. In a nutshell, the first one states that discontinuities do not occur outside of zero-crossing events, that is, signals are continuous during integration. The second one states that the semantics should not depend on the choice of the infinitesimal. These two invariants are sufficient conditions to ensure that a standardization exists. Proof that these properties are invariants of the semantics (Theorem 1) and the precise statement of the assumptions under which they hold (Assumptions 1 and 2) are detailed in Section 5. The *Nonsmooth Program*, in page 29, is a typical example of an "unreasonable" program. It violates both Assumption 1 and the first invariant, stated below.

Invariant 1 (Zero-crossings). *An expression e evaluated under G, ρ and a time base T has no discontinuities outside of zero-crossing events. Formally, we define $s(t), z(t) = {}^{\star}[\![e]\!]^{\rho}_{G}(T)(t)$, then $\forall t, t' \in T$ such that $t \leq t'$:*

$$t \approx t' \Rightarrow (\exists t'' \in T, t \leq t'' \leq t' \wedge z(t'')) \vee s(t) \approx s(t')$$

This invariant states that all discontinuities are aligned on zero-crossings, that is, signals must evolve continuously during integration. Discrete changes must be announced to the solver using $\texttt{up}(\cdot)$. Not all programs satisfy the invariant, for example,

```
let hybrid f()= y where rec y = last y + 1.0 and init y = 0.0
```

`f` takes a single argument `()` of type `unit` and returns a value `y`. Writing ${}^\star y(n)$ for the value of y at instant $n\partial$ with $n \in {}^\star\mathbb{N}$, we get ${}^\star y(0) = 0$ and ${}^\star y(n) = {}^\star y(n-1) + 1$. Yet, ${}^\star y(n) \not\approx {}^\star y(n-1)$ while no zero-crossing is registered for any instant $n \in {}^\star\mathbb{N}$. This program will be statically rejected using the type system presented in the next section.

Invariant 2 (Independence from ∂). *The semantics of e evaluated under G, ρ and a base time T is independent of the infinitesimal time step. Formally, we define two signals $s(t) = fst({}^\star[\![e]\!]^\rho_G(T_\partial)(t))$ and $s'(t) = fst({}^\star[\![e]\!]^\rho_G(T_{\partial'})(t))$, then:*

$$\forall t \in \mathbb{R},\, n \in \mathbb{N},\;\; st(s)(t,n) = st(s')(t,n)$$

where $fst(v_1, v_2) = v_1$ (first projection).

When satisfied, this invariant ensures that properties and values on non-standard time carry over to standard time and values.

4. A Lustre-like Causality

Programs are statically typed. We adopt, for our language, the type system presented in [13] and is not reminded here. Well-typed programs may still exhibit causality issues, that is, the definition of a signal at instant t may instantaneously depend on itself. A sufficient solution for programs to be causally correct is to reject feedback loops which do not cross a delay. This ensures that outputs can be computed sequentially from current inputs and an internal state, and that programs can be statically scheduled. This is the solution used in the academic Lustre compiler [4], Lucid Synchrone [19] and Scade 6.[18] We propose generalizing it to a language mixing stream equations, ODEs and their synchronous composition. The causality analysis essentially amounts to checking that every loop is broken either by a unit delay or an integrator, nothing more.

The analysis gives sufficient conditions for invariants 1 and 2. We adopt the convention quoted below [13, 10]. A signal is termed *discrete* if it only changes on a *discrete clock*:

> A clock is termed *discrete* if it has been declared so or if it is the result of a zero-crossing or a sub-sampling of a discrete clock. Otherwise, it is termed *continuous*.

[18] http://www.esterel-technologies.com/scade

A discrete change on x at instant $t \in \mathbb{T}$ means that $x(^\bullet t) \not\approx x(t)$ or $x(t) \not\approx x(t^\bullet)$. In other words, all discontinuities have to be announced using the construct $\mathtt{up}(\cdot)$.

Two classes of approaches exist to formalize causality analyses. In the first, causality is defined as an abstract preorder relation on signal names. The causality preorder evolves dynamically at each reaction. A program is causally correct if its associated causality preorder is provably a partial order at every reaction. In the second class, causality is defined as the tagging of each event by a 'stamp' taken from some preordered set. The considered program is causally correct if its set of stamps can be partially ordered—similarly to Lamport vector clocks. Previous works [6, 10] belong to the first class whereas this paper belongs to the second.

Our analysis associates a type to every expression and function via two predicates: (TYP-EXP) states that, under constraints C, global environment G, local environment H, and kind $k \in \{\mathtt{A}, \mathtt{D}, \mathtt{C}\}$, an expression e has type ct; (TYP-ENV) states that under constraints C, global environment G, local environment H, and kind k, the equation E produces the type environment H'.

$$\begin{array}{ll} \text{(TYP-EXP)} & \text{(TYP-ENV)} \\ C \mid G, H \vdash_k e : ct & C \mid G, H \vdash_k E : H' \end{array}$$

The type language is

$$\begin{array}{lcl} \sigma & ::= & \forall \alpha_1, ..., \alpha_n : C,\ ct \xrightarrow{k} ct \\ ct & ::= & ct \times ct \mid \alpha \\ k & ::= & \mathtt{D} \mid \mathtt{C} \mid \mathtt{A} \end{array}$$

where σ defines type schemes, $\alpha_1, ..., \alpha_n$ are type variables and C is a set of constraints. A type is either a pair ($ct \times ct$) or a type variable (α). Intuitively, a type variable is a *time stamp*. The typing rules for causality are defined with respect to an environment of causality types. G is a global environment mapping each function name to a type scheme (σ). H is a local environment mapping each variable x or $\mathtt{last}\, x$ to its type ct:

$$G ::= [f_1 : \sigma_1, ..., f_k : \sigma_k] \qquad H ::= \emptyset \mid H + [x : ct] \mid H + [\mathtt{last}\, x : ct]$$

Precedence relation: C is a precedence relation between variables from $\{\alpha_1, ..., \alpha_n\}$.

$$C ::= \{\alpha_1 < \alpha_1', ..., \alpha_n < \alpha_n'\}$$

$<$ must be a strict partial order: it must not be possible to deduce both $\alpha_1 < \alpha_2$ and $\alpha_2 < \alpha_1$ from the transitive closure of C. If α_1 is interpreted as a time stamp, the intuition is this: all computation must be ordered stictly and statically. If the predicate $C \mid G, H \vdash_k e_1 : \alpha_1$ holds (that is, e_1 has time stamp α_1), $C \mid G, H \vdash_k e_2 : \alpha_2$ holds (that is, e_2 has time stamp α_2) and C is a set of constraints such that $\alpha_1 < \alpha_2$ holds, this means that the result of e_1 is computed stictly before that of e_2. The relation $<$ is lifted to hold for types.

$$\frac{C \vdash ct_1 < ct' \qquad C \vdash ct' < ct_2}{C \vdash ct_1 < ct_2}\ (\textsc{trans}) \qquad \frac{C \vdash ct_1 < ct_1' \qquad C \vdash ct_2 < ct_2'}{C \vdash ct_1 \times ct_2 < ct_1' \times ct_2'}\ (\textsc{pair})$$

$$C + \alpha_1 < \alpha_2 \vdash \alpha_1 < \alpha_2\ (\textsc{taut}) \qquad C \vdash \emptyset < \emptyset\ (\textsc{env-empty}) \qquad \frac{C \vdash H < H' \qquad C \vdash ct_1 < ct_2}{C \vdash H + [x : ct_1] < H' + [x : ct_2]}\ (\textsc{env})$$

$$\frac{C \vdash H < H' \quad C \vdash ct_1 < ct_3 \quad C \vdash ct_2 < ct_4 \quad C \vdash ct_1 < ct_2 \quad C \vdash ct_3 < ct_4}{C \vdash H + [\texttt{last}\, x : ct_1] + [x : ct_2] < H' + [\texttt{last}\, x : ct_3] + [x : ct_4]}\ (\textsc{env-last})$$

Figure 7: Constraints between types

The predicate $C \vdash ct_1 < ct_2$, defined in Figure 7, means that ct_1 precedes ct_2 according to C. All rules are simple distribution rules. The relation $<$ can also be lifted to environments. (ENV) is also a simple morphism. (ENV-LAST) is more interesting: it states that the causality type for $\texttt{last}\, x$ must be less than that of x. Said differently, $\texttt{last}\, x$ must always be computed before x is computed.

When a variable x is initialized (using an equation $\texttt{init}\ x = e$), H associates a causality type to $\texttt{last}\, x$. If H_1 and H_2 are environments, H_1, H_2 is their concatenation, $H + H_2$ is their union provided that their domains are disjoint, and $merge(\alpha)(H_1, H_2)$ is the merge of two environments H_1 and H_2, which is defined as:

$$\begin{array}{lcl} merge(\alpha)((H_1 + [x:\alpha]), (H_2 + [x:\alpha])) & = & (merge(\alpha)(H_1, H_2)) + [x:\alpha] \\ merge(\alpha)((H_1 + [x:\alpha]), H_2) & = & (merge(\alpha)(H_1, H_2)) + [x:\alpha] \\ & & \text{if } x \notin Dom(H_2) \\ merge(\alpha)(H_1, (H_2 + [x:\alpha])) & = & (merge(\alpha)(H_1, H_2)) + [x:\alpha] \\ & & \text{if } x \notin Dom(H_1) \end{array}$$

The initial environment G_0 gives type signatures to imported operators, synchronous primitives and the zero-crossing function.

$$\begin{array}{lcl} (+), (-), (*), (/) & : & \forall \alpha,\ \alpha \times \alpha \xrightarrow{\texttt{A}} \alpha \\ \texttt{pre}(\cdot) & : & \forall \alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\},\ \alpha_1 \xrightarrow{\texttt{D}} \alpha_2 \\ \cdot\ \texttt{fby}\ \cdot & : & \forall \alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\},\ \alpha_1 \times \alpha_2 \xrightarrow{\texttt{D}} \alpha_1 \end{array}$$

Example. The type signature for functions applied point-wise, e.g., $x + y$, is illustrated below. Consider two variables x and y such that $C \mid G, H \vdash x : \alpha_1$ and $C \mid G, H \vdash y : \alpha_2$, that is, x is ready at instant α_1, y at instant α_2. Provided $C \vdash \alpha_1, \alpha_2 < \alpha_3$, it is also the case that $C \mid G, H \vdash x : \alpha_3$, that is, if x has time stamp α_1, it can also be given time stamp α_3. This corresponds to a *subtyping* rule [14]. Applying the same for y, $C \mid G, H \vdash y : \alpha_3$. Now, the

time stamp for $x + y$, that is, the application of function $(+)$ to (x, y) lead to $C \mid G, H \vdash x + y : \alpha_3$. Precisely, if $(+) : \forall\alpha.\alpha \times \alpha \xrightarrow{\mathtt{A}} \alpha$, it can be given the particular instance $\alpha_3 \times \alpha_3 \xrightarrow{\mathtt{A}} \alpha_3$, that is, the time stamp of the result is that of the two arguments.

An uninitialized unit delay like $\mathtt{pre}(x)$ does not depend on x nor does the initialized unit delay x_1 `fby` x_2 whose output depends instantaneously on x_1 but not on x_2. Indeed, if $C \mid G, H \vdash x : \alpha_1$ then $C \mid G, H \vdash \mathtt{pre}(x) : \alpha_2$, with $C \vdash \alpha_2 < \alpha_1$. This means that $\mathtt{pre}(x)$ can be read before x is computed. This explains why the equation $x = \mathtt{pre}(x)$ has no causality loop while $x = x$ has one. Indeed, for it to be causally correct, we would have to prove $C \vdash \alpha_1 < \alpha_1$, under the hypothesis that $C \mid [x : \alpha_1] \vdash_k x : \alpha_1$. This proof is impossible and so the program is rejected.

For $\mathtt{up}(x)$, two policies can be considered that correspond to two signatures:

$$\mathtt{up}(\cdot) : \forall\alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\} : \alpha_1 \xrightarrow{\mathtt{C}} \alpha_2 \qquad\qquad \mathtt{up}(\cdot) : \forall\alpha_1 : \alpha_1 \xrightarrow{\mathtt{C}} \alpha_1$$

In the first one, the effect of a zero-crossing is delayed by one cycle. Hence, $\mathtt{up}(x)$ does not depend instantaneously on x. In the second, the effect is instantaneous.

In the current version of Zélus, writting $\mathtt{up}(x)$ only detects zero-crossings that happen during integration. So, the result of $\mathtt{up}(x)$ does not depend instantaneously on x. Hence, the type signature for $\mathtt{up}(.)$ is the one on the left.

Instantiation/Generalization The types of global definitions are generalized to types schemes (σ) by quantifying over free variables.

$$Gen(C)(ct_1 \xrightarrow{k} ct_2) = \forall\alpha_1, ..., \alpha_n : C.ct_1 \xrightarrow{k} ct_2$$

where $\{\alpha_1, ..., \alpha_n\} = Vars(C) \cup Vars(ct_1) \cup Vars(ct_2)$. The variables in a type scheme σ can be instantiated, $ct \in Inst(\sigma)$ means that ct is an instance of σ. For $\vec{\alpha'}$ and $k \leq k'$:

$$C[\vec{\alpha'}/\vec{\alpha}], ct_1[\vec{\alpha'}/\vec{\alpha}] \xrightarrow{k'} ct_2[\vec{\alpha'}/\vec{\alpha}] \in Inst(\forall\vec{\alpha} : C.ty_1 \xrightarrow{k} ty_2)$$

Example. Consider a function f with type signature:

$$f : \forall\alpha_1, \alpha_2, \alpha_3 : \{\alpha_1, \alpha_2 < \alpha_3\}.\alpha_1 \times \alpha_2 \xrightarrow{k} \alpha_1 \times \alpha_2 \times \alpha_3$$

This signature summarises the following information: the first output depends on the first input; the second output depends on the second input; the third output depend on both inputs. As a consequence, equation $(y_1, y_2, y_3) = f(y_2 + 1, x_1)$, for example is valid. Whereas equations $(y_1, y_2, y_3) = f(y_3 + 1, x_1)$ or $(y_1, y_2, y_3) = f(y_1 + 1, x_2)$ are rejected.

Instantiation of a type signature consists in replacing universally quantified variables by any variable. E.g., a type instance of the signature of f is any type of the form $\alpha'_1 \times \alpha'_2 \xrightarrow{k} \alpha'_1 \times \alpha'_2 \times \alpha'_3$ provided a constraint $C' = \{\alpha'_1, \alpha'_2 < \alpha'_3\}$.

We now define the typing relation according to the syntactic constraints of the language. It is given in Figure 8:

$$\frac{}{C \mid G, H \vdash_k x : H(x)}\ (\textsc{var}) \qquad \frac{}{C \mid G, H \vdash_{\mathtt{D}} \mathtt{last}\, x : H(\mathtt{last}\, x)}\ (\textsc{last}) \qquad \frac{}{C \mid G, H \vdash_k v : ct}\ (\textsc{const})$$

$$\frac{C, ct_1 \xrightarrow{k} ct_2 \in \mathit{Inst}(G(f)) \qquad C \mid G, H \vdash_k e : ct_1}{C \mid G, H \vdash_k f(e) : ct_2}\ (\textsc{app})$$

$$\frac{C \mid G, H \vdash_k x : ct \qquad C \mid G, H \vdash_k e : ct}{C \mid G, H \vdash_k x = e : [x : ct]}\ (\textsc{eq}) \qquad \frac{C \mid G, H \vdash_k e : ct \qquad C \vdash ct < ct'}{C \mid G, H \vdash_k e : ct'}\ (\textsc{sub})$$

$$\frac{C \mid G, H \vdash_{\mathtt{C}} e : ct_1 \qquad C \vdash ct_2 < ct_1}{C \mid G, H \vdash_{\mathtt{C}} \mathtt{der}\, x \;=\; e : [x : ct_2]}\ (\textsc{der}) \qquad \frac{C \mid G, H \vdash_k e : ct \qquad k \in \{\mathtt{D}, \mathtt{C}\}}{C \mid G, H \vdash_k \mathtt{init}\, x = e : [\mathtt{last}\, x : ct]}\ (\textsc{init})$$

$$\frac{C \mid G, H, H' \vdash_k E : H'' \qquad C \vdash H'' < H'}{C \mid G, H \vdash_k E : H''}\ (\textsc{sub-eq})$$

$$\frac{C \mid G, H \vdash_{\mathtt{C}} e : \alpha \qquad C \mid G, H \vdash_{\mathtt{D}} E_1 : H_1 \qquad C \mid G, H \vdash_{\mathtt{C}} E_2 : H_2}{C \mid G, H \vdash_{\mathtt{C}} \mathtt{present}\, e\, \mathtt{then}\, E_1\, \mathtt{else}\, E_2 : \mathit{merge}(\alpha)(H_1, H_2)}\ (\textsc{present})$$

$$\frac{C \mid G, H \vdash_k e : \alpha \qquad \forall i \in \{1, 2\} :\ C \mid G, H \vdash_k E_i : H_i}{C \mid G, H \vdash_k \mathtt{if}\, e\, \mathtt{then}\, E_1\, \mathtt{else}\, E_2 : \mathit{merge}(\alpha)(H_1, H_2)}\ (\textsc{if})$$

$$\frac{C \mid G, H \vdash_k E_1 : H_1 \qquad C \mid G, H \vdash_k E_2 : H_2}{C \mid G, H \vdash_k E_1\, \mathtt{and}\, E_2 : H_1 + H_2}\ (\textsc{and})$$

$$\frac{C \mid G, H + [x : ct_1] \vdash_k E : H' + [x : ct_2] \qquad C \vdash ct_2 < ct_1}{C \mid G, H \vdash_k \mathtt{local}\, x\, \mathtt{in}\, E : H'}\ (\textsc{local})$$

$$\frac{\forall i \in \{1, 2\} :\ C \mid G, H \vdash_k e_i : ct_i}{C \mid G, H \vdash_k (e_1, e_2) : ct_1 \times ct_2}\ (\textsc{pair})$$

$$\frac{C \mid G, H \vdash_k p : ct_1 \quad C \mid G, H, H' \vdash_k E : H'' \quad C \vdash H'' < H' \quad C \mid G, H, H'' \vdash_k e : ct_2}{\vdash \mathtt{let}\, k\, f(p) = e\, \mathtt{where}\, E : [f : \mathit{Gen}(C)(ct_1 \xrightarrow{k} ct_2)]}\ (\textsc{def})$$

Figure 8: The Causality Analysis

Rule (VAR). A variable x inherits the declared type ct.

Rule (LAST). $\texttt{last}\, x$ is the previous value of x. In this system, we only allow $\texttt{last}\, x$ to appear during a discrete step (of kind `D`).

Rule (CONST). A constant v can have any causality type.

Rule (APP). An application $f(e)$ has type ct_2 if f has type $ct_1 \xrightarrow{k} ct_2$ from the instantiation of a type scheme with a new set of constraints C, and e has type ct_1.

Rule (EQ). An equation $x = e$ defines an environment $[ct/x]$ if x and e are of type ct.

Rule (SUB). If e is of type ct and $ct < ct'$ then e can also be given the type ct'.

Rule (SUB-EQ). If an equation E can be typed in an environment $H + H'$ producing a environment H'' such that $H'' < H'$, then E defines the environment H''. The intuition is the following. Let x be a variable defined in E by an equation $x = e$. If $H''(x) = ct''_x$ gives the date at which x is computed and $H'(x) = ct'_x$ is the date at which x can be read, it must be the case that $ct''_x < ct'_x$, that is, the write of x must precede any read of x.

Rule (DER). In terms of causality, an integrator is similar to a unit delay: it breaks dependencies during integration. If $e : ct_1$ then any use of x does not depend instantaneously on the computation of e and can thus be given a type ct_2.

Rule (INIT). An initialization defines the value of $\texttt{last}\, x$. As x is a state variable, an initialization must occur only in a context $k = \texttt{D}$ or $k = \texttt{C}$.

Rules (PRESENT) and (IF). The present statement returns an environment that merges two environments, $merge(\alpha)(H_1, H_2)$. The first handler is activated during discrete steps and the second one has kind `C`. The rule for conditionals is the same except that the handlers and condition must all be of kind k. $merge(\alpha)(H_1, H_2)$ forces all bindings in H_1 and H_2 to be of the form $[x : \alpha]$ where α is the type of the conditional. Forcing variables defined in the two branches to a single type α ensures that they all depend on the conditional. For example, if $C \mid G, H \vdash_k x : \alpha_x$, then $C \mid G, H \vdash_k \texttt{if}\, x\, \texttt{then}\, y = 1\, \texttt{else}\, y = 2 : [y : \alpha_x]$. Without this constraint, y could be wrongly computed before x.

Rule (LOCAL). The declaration of a local variable x is valid if E gives an equation for x which is itself causal.

Rule (DEF). For a function f with parameter p and result e, the body E is first typed under an environment H and constraints C. The resulting environment H' must be strictly less than H. This forbids any direct use of variables in H when typing E.

We can now illustrate the system on several examples presented in the concrete syntax of ZÉLUS.

Examples. We start with the examples given in Section 2.1 on page 10. The causality type signatures are: [19]

```
val min_max: 'a * 'a -A-> 'a * 'a
val count_with_reset: 'a -D-> 'a
val integr : 'a * 'b -C-> 'a
val heat   : 'a * 'b -C-> 'a
```

The signature for `integr` states that the output depends instantaneously on its first argument but not the second one. The following program is statically rejected because no proof derivation can be built.

```
let fun cycle() = (x, y) where rec y = x + 1.0 and x = y + 2.0
```

Consider environments H, G and constraint C such that:

(*a*) $C \mid G, H \vdash_{\mathtt{A}} x : \alpha_x$ and $C \mid G, H \vdash_{\mathtt{A}} y : \alpha_y$

(*b*) Then, $C \mid G, H \vdash_{\mathtt{A}} x + 1 : \alpha_x$ and $C \mid G, H \vdash_{\mathtt{A}} y + 2 : \alpha_y$ (by rule (APP)).

For the body of function `cycle` to be well typed, we need to prove that the result of `x+1` is ready before `y` is read, and `y+2` before `x` is read, that is, both $C \vdash \alpha_x < \alpha_y$ and $C \vdash \alpha_y < \alpha_x$ hold. This is not possible, according to the hypothesis that C must define a strict partial order.

Consider now a hybrid node. The following two functions are well typed.

```
let hybrid f(x) = o where
  rec der y = 1.0 - x init 0.0 and o = y + 1.0

let hybrid loop(x) = y where rec y = f(y) + x

val f    : 'a -C-> 'b
val loop : 'a -C-> 'a
```

The `der` construct plays the role of a unit delay. Consider the typing of function f. Let $H_x = [x : \alpha_x]$ and $H = [y : \alpha_y;\ o : \alpha_o]$. Typing the body $\mathtt{der}\, y = 1.0 - x\ \mathtt{init}\ 0.0\ \mathtt{and}\ o = y + 1.0$ produces an environment $H' = [y : \alpha_{y'};\ o : \alpha_{o'}]$ with a constraint C that must satisfy $\alpha_y < \alpha_{o'}$, $\alpha_{y'} < \alpha_y$ and $\alpha_{o'} < \alpha_o$. The signature is thus $\forall \alpha_x, \alpha_o : C.\alpha_x \xrightarrow{\mathtt{C}} \alpha_o$. Since variables α_y, $\alpha_{o'}$, $\alpha_{y'}$ do not appear as inputs or outputs, C can be reduced to the empty set. The signature $\forall \alpha_x, \alpha_o.\alpha_x \xrightarrow{\mathtt{C}} \alpha_o$ expresses that there is no order constraint between the input and output of f.

The following example is the so-called bouncing ball, starting with initial height `y0` and initial speed `y'0` (`g` is a global constant). When `y` crosses zero from positive to negative, the speed `y'` is reset with `-0.8 * last y'`.

[19] as computed by the ZÉLUS compiler.

```
let hybrid ball(y0, y'0) = y where
  rec der y = y' init y0
  and der y' = - g init y'0 reset up(- y) -> -0.8 * last y'
```

The causality type of `ball` is:

$$\forall \alpha_{y_0}, \alpha_{y'_0}, \alpha_y : \{\alpha_{y_0} < \alpha_y, \alpha_{y'_0} < \alpha_{y'}\}.\alpha_{y_0} \times \alpha_{y'_0} \xrightarrow{\mathtt{C}} \alpha_y$$

It expresses that the output `y` depends instantaneously on `y0` but not on `y'0`. This type can be further simplified into:

$$\forall \alpha_{y_0}, \alpha_{y'_0}.\alpha_{y_0} \times \alpha_{y'_0} \xrightarrow{\mathtt{C}} \alpha_{y_0}$$

because any program that is accepted by giving the first type signature will also be accepted if the second type signature is given. Indeed, consider the equation $y = \mathtt{ball}(y_0, y'_0)$ and $C \mid G, H \vdash_k y_0 : \alpha_{y_0}$, $C \mid G, H \vdash_k y'_0 : \alpha_{y'_0}$. Taking the first (verbose) signature for `ball`, the typing of the equation gives:

$$C \mid G, H \vdash_k y = \mathtt{ball}(y_0, y'_0) : [y : \alpha_y] \text{ with } \{\alpha_{y_0} < \alpha_y, \alpha_{y'_0} < \alpha_{y'}\}$$

Taking the second signature, $\mathtt{ball}(y_0, y'_0)$ gets type α_{y_0} applying rule (APP). Thus, $C \mid G, H \vdash_k y = \mathtt{ball}(y_0, y'_0) : [y : \alpha_{y_0}]$ with $\{\alpha_{y_0} < \alpha_y\}$.

As the rule (LAST) indicates, `last` x can only appear in a discrete context. Hence, the following program is rejected.

```
let hybrid g(x) = o where
  rec der y = 1.0 init 0.0 and x = last x + y and init x = 0.0
```

In the following program, `last o1` is used in a discrete context. This program is causally correct. `last o1` is initialized with the very first value of `x`, and so depends on it. Hence, `o` depends on `x` too.

```
let node f(x) = o where
  rec init o1 = x and o = last o1 + 1.0

val f: 'a -D-> 'a
```

In the following, the function `loop` is rejected as `y` instantaneously depends on itself.

```
let node loop(z) = y where rec y = f(y)
```

For `loop` to be well-typed, we would need to be able to prove that $C \vdash \alpha_y < \alpha_y$, which is not possible.

Type simplification: The ZÉLUS compiler implements a simplification algorithm that eliminates superfluous constraints. Type simplification reduces the size of type constraints and the number of different variables, without which typing would be unpractible (the number of constraints can grow linearly with the size of the function to type). A general solution is proposed in [20]. For ZÉLUS, the simplification is based on the computation of *Input-Output* relations [21]. Moreover, because causality

analysis is performed after typing, some causality relations are not necessary. For example, the signatures of unit delays can be simply:

$$\texttt{pre}(\cdot) : \forall \alpha_1, \alpha_2 : \alpha_1 \xrightarrow{\texttt{D}} \alpha_2 \qquad \cdot \texttt{ fby } \cdot : \forall \alpha_1, \alpha_2 : \alpha_1 \times \alpha_2 \xrightarrow{\texttt{D}} \alpha_1$$

They state that there is no dependency between the input and output of the operator $\texttt{pre}(\cdot)$, and that the output of $\cdot$ `fby` $\cdot$ depends on its first argument only.

The State Port: The present causality analysis restricts the use of `last` x to discrete contexts. A minor extension implemented in the Zélus compiler allows `last` x to appear in a continuous context provided x is a continuous state variable, that is, it is defined by an equation `der` $x = e$. Indeed, during integration `last` x and x are infinitely close to each other (${}^\star x(n-1) \approx {}^\star x(n)$).

5. The Main Theorem

We can now state the main result of this paper: The semantics of well-typed programs satisfies Invariants 1 and 2. This theorem requires assumptions on primitive operators and imported functions, as the following example shows.

A Nonsmooth Program ♣: Several modules, written in Zélus syntax, are required. The first two are an integrator and a time base with a parameterized initial value `t0`:

```
let hybrid integrator(y0, x) = y where
  rec init y = y0 and der y = x

let hybrid time(t0) = integrator(t0, 1.0)
```

We add a function `dirac(d, t)` producing a quasi-Dirac pulse (a Dirac with a width strictly greater than 0), centered at `t` $= 0$, and such that $\int_{-\infty}^{+\infty} \text{dirac}(\texttt{d}, t)dt = 1$ for every constant `d` > 0:

```
let dirac(d, t) = (1.0 / pi) * d / (d * d + t * t)
```

Our goal is to produce, using a hybrid program, an infinitesimal value for `d`, so that `dirac(d, t)` standardizes as a Dirac measure [22]. This is achieved by integrating a pulse of magnitude 1 but of infinitesimal width. Such a pulse can be produced by taking the difference of two variables that each change from 0 to 1, but at instants separated by a ∂ time step:

```
let hybrid doublecrossing(t) = y - x where
  rec init x = 0.0
  and init y = 0.0
  and present up(t) -> do y = 1.0 done else do der y = 0.0 done
  and present (up(last y)) -> do x = 1.0 done
      else do der x = 0.0 done

let hybrid infinitesimal(t) =
  integrator(0.0, doublecrossing(t))
```

Let us assume that parameter `t` is an affine function of time with a constant derivative equal to 1 and a strictly negative initial value. The first zero-crossing in `doublecrossing(t)` occurs at some time $n\partial$ when `t` crosses zero and causes a reset of `x` from -1 to $+1$, at the following infinitesimal time step $(n+1)\partial$. This in turn

triggers an immediate zero-crossing on `x` and a reset of `x` back to -1 at the next time step $(n+2)\partial$. The input of the integrator is thus one for exactly one ∂ time step, and zero elsewhere. The output of the integrator, initially 0, changes to ∂ at time $(n+2)\partial$ and remains unchanged afterward.

The main program is the following, where `t0` $<$ `t1` $<$ `t2`:

```
let hybrid nonsmooth(t0, t1, t2) = x where
  rec t = time(t0) and d = infinitesimal(t - t1)
  and x = integrator(0.0, dirac(d,(t - t2)))
```

What is the point of this example? It is causally correct and yet the standardization of its semantics has a discontinuity at time `t2` though no zero-crossing occurs at this instant. This is because `dirac` standardizes to a Dirac mass centered at `t2` and variable `x` in `nonsmooth` jumps from 0 to 1 at this instant.

Discussion: This discontinuity happens only within the non-standard semantics, where variables can take values from the non-standard reals and discretization is infinitesimal. In practice, the execution of this ZÉLUS program produces a `NaN` (Not a Number) value for variable `x` shortly before time `t2`. This comes from the fact that variable `d` remains at 0 and function `dirac(d,t)` is singular at `t` $= 0$. However, the same program behaves differently with the non-standard semantics, since function `dirac(d,t)` is defined everywhere when `d` $\neq 0$. In particular, it is defined for `d` $= \partial$.

What assumptions are needed to reproduce the behavior observed in practice? The solution seems clear: *if a standard function $f(x)$ of a real variable x is such that $f(x_0) = \perp$ for some x_0, then the semantics must enforce $f(x) = \perp$ for any x infinitesimally close to x_0.* Applying this to the function $d \mapsto \frac{d}{d^2+t^2}$ where $t = 0$ is fixed, gives $\frac{\partial}{\partial^2+t^2} = \perp$. The reason is that when $d = 0$, the function evaluates to $\perp$, and since ∂ is infinitesimal, the function must also evaluate to $\perp$, when $d = \partial$. This simple assumption precludes the possibility of generating a Dirac mass as seen in the example above. This is formalized through the assumptions on operators and functions given below.

Given $x, y \in {}^\star\mathbb{R}$, relation $x \approx y$ holds iff $st(x-y) = 0$. Recall that function $f : {}^\star\mathbb{R} \mapsto {}^\star\mathbb{R}$ is *microcontinuous* iff for all $x, y \in {}^\star\mathbb{R}$, $x \approx y$ implies $f(x) \approx f(y)$. Recall that the microcontinuity of f implies the uniform continuity of $st(f) : \mathbb{R} \mapsto \mathbb{R}$ [23]. Denote $[t_0, t_1]_{\mathbb{T}} = \{t \in \mathbb{T} \mid t_0 \leq t \leq t_1\}$, with $t_0, t_1 \in \mathbb{T}$ finite.

Assumption 1. *Operators $op(\cdot)$ of kind* `C` *are standard and satisfy the following definedness, finiteness and continuity properties:*

$$
\begin{array}{l}
op(\perp) = \perp \\
\forall v,\ op(v) \neq \perp \mathit{\ implies\ } op(v) \mathit{\ finite} \\
\forall u, v,\ u \approx v \mathit{\ and\ } op(u) \not\approx op(v) \mathit{\ implies\ } op(u) = \perp
\end{array}
$$

Assumption 2. *Environment G is assumed to satisfy the following, for all external functions f of kind* `C`*, for any bounded interval $K = [t_1, t_2]_{\mathbb{T}}$, for any input u that is defined, finite and microcontinuous on K, if function $G(f)(u)$ is defined and produces*

no zero-crossing in K, then it is assumed to be finite and microcontinuous on K:

$$\forall t \in K,\ \begin{cases} fst(G(f)(u)(t)) & \neq \bot \text{ and} \\ snd(G(f)(u)(t)) & = \texttt{false} \end{cases}$$
$$\Downarrow$$
$$\begin{array}{l} \forall t \in K,\ fst(G(f)(u)(t)) \textit{ finite, and} \\ \forall t, t' \in K,\ t \approx t' \textit{ implies} \\ fst(G(f)(u)(t)) \approx fst(G(f)(u)(t')) \end{array}$$

Assumption 1 has several implications on the definitions of the usual operators.

- For the square root function: $\sqrt{\epsilon} = \sqrt{-\epsilon} = \sqrt{0}$, for any $\epsilon \approx 0$, which yields two meaningful solutions: $\sqrt{\epsilon} = \bot$ or $\sqrt{\epsilon} = 0$.
- For the inverse operator: $1/\epsilon = \bot$ for any infinitesimal, ϵ is the only solution.
- Consequently, the function $\mathrm{sgn}(x) = x/\sqrt{x^2}$ returning the sign of x must satisfy $\mathrm{sgn}(\epsilon) = \mathrm{sgn}(-\epsilon) = \mathrm{sgn}(0) = \bot$, for any infinitesimal ϵ.

Theorem 1. *Under Assumptions 1 and 2, the semantics of every causally correct equation E (with respect to the typing rules of Section 4) satisfies Invariants 1 and 2 and is standardizable.*

This is a direct consequence of the following lemmas.

Lemma 1. *Assume that Assumptions 1 and 2 hold. For any activation clock $T \subseteq \mathbb{T}$, for any bounded interval $K = [t_1, t_2]_{\mathbb{T}}$, for any environment ρ that is defined, finite and microcontinuous on K, if expression e, of kind* `A` *or* `C`*, is defined and produces no zero-crossing on K, then it is finite and microcontinuous on K:*

$$\forall t \in K,\ \begin{cases} fst({}^{\star}[\![e]\!]^{\rho}_{G}(T)(t)) & \neq \bot \text{ and} \\ snd({}^{\star}[\![e]\!]^{\rho}_{G}(T)(t)) & = \texttt{false} \end{cases}$$
$$\Downarrow$$
$$\begin{array}{l} \forall t \in K,\ fst({}^{\star}[\![e]\!]^{\rho}_{G}(T)(t)) \textit{ finite, and} \\ \forall t, t' \in K,\ t \approx t' \textit{ implies} \\ fst({}^{\star}[\![e]\!]^{\rho}_{G}(T)(t)) \approx fst({}^{\star}[\![e]\!]^{\rho}_{G}(T)(t')) \end{array}$$

Proof: Since ${}^{\star}[\![e]\!]^{\rho}_{G}(T)(t) = \bot, \bot$ for all $t \notin T$, we can assume that $K \subseteq T$. The lemma is proved by induction on the structure of expression e. We prove that it holds for all atomic expressions:

- The semantics of a constant v is a constant function ${}^{\star}[\![v]\!]^{\rho}_{G}(T)(t) = v, \texttt{false}$. Thus it is finite and microcontinuous.
- The semantics of expression x is a function of time defined in environment ρ: ${}^{\star}[\![x]\!]^{\rho}_{G}(T)(t) = \rho(x)(t), \texttt{false}$, which is by assumption defined, finite and microcontinuous on K.
- ${}^{\star}[\![\texttt{last}\, x]\!]^{\rho}_{G}(T)(t)$: $\texttt{last}\, x$ has kind `D` and is thus excluded by assumption.

Then, we assume that the Lemma holds for all causally correct expressions e, e_1 and e_2 of kind `A` or `C`, and prove that it holds for expressions built from e, e_1 and e_2, using one of the following constructors:

- ${}^\star[\![(e_1, e_2)]\!]^\rho_G(T)(t)$ is finite and microcontinuous if and only if ${}^\star[\![(e_i)]\!]^\rho_G(T)(t)$, for $i = 1, 2$ are defined and microcontinuous.
- Consider the application of the operator op on expression e. Two cases must be distinguished: (1) op is of kind `D`, in which case, the expression $op(e)$ has the same kind, which is forbidden by assumption. (2) op is of kind `A`, and by Assumption 1, if defined, the semantics of op is a finite and microcontinuous function. Using the induction hypothesis, ${}^\star[\![op(e)]\!]^\rho_G(T)(t) = op(v), z$ where $v, z = {}^\star[\![e]\!]^\rho_G(T)(t)$ is also finite and microcontinuous.
- e_1 `fby` e_2 expressions have kind `D` and they can only appear in expressions of the same kind.
- In `A` or `C`, expressions of the form $f(e)$, f and e cannot have kind `D`. Therefore Assumption 2 applies to the function f and the induction hypothesis applies to the expression e. Assume ${}^\star[\![f(e)]\!]^\rho_G(T)(t) = v', z(t) \vee z'$, where $v', z' = G(f)(s)(t)$ and $\forall t',\ s(t'), z(t') = {}^\star[\![e]\!]^\rho_G(T)(t')$ is defined and produces no zero-crossing in K. It is then the composition of two finite and microcontinuous functions, and therefore microcontinuous over K.
- ${}^\star[\![\texttt{up}(e)]\!]^\rho_G(T)(t)$ defined and producing no zero crossing for all $t \in K$ implies that it is constant and therefore microcontinuous over K.

This proves that the induction hypothesis holds for all causally correct expressions of kind `A` or `C`. □

Consider the following non-standard system over a bounded interval $T = [t_0, t_1]_{\mathbb{T}}$,:

$$\begin{array}{rcl} x(t_0) & = & x_0 \text{ finite} \\ \forall t \in T \setminus \{t_1\},\ x(t + \partial) & = & x(t) + \partial \times f(t, x(t)) \end{array}$$

Lemma 2. *If the solution $x : T \mapsto {}^\star\mathbb{R}$ of the dynamical system defined above is infinite or discontinuous at t, then there exists $t' < t$ such that $f(t', x(t'))$ is infinite.*

Proof: We will be using the following property, for any $t_1 < t_2$:

$$\exists t' \in T,\ t_1 \leq t' \leq t_2, \text{ such that } \frac{|x(t'+\partial) - x(t')|}{\partial} \geq \frac{|x(t_2) - x(t_1)|}{t_2 - t_1}. \tag{3}$$

First case: assume that $x(t)$ is infinite, for some $t \in T$. Recall $x(t_0)$ is finite. Applying (3) with $t_1 = t_0$, $t_2 = t$ yields the existence of $t', t_0 \leq t' \leq t$ such that $|f(t', x(t'))| \geq \frac{|x(t) - x(t_0)|}{t - t_0}$ is infinite.

Second case: assume $x(t)$ is not continuous for some $t \in T$. There exists a $t' \in T$, $t' \approx t$, such that $x(t') \not\approx x(t)$. Assume without loss of generality that $t' < t$. Observe that $\frac{|x(t) - x(t')|}{t - t'}$ is infinite since $x(t) \not\approx x(t')$ and $t \approx t'$. Applying (3) with $t_1 = t'$ and $t_2 = t$ yields the existence of $t'', t' \leq t'' \leq t$ such that $|f(t'', x(t''))|$ is also infinite. □

Under Assumptions 1 and 2, we have as corollary that the semantics of $\texttt{der}\, x = e$ are smooth if the expression e is defined and does not trigger any zero-crossings:

Corollary 1. *Assume that Assumptions 1 and 2 hold, and that e is a causally correct expression of kind `A` or `C`. For any activation clock $T \subseteq \mathbb{T}$, for any bounded interval $K = [t_1, t_2]_{\mathbb{T}}$, for any environment ρ that is defined, finite and microcontinuous on K, if the least fixed point of the operator $\rho', z' \mapsto {}^\star[\![\texttt{der}\, x = e]\!]^{\rho' + \rho}_G(T)$ is defined and raises no zero-crossings on K, then ρ' is microcontinuous on K.*

Proof: Assume that ρ' is defined and that z' is false on K, and also that ρ' is infinite or discontinuous at $t \in K$. Using Lemma 2, there exists $t' \in K$, $t' < t$ where the semantics of expression e, ${}^{\star}[\![e]\!]_G^{\rho'+\rho}(T)(t')$ is infinite, which contradicts Lemma 1. □

Lemma 3. *Given Assumptions 1 and 2, for any activation clock $T \subseteq \mathbb{T}$, any bounded interval $K = [t_1, t_2]_{\mathbb{T}}$, and for any environment ρ that is defined, finite and microcontinuous on K, if the semantics of E, a causally correct equation of kind* `C`*, is defined and produces no zero-crossings on K, then it is finite and microcontinuous on K:*

$$\forall x, \forall t \in K, \begin{cases} fst({}^{\star}[\![E]\!]_G^{\rho}(T))(x)(t) \neq \bot \text{ and} \\ \quad snd({}^{\star}[\![E]\!]_G^{\rho}(T))(t) = \texttt{false} \end{cases}$$

$$\Downarrow$$

$$\begin{array}{l} \forall x,\ (\forall t \in K,\ fst({}^{\star}[\![E]\!]_G^{\rho}(T))(x)(t) \textit{ finite, and} \\ \forall t, t' \in K,\ t \approx t' \textit{ implies} \\ fst({}^{\star}[\![E]\!]_G^{\rho}(T))(x)(t) \approx fst({}^{\star}[\![E]\!]_G^{\rho}(T))(x)(t')) \end{array}$$

Proof: By induction on the structure of equation E.

- Consider a causally correct equation of kind `C` and of the form $x = e$. The finiteness and microcontinuity of $fst({}^{\star}[\![x = e]\!]_G^{\rho}(T)) = fst({}^{\star}[\![e]\!]_G^{\rho}(T))$ is a direct consequence of Lemma 1.
- Equation `init` $x = e$ defines the value of `last` x to be initialized with the first value of e and to be the previous value of x otherwise. In a context `C` and interval K with no zero-crossings, the equation `init` $x = e$ has no influence on the value of x: either x is constant during the interval K (no derivative is given) or it is defined by an equation `der` $x = e'$.
- For a causally correct equation of kind `C` and of the form `der` $x = e$, Corollary 1 gives that $fst({}^{\star}[\![\texttt{der}\, x = e]\!]_G^{\rho}(T))$ is finite and microcontinuous.

We continue with compositions of equations. Assuming the lemma holds for equations E_1 and E_2, we show that it holds for equations E_1 `and` E_2, `present` e `then` E_1 `else` E_2 and `if` e `then` E_1 `else` E_2:

- Consider a causally correct equation $E = E_1$ `and` E_2 of kind `C`. The finiteness and microcontinuity of the equation $fst({}^{\star}[\![E]\!]_G^{\rho}(T)) = fst({}^{\star}[\![E_1]\!]_G^{\rho}(T)) + fst({}^{\star}[\![E_2]\!]_G^{\rho}(T))$ are consequences of the induction hypothesis.
- Assume the equation $E =$ `present` e `then` E_1 `else` E_2 is causally correct. Since $snd({}^{\star}[\![E]\!]_G^{\rho}(T))$ is equal to `false` at every $t \in K$, $fst({}^{\star}[\![E]\!]_G^{\rho}(T)) = fst({}^{\star}[\![E_2]\!]_G^{\rho}(T))$ is finite and microcontinuous by the induction hypothesis.
- Assume that $E =$ `if` e `then` E_1 `else` E_2 is a causally correct equation of kind `C`. Type correctness implies that expression e is a causally correct expression with the same kind. By Lemma 1, it is microcontinuous on K, and since its values are boolean, it is constant. Without loss of generality, assume the expression evaluates to true. Hence, $fst({}^{\star}[\![E]\!]_G^{\rho}(T)) = fst({}^{\star}[\![E_1]\!]_G^{\rho}(T))$ is finite and microcontinuous by induction hypothesis. □

6. Discussion and Related Work

The present work continues that of Benveniste et al. [10] by exploiting non-standard semantics to define causality in a hybrid program. The proposed analysis gives a sufficient condition for the program to be statically scheduled.

Our work is related to Ptolemy [24] and the use of synchronous language concepts to define the semantics of hybrid modelers [25]. We follow the same path, but replace super-dense semantics by a non-standard one that we found more helpful for explaining causality constraints and generalizing solutions adopted in synchronous compilers. The presented material is implemented in Zélus, a synchronous language extended with ODEs [15]. Zélus is more single-minded than Ptolemy but it allows programs to be compiled into sequential code whereas Ptolemy only provides an interpreter.

Causality has been extensively studied in the synchronous languages Signal [6] and Esterel [7]. Instead of imposing that every feedback loop cross a delay, *constructive causality* checks that the corresponding circuit is constructive. A circuit is constructive if its outputs stabilize in bounded time when inputs are fed with a constant input. In the present work, we adapted the simpler causality of Lustre and Lucid Synchrone based on a precedence relation in order to focus on the specific issues raised when mixing discrete and continuous-time signals. Schneider et al. [26] have considered the causality problem for a hybrid extension of Quartz, a variant of Esterel with ODEs. But, they did not address issues arising from the interaction of discrete and continuous behaviors.

Regarding tools like Simulink, we think that the synchronous interpretation of signals where time advances by infinitesimal steps can be helpful to define causality constraints and safe interactions between mixed signals.

Finally, type signatures can express the way a component may be used. To specify that an output instantaneously depends on an input—the *direct feedthrough port* of a Simulink function,—it suffices to give them the same type variable. For example, the signature $\forall\alpha_1, \alpha_2.\alpha_1 \times \alpha_1 \to \alpha_1 \times \alpha_2$ states that the first output depends on the two inputs and the second output does not depend on any input.

7. Conclusion

Causality in system modelers is a sufficient condition for ensuring that a hybrid system can be implemented: general fix-point equations may have solutions or not, but the subset of causally correct systems can definitely be computed sequentially using off-the-shelf solvers. The notion of causality we propose is that of a synchronous language where instantaneous feedback loops are statically rejected. An integrator plays the role of a unit delay for continuous signals as the previous value is infinitesimally close to the current value.

We introduced the construction `last` x which stands for the previous value of a signal and coincides with the *left limit* when the signal is left continuous. Then, we introduced a causality analysis to check for the absence of instantaneous algebraic loops. Finally, we established the main result: causally correct programs have no discontinuous changes during integration.

The proposed material has been implemented in Zélus, a conservative extension of a synchronous language with ODEs.

8. Acknowledgments

This work has been partially funded by the Sys2Soft, *Briques Génériques du Logiciel Embarqué*, *Investissements d'Avenir* French national project.

References

[1] L. Carloni, R. Passerone, A. Pinto, A. Sangiovanni-Vincentelli, Languages and tools for hybrid systems design, Foundations & Trends in EDA vol. 1.

[2] E. A. Lee, H. Zheng, Operational semantics of hybrid systems, in: Hybrid Systems: Computation and Control (HSCC), Zurich, Switzerland, 2005, pp. 25–53.

[3] The Mathworks, Natick, MA, U.S.A., Simulink 7—User's Guide, 7th Edition (2010).

[4] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous dataflow programming language LUSTRE, Proc. IEEE 79 (9) (1991) 1305–1320.

[5] N. Halbwachs, P. Raymond, C. Ratel, Generating efficient code from data-flow programs, in: 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP), LNCS, Springer, Passau (Germany), 1991, pp. 207–218.

[6] T. Amagbegnon, L. Besnard, P. Le Guernic, Implementation of the data-flow synchronous language Signal, in: Programming Languages Design and Implementation (PLDI), ACM, 1995, pp. 163–173.

[7] G. Berry, The constructive semantics of pure Esterel, unpublished (1999).

[8] L. Gérard, A. Guatto, C. Pasteur, M. Pouzet, A modular memory optimization for synchronous data-flow languages: Application to arrays in a Lustre compiler, in: Languages, Compilers and Tools for Embedded Systems (LCTES), ACM, Beijing, 2012, pp. 51–60.

[9] G. Dahlquist, Å. Björck, Numerical Methods in Scientific Computing: Volume 1, SIAM, 2008.

[10] A. Benveniste, T. Bourke, B. Caillaud, M. Pouzet, Non-Standard Semantics of Hybrid Systems Modelers, Journal of Computer and System Sciences (JCSS) 78 (3) (2012) 877–910, special issue in honor of Amir Pnueli.

[11] O. Maler, Z. Manna, A. Pnueli, From Timed to Hybrid Systems, in: Real-Time: Theory in Practice, Vol. 600 of LNCS, Springer, 1992, pp. 447–484.

[12] The Mathworks, Natick, MA, U.S.A., Simulink 7—Reference, 7th Edition (2010).

[13] A. Benveniste, T. Bourke, B. Caillaud, M. Pouzet, Divide and recycle: types and compilation for a hybrid synchronous language, in: Languages, Compilers, Tools and Theory for Embedded Systems (LCTES), Chicago, USA, 2011, pp. 61–70.

[14] B. C. Pierce, Types and Programming Languages, MIT Press, 2002.

[15] T. Bourke, M. Pouzet, Zélus: A synchronous language with ODEs, in: Hybrid Systems: Computation and Control (HSCC), ACM, Philadelphia, USA, 2013, pp. 113–118.

[16] T. Bourke, J.-L. Colaço, B. Pagano, C. Pasteur, M. Pouzet, A Synchronous-based Code Generator For Explicit Hybrid Systems Languages, in: Compiler Construction (CC), LNCS, London, UK, 2015, pp. 69–88.

[17] G. Kahn, The semantics of a simple language for parallel programming, in: J. L. Rosenfeld (Ed.), IFIP 74 Congress, North-Holland, 1974, pp. 471–475.

[18] A. Benveniste, P. Caspi, R. Lublinerman, S. Tripakis, Actors without directors: a Kahnian view of heterogeneous systems, Tech. Rep. TR-2008-6, Verimag (2008).

[19] M. Pouzet, Lucid Synchrone, version 3. Tutorial and reference manual, Université Paris-Sud, LRI (April 2006).

[20] F. Pottier, Simplifying subtyping constraints: A theory, Information and Computation 170 (2) (2001) 153 – 183.

[21] M. Pouzet, P. Raymond, Modular static scheduling of synchronous data-flow networks: An efficient symbolic representation, in: Embedded Software (EMSOFT), Grenoble, France, 2009, pp. 215–224.

[22] E. Palmgren, Constructive nonstandard representations of generalized functions, Indagationes Mathematicae 11 (1) (2000) 129–138.

[23] T. Lindstrom, An invitation to nonstandard analysis, in: N. Cutland (Ed.), Nonstandard Analysis and its Applications, no. 10 in London Mathematical Society Student Texts, Cambridge Univ. Press, 1988, pp. 1–105.

[24] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, Taming heterogeneity—the Ptolemy approach, Proc. IEEE 91 (1) (2003) 127–144.

[25] E. A. Lee, H. Zheng, Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems, in: Embedded Software (EMSOFT), 2007, pp. 114–123.

[26] K. Bauer, K. Schneider, From synchronous programs to symbolic representations of hybrid systems, in: Hybrid Systems: Computation and Control (HSCC), 2010, pp. 41–50.