A Type-Based Analysis of Causality Loops In Hybrid Systems Modelers

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

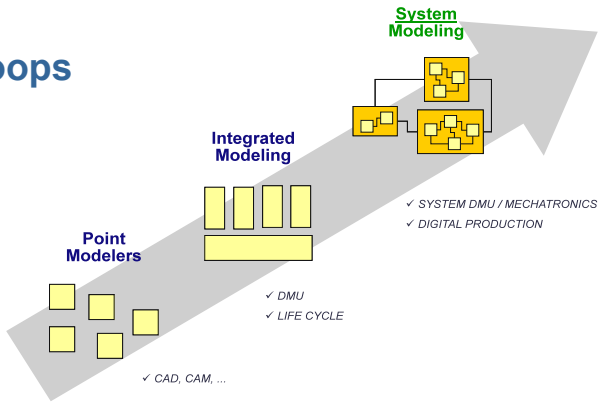
Investissements d'Avenir
Développement de l'Economie Numérique

"Briques génériques du logiciel embarqué"

Sys2soft

Trustable Embedded Software for Systems that are tightly coupled with Physics

WP3.1
D3.1_1
A Type-Based Analysis of Causality Loops In Hybrid Systems Modelers



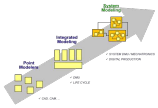
Albert Benveniste (Inria)
Timothy Bourke (Inria)
Benoît Caillaud (Inria)
Bruno Pagano (Esterel Technologies)
Marc Pouzet (École Normale Supérieure)

Version du document: 1.0

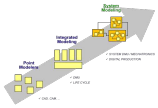
Date: 20/01/2014

Historique du document

Version du document	Date	Rédacteur (s)	Commentaire / modifications
0.0	06/01/2014	Benoît Caillaud (Inria)	Version initiale du livrable
1.0	24/01/2014	Benoît Caillaud (Inria)	Première version du livrable



Contents



1 Synthèse

1.1 Identification du projet

Programme BGLE 2

Projet (Acronyme) Sys2soft

Date de commencement 1er juin 2012

Date d'achèvement 30 novembre 2015

1.2 Objet du document

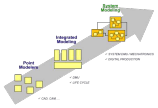
Une question centrale dans la conception des langages de modélisation des systèmes hybrides, y compris Modelica, est la détection, à la compilation, des circuits algébriques ou boucles de causalité. De tels circuits provoquent le blocage du modèle lors de sa simulation et empêchent la génération de code ordonancé statiquement.

Ce livrable détaille une solution à ce problème, pour un langage de modélisation hybride qui combine des equations de flots à la Lustre et des equations différentielles ordinaires. Le langage comporte un opérateur `last`(x) dont la valeur est la limite à gauche de la variable x. Cet opérateur permet de casser des circuits algébriques et a l'avantage de s'appliquer indifféremment sur des variables discretes ou continues. La sémantique du langage est à base de nombres réels non-standards et définit une exécution comme une suite de pas, progressant de manière infinitésimale. Un signal est considéré *causalement correct* quand il peut être calculé séquentiellement et est continu en dehors des instants où un pas de calcul discret a lieu. L'analyse de causalité est définie sous la forme d'un système d'inférence de type . Il est prouvé que dans tout programme correctement typé, les signaux sont continus en dehors des seuls instants où des calculs discrets ont lieu. Cette analyse de causalité permet de générer un code de simulation ordonnancé statiquement qui fait appel à une biblioth‘eque standard de solveurs de systèmes déquations differentielles.

La pertinance de cette approche est illustrée par plusieurs exemples écrits dans le langage Zélus, qui est un langage de modélisation des systèmes hybrides, qui combine des equations de flôts synchrones et des equations différentielles.

1.3 Contraintes

Ce document est publique.



2 Causality and Scheduling Issues in Modelers

Tools for modeling hybrid systems [8] 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 a formal operational semantics for specifying their implementations and proving them correct [18, 10].

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

Causality or *algebraic* loops [22, 2-34] pose problems of well-definedness and compilation. They can lead to models that are mathematically unsound, and prevent simulators from statically ensuring the existence of a fixed point, and compilers from generating statically scheduled code. The static detection of such loops, termed *causality analysis*, has been extensively studied and implemented since the mid-1980s in synchronous language compilers combining stream equations and control structures [13, 14, 15, 6, 1]. The classical and simplest solution is to reject loops which do not cross a unit delay. For instance, the LUSTRE-like equations:[4]

```
x = 0 -> pre y    and    y = if c then x + 1 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` (for "previous"). Replacing `pre y` with `y` would make the two equations non-causal. Causally correct equations can be statically scheduled to produce a sequential loop-free *step* function. Below is an excerpt of the C code generated by the HEPTAGON compiler [12] of LUSTRE:

```
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 and `self->v_2` storing the value of `pre y`.

[1] `http://www.modelica.org`
[2] `http://www.ni.com/labview`
[3] `http://www.mathworks.com/products/simulink`
[4] The unit delay `0 -> pre`$(\cdot)$, initialized to 0, is written $\frac{1}{z}$ in SIMULINK.

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

```
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 $t \in \mathbb{R}^+$, $\frac{dy}{dt}(t) = z(t)$, $y(0) = 4.0$, $z(t) = 10.0 - 0.1 \cdot y(t)$, and $k(t) = y(t) + 1$.[5] This program is causal simply because it is possible to generate a sequential function $\mathit{derivative}(y) = \texttt{let}\, z = 10.0 - 0.1 * y\, \texttt{in}\, z$ and initial value 4.0 for y so that a numeric solver [9] can compute a sequence of approximations $y(t_n)$ for increasing values of time $t_n \in \mathbb{R}^+$ and $n \in \mathbb{N}$. Thus, for continuous-time signals, integrators break algebraic loops just as delays do for discrete-time signals.

Can we reuse the simple justification we used for data-flow equations? Consider the value that y would have if computed by an ideal solver taking an infinitesimal step of duration ∂ [5]. 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. E.g., the sawtooth signal $y : \mathbb{R}^+ \mapsto \mathbb{R}^+$ such that $\frac{dy}{dt}(t) = 1$ and $y(t) = 0$ if $t \in \mathbb{N}$ can be defined by an ODE with 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 from negative to positive. 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:

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

This set of equations is clearly not causal: 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 $z(n)$, or, *(b)* distinguish the current value of ${}^\star y(n)$ from the value it

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

would have had were there no reset, namely ${}^\star ly(n)$. Testing a zero-crossing of ly (instead of y),

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

gives a program that is causal since ${}^\star y(n)$ no longer depends instantaneously on itself. We propose writing this ♣[6]:

```
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 [5], it is infinitely close to the previous value of y, and written $ly(n) \approx y(n-1)$. In the case where y is defined by its derivative, `last`(y) is the so-called "state port" of the integrator block $\frac{1}{s}$ of Simulink, which is introduced expressly to break causality loops like the one above ♣.[7] According to the Simulink documentation [21, 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 it cannot be returned as a block 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 causes subtle bugs in the Simulink compiler. Consider the Simulink model given 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 x - 2.0)
```

Replaying 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 their previous values, that is:

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

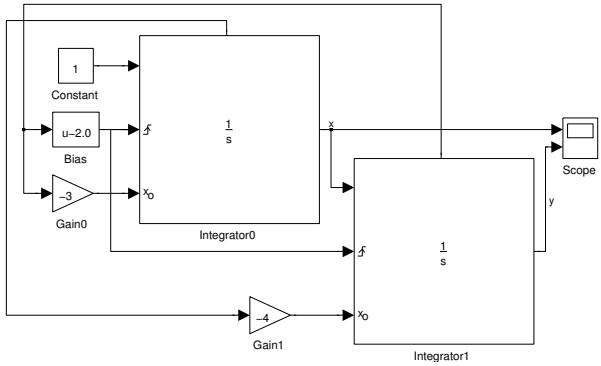
[6] The ♣'s link to the web page: `http://zelus.di.ens.fr/hscc2014/`.

[7] 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 example would thus be written:
$(y, ly) = \frac{1}{s}(0.0, \texttt{up}(ly - 1.0), 1.0)$.

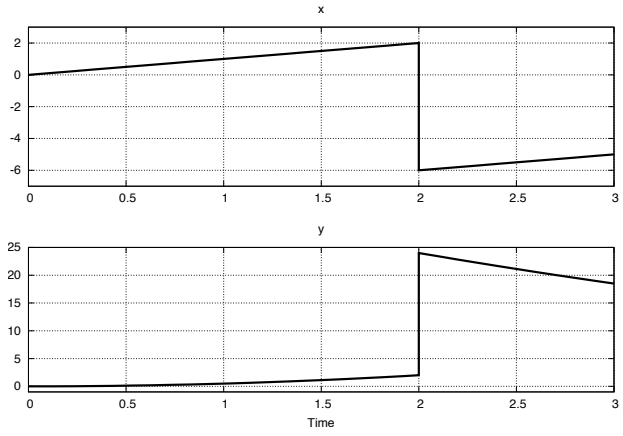
$$^\star x(0) = 0 \qquad ^\star y(0) = 0$$
$$^\star c(n) = (^\star x(n-1) - 2) \geq 0 \qquad ^\star z(n) = {}^\star c(n) \wedge \neg {}^\star c(n-1)$$

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 $(= 2 \cdot -4)$!



(a) Simulink model



(b) Simulation results

Fig. 1: A miscompiled Simulink model (release R2009b) ♣

The SIMULINK implementation does not satisfy its documented behavior [21, 2-685]. Inspecting the C function which computes the current outputs, `mdlOutput` in Figure 2, the code of the two integrators appears in an incorrectly scheduled sequence. At the instant of

the zero-crossing (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 (${}^\star x(n)$) leading to an incorrect output. The SIMULINK model is not correctly compiled — it needs another variable 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 give rise to a warning or error message.

Any loop in SIMULINK, whether of discrete- or continuous-time signals, can be broken by inserting the so-called memory block [21, 2-831].[8] 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 [4].[9] 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 very common: it is typically that of a system with continuous modes M_1 and M_2 producing a signal x, with each of them being started with the value computed previously by the solver, and $\texttt{mem}(x)$ being used to pass a value from one mode to the other ♣. Instead of the unsafe operator $\texttt{mem}(x)$, we better need to refer to the *left limit* of x, and write it again $\texttt{last}(x)$. Yet, the unrestricted use of this operation may cause a new kind of causality loop which have to be statically rejected. Consider the following equation activated at a continuous time base:

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

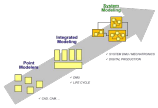
which defines, for all $n \in {}^\star\mathbb{N}$, the sequence ${}^\star y(n)$ such that:

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

Indeed, there is little difference with an equation `y = -1.0 * y`. Even though ${}^\star y(n)$ can be computed sequentially, its value does not increase infinitesimally at each step, that is, y is not left continuous while no signal is looked for a zero-crossing. 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.

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

[9] Quoting the SIMULINK manual (`http://www.mathworks.com/help/simulink/slref/memory.html`), "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."




```
// 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;
       }
      ...
    }
    ... }
```

Fig. 2: Excerpt of C code produced by RTW (release R2009b)

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 `last`(x) stands for the previous value of x in non standard semantics and coincide with its left-limit when x is left-continuous. This operation plays the role of a delay but is safer than the memory block `mem`(x) as its semantics does not depend on any particular solver. When x is a continuous-state variable, it coincides with the so-called Simulink *state port*. We develop a non-standard semantics following [5] 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 [23]. A type signature for a function expresses the instantaneous dependences 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. We are not aware of such a correctness theorem based on static typing for hybrid modelers.

The presented material has been implemented in ZÉLUS, [7] a synchronous LUSTRE-like language extended with ODEs. Moreover, all examples in the paper are written with ZÉLUS.

The paper is organized as follows. Section 3 introduces a core synchronous language with ODEs. Section 4 presents its semantics based on non-standard analysis. Section 5 presents two type systems for causality and Section 6 a major property: any well-typed program is proved not to have any discontinuities during integration. Section 7 discusses related work and we conclude in Section 8.

3 A Core Synchronous Language with ODEs

We now introduce a kernel language. It is not intended to be a full language but a minimal one for the purpose of the present paper. It allows for writing data-flow equations, ordinary differential equations and control structures. The syntax is given below.

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

A program is a sequence of definitions (d), of either a value ($\texttt{let}\, x = e$) that binds the value of expression e to x, or 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, e.g., 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), e.g., 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 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 ($\texttt{init}\ x = e$); the value of a state variable x at the next instant ($\texttt{next}\ x = e$); 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*) $\texttt{if}\ e\ \texttt{then}\ E \stackrel{def}{=} \texttt{if}\, e\, \texttt{then}\, E\, \texttt{else}\, ()$.

(*b*) $\texttt{present}\ e\ \texttt{then}\ E \stackrel{def}{=} \texttt{present}\ e\ \texttt{then}\ E\ \texttt{else}\ ()$.

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

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

Equations (E) must be in Static Single Assignment (SSA) form, that is, every variable has a unique definition at every instant.

4 Non-standard Semantics and Standardization

4.1 Non-Standard 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 that is bounded from above (resp. below) has a unique maximal (resp. minimal) element. Let $\partial \in {}^\star\mathbb{R}$ be an infinitesimal, i.e., $\partial > 0$, $\partial \approx 0$. 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 its total order from ${}^\star\mathbb{N}$; in addition, for every 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 will thus be of the form $k\partial$ where $k \in {}^\star\mathbb{N}$. If $T \subseteq \mathbb{T}$, we write ${}^\bullet T(t)$ the immediate predecessor of t in T and $T^\bullet(t)$ 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. The clock of a signal x is $clock(x) = \{t \in \mathbb{T} \mid x(t) \neq \bot\}$.

Definition 1 (Signals): Let $V_\perp = V + \{\perp\}$ where V is a set. $S(V) = \mathbb{T} \mapsto V_\perp$ is the set of signals. 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.

Sampling: Let $\texttt{bool} = \{\texttt{false}, \texttt{true}\}$ and $x : T \mapsto \texttt{bool}_\perp$. The *sampling of T according to x*, written T $\texttt{on}$ 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 $\texttt{on}$ $x \subseteq T$, it is also totally ordered. Let $x : T \mapsto {}^\star\mathbb{R}_\perp$. The zero-crossing of an x is $up(x) : T \mapsto \texttt{bool}_\perp$. To underline the fact that $up(x)$ is defined only when $t \in T$, we write $up(x)(T)(t)$ for its value at time t. Outside of T, $up(x)(T)(t) = \perp$. In the definition below, $<$ is the total order on ${}^\star\mathbb{R}$.

$$\begin{aligned} up(x)(T)(t_0) &= \texttt{false} \text{ where } t_0 = min(T) \\ up(x)(T)(t) &= \exists n \in {}^\star\mathbb{N}, n{\geq}1. \ \wedge (x(t{-}n\partial) < 0) \\ &\qquad\qquad\qquad\quad \wedge (x(t{-}(n{-}1)\partial) = 0) \\ &\qquad\qquad\qquad\quad \wedge \ \ldots \\ &\qquad\qquad\qquad\quad \wedge (x(t{-}\partial) = 0) \\ &\qquad\qquad\qquad\quad \wedge (x(t) > 0) \\ &\qquad \text{where } t \in T \end{aligned} \tag{1}$$

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 intermediate values equal to 0.

Let V be a set of values closed under product and sum. ${}^\star V$ is its non-standard extension such that ${}^\star(V_1 \times V_2) = {}^\star V_1 \times {}^\star V_2$, ${}^\star V = V$ for any finite set V. ${}^\star V_\perp = {}^\star V + \{\perp\}$ with $\perp$ as the minimum element. Let $L = \{x_1, ..., x_n, ...\}$ be a set of local variables and $L_g = \{f_1, ..., f_n, ...\}$ a set of global variables. An environment associates names to values. A local environment ρ and a global environment G map names to signals and signal functions:

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

Operations on environments: Consider ρ_1 and ρ_2.

- $(\rho_1 + \rho_2)(x)(t)$ is $\rho_1(x)(t)$ if $\rho_2(x)(t) = \perp$, $\rho_2(x)(t)$ if $\rho_1(x)(t) = \perp$, and $\perp$ otherwise.
- $\rho = merge\,(T)\,(s)\,(\rho_1)\,(\rho_2)$ is the merge of two environments according to a signal $s \in S(\texttt{bool})$. The value of a signal x at instant $t \in T$ is the one given by ρ_1 if $s(t)$ is true and that of ρ_2 otherwise. Nonetheless, in case x is not defined in ρ_1 (respectively ρ_2), it implicitly keeps its previous value, that is $\rho_1({}^\bullet clock(x)(t))$. This corresponds to adding an equation $x = \texttt{last}(x)$ when no equation is given in one branch of a conditional. For all x and instant $t \in T$, $\rho(x)(t) = \rho_1(x)(t)$ if $s(t) = \texttt{true}$ and $x \in Dom(\rho_1)$; $\rho(x)(t) = \rho(x)({}^\bullet clock(x)(t))$ if $s(t) = \texttt{true}$ and $x \notin Dom(\rho_1)$. $\rho(x)(t) = \rho_2(x)(t)$ if $s(t) = \texttt{false}$ and $x \in Dom(\rho_2)$; $\rho(x)(t) = \rho(x)({}^\bullet clock(x)(t))$ otherwise.

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

Fig. 3: The Non-standard Semantics of Expressions

Expressions: Expressions are interpreted as signals and node definitions as functions from signals to signals. For expressions, we define ${}^{\star}[\![e]\!]^{\rho}_{G}(T)(t)$ such that for every instant $t \in T$, it returns both the value of e and a Boolean value true if e raises a zero-crossing event. The definition is given in Figure 3.

Let us explain the definition. 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 environment $\rho(x)$ and no event is raised. The semantics of $op(e)$ 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 of e_1 or e_2 raises one. 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. E.g., 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 semantics of $f(e)$ is the application of the function f to the signal value of e, which raises an event when either e or the body of f raises one. Finally, the semantics of $\texttt{up}(e)$ is given by operator $up(.)$, which raises a zero-crossing event when

$$
\begin{array}{lll}
{}^{\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) \wedge \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) = & & \\
\qquad \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) \\
& \textit{and} & \rho_1, z_1 = {}^{\star}[\![E_1]\!]^{\rho}_{G}(T \texttt{ on } s) \\
& \textit{and} & \rho_2, z_2 = {}^{\star}[\![E_2]\!]^{\rho}_{G}(T \texttt{ on } not(s)) \\
& \textit{and} & \rho' = merge\,(T)\,(s)\,(\rho_1)\,(\rho_2) \\
{}^{\star}[\![\texttt{if } e \texttt{ then } E_1 \texttt{ else } E_2]\!]^{\rho}_{G}(T) = & & \\
\qquad \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) \\
& \textit{and} & \rho_1, z_1 = {}^{\star}[\![E_1]\!]^{\rho}_{G}(T \texttt{ on } s) \\
& \textit{and} & \rho_2, z_2 = {}^{\star}[\![E_2]\!]^{\rho}_{G}(T \texttt{ on } not(s)) \\
& \textit{and} & \rho' = merge\,(T)\,(s)\,(\rho_1)\,(\rho_2) \\
{}^{\star}[\![\texttt{init } x = e]\!]^{\rho}_{G}(T) = [s/x], z & \textit{where} & s(t_0), z(t_0) = {}^{\star}[\![e]\!]^{\rho}_{G}(T)(t_0) \\
& \textit{and} & t_0 = min(T) \\
& \textit{and} & \forall t \neq t_0.s(t) = \rho(x)(t) \wedge z(t) = \texttt{false} \\
{}^{\star}[\![\texttt{next } x = e]\!]^{\rho}_{G}(T) = [s/x], z & \textit{where} & \forall t \in T.\ (v, z = {}^{\star}[\![e]\!]^{\rho}_{G}(T)(t)) \wedge (s(t^{\bullet}) = v) \\
{}^{\star}[\![\texttt{der } x = e]\!]^{\rho}_{G}(T) = [s/x], z & \textit{where} & \forall t \in T.\ (v, z = {}^{\star}[\![e]\!]^{\rho}_{G}(T)(t)) \wedge \\
& & \qquad (s(t^{\bullet}) = s(t) + \partial \times v)
\end{array}
$$

Fig. 4: The Non-standard Semantics for Equations

either e raises one 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 a zero-crossing occurs at instant t and $z(t) = \texttt{false}$ means that no zero-crossing occurred at that instant. The semantics of equations is given in Figure 4, 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, $(z_1 \textit{ or } z_2)(t) = \bot$.

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 $S(V)$ indexed by $\mathbb{T}$ is defined as: signal x is a *prefix* of signal y, written $x \leq_{S(V)} y$, if $x(t) \neq y(t)$ implies $x(t') = \bot$ for all t' such that $t \leq t'$. The minimum element is the undefined signal $\bot_{S(V)}$ for which $\forall t \in \mathbb{T}$, $\bot_{S(V)}(t) = \bot$. When possible, we write $\bot$ for $\bot_{S(V)}$ and $x \leq y$ for $x \leq_{S(V)} y$. The symbol $\bigvee$ denotes a supremum in the prefix order. A function $f : S(^\star V) \mapsto S(^\star V)$ 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 $\bigvee_i f^i(\bot)$. We name such continuity on the prefix order *Kahn continuity* [16].

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

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

such that:

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

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

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

Proof: We only provide a sketch. We first need to prove the result for expressions listed in Figure 3. We only review the expressions involving the non-standard semantics in a nontrivial way, as the other cases are routine. Consider $e_1\, \texttt{fby}\, e_2$ and $\texttt{last}(x)$. 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 $e_1\, \texttt{fby}\, e_2$ is proved exactly as for Lustre [3], 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 $\texttt{last}(x)$, which corresponds to $\texttt{pre}(x)$ in Lustre in the lifting from $\mathbb{N}$ to $\mathbb{T}$. The expression $\texttt{up}(e)$ 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 4. We discuss only $\texttt{next}\ x = e$ and $\texttt{der}\, x = e$ since the other cases as handled as in Lustre (including the composition of equations $E_1\ \texttt{and}\ E_2$). Consider the first field of the semantics. If e returns the value v at the considered instant t, then the first field of x returns v at the next instant $t^\bullet$, i.e., $x(t^\bullet) = v(t)$. Kahn continuity follows directly. The same reasoning holds for $\texttt{der}\, x = e$. □

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

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

and with

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

Yet, Kahn-continuity of $^\star f$ does not mean that the function computes anything interesting. In particular, the semantics gives a unique meaning to functions that become 'stuck', like[10]

```
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.

4.2 Standardization

We now relate the non-standard semantics to the usual super-dense semantics of hybrid systems. Following [20], 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* [20, 18] 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:

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

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

$$\mathit{st}(x) : \mathbb{R} \times \mathbb{N} \mapsto V_\bot, \text{ such that } \mathit{st}(x)(t, n) = \bot \text{ for } n > N_x(t).$$

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

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

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

[10] The keyword `hybrid` stands for $k = \texttt{C}$ and `node` for $k = \texttt{D}$.

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

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

i.e., 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 [18], 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.

4.3 Key properties

We now define two main properties that reasonable programs should satisfy. 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.

Invariant 1 (All discontinuities aligned on zero-crossings): An expression e evaluated under G, ρ and a base time T has no discontinuity outside of zero-crossing events. Formally, 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 signals must evolve continuously during integration. Discrete changes must be announced to the solver using the construct `up`(.). Not all programs satisfy the invariant, e.g.,

```
let hybrid f()= y where rec y= last y + 1 and init y= 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 by using the type system developed 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, define $s(t) = fst({}^\star[\![e]\!]^\rho_G(T_\partial)(t))$ and $s'(t) = fst({}^\star[\![e]\!]^\rho_G(T_{\partial'})(t))$, then:

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

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

5 Two Type-based Causality Analyses

Programs are statically typed. We adopt, for our language, the type system presented in [4]. Well-typed programs may still exhibit causality issues, that is, the definition of a signal at instant t may instantaneously depend on itself. We present two systems. The first essentially amounts to checking that every loop is broken either by a unit delay or an integrator. It is reminiscent of the causality analysis of both Lustre and Lucid Synchrone. Yet, the operation `last`(x) can only be activated at a discrete instant. The second is more expressive and exploits the fact that `last`(x) is the left limit of a signal allowing to analyze the Simulink example of Section 2. It thus breaks causality loops during discrete steps but not during integration steps.

The analysis aims to give sufficient conditions for the invariants 1 and 2. We adopt the convention quoted below [4, 5]. 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*.

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

5.1 A Lustre-like Causality Analysis

A classical causality analysis is to reject loops which do not cross a delay. This ensures that outputs can be computed sequentially from current inputs and an internal state. This simple solution is used in the academic Lustre compiler [13], Lucid Synchrone [24] and Scade 6.[11] We propose generalizing it to a language mixing stream equations, ODEs and their synchronous composition.

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

Two classes of approaches exist to formalize causality analyses. In the first class, the causality is defined as an abstract preorder relation on signal names. The causality preorder evolves dynamically at each reaction. The considered program is causally correct if its associated causality preorder is provably a partial order at every reaction. In the second class, the 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—somehow like Lamport vector clocks. Previous works [1, 5] belong to the first class, whereas this paper belongs to the second class.

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'.

$$\frac{}{C \mid G, H \vdash_k e : ct}\ (\text{TYP-EXP}) \qquad \frac{}{C \mid G, H \vdash_k E : H'}\ (\text{TYP-ENV})$$

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). 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 to its type ct:

$$G ::= [\sigma_1/f_1, ..., \sigma_k/f_k] \qquad H ::= [ct_1/x_1, ..., ct_n/x_n]$$

If H_1 and H_2 are environments, $H_1 + H_2$ is their disjoint union. H_1, H_2 is their concatenation; and $H_1 * H_2$ is a new environment such that $(H_1 + [x : ct]) * (H_2 + [x : ct]) = (H_1 * H_2) + [x : ct]$ where $+$ and $*$ are associative and commutative.

Precedence relation: C is a precedence relation between variables with the following intuition. If $C \mid G, H \vdash_k e : \alpha_1$ holds and $\alpha_1 < \alpha_2$, the current value of e is ready at time α_1 then it is also ready at a later instant α_2. $<$ must be a strict partial order: it must not be possible to deduce $\alpha < \alpha$ from the transitive closure of C.

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

The predicate $C \vdash ct_1 < ct_2$, defined in Figure 5, means that ct_1 precedes ct_2 according to C. All rules are simple distribution rules.

$$\text{(TAUT)}\quad C + \alpha_1 < \alpha_2 \vdash \alpha_1 < \alpha_2 \qquad \text{(TRANS)}\quad \frac{C \vdash ct_1 < ct' \qquad C \vdash ct' < ct_2}{C \vdash ct_1 < ct_2}$$

$$\text{(PAIR)}\quad \frac{C \vdash ct_1 < ct_1' \qquad C \vdash ct_2 < ct_2'}{C \vdash ct_1 \times ct_2 < ct_1' \times ct_2'} \qquad \text{(ENV)}\quad \frac{\forall i \in \{1,..,n\}, C \vdash ct_i < ct_i'}{C \vdash [x_1 : ct_1; ...; x_n : ct_n] < [x_1 : ct_1'; ...; x_n : ct_n']}$$

Fig. 5: Constraints between types

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

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

For example, the operation $x + y$ depends on both x and y, that is, it must be computed after x and y have been computed. Indeed, if $C \mid G, H \vdash x : \alpha_1$ and $C \mid G, H \vdash y : \alpha_2$, $C \vdash \alpha_1 < \alpha$ and $C \vdash \alpha_2 < \alpha$, then $C \mid G, H \vdash x + y : \alpha$. $\mathtt{pre}(x)$ does not depend on x. Moreover, $\mathtt{pre}(x)$ has to be used before x is computed. For $\mathtt{up}(x)$, we consider in this first system that the effect of a zero-crossing is delayed by one cycle. Hence, $\mathtt{up}(x)$ does not depend instantaneously on x.

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 $k \leq k'$:

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

The typing relation is defined in Figure 6 and described below.

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

Rule (CONST). A constant v has any causality type.

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

$$\frac{C, ct_1 \xrightarrow{k} ct_2 \in Inst(G(f)) \qquad C \mid G, H \vdash_k e : ct_1}{C \mid G, H \vdash_k f(e) : ct_2}\ (\textsc{app}) \qquad \frac{C \vdash ct_2 < ct_1}{C \mid G, H + x : ct_1 \vdash_{\mathtt{D}} \mathtt{last}(x) : ct_2}\ (\textsc{last})$$

$$\frac{C \mid G, H \vdash_k p : ct \qquad C \mid G, H \vdash_k e : ct}{C \mid G, H \vdash_k p = e : [ct/p]}\ (\textsc{eq}) \qquad \frac{C \mid G, H \vdash_{\mathtt{C}} e : ct_1 \qquad C \vdash ct_2 < ct_1}{C \mid G, H \vdash_{\mathtt{C}} \mathtt{der}\, x \ = \ e : [ct_2/x]}\ (\textsc{der})$$

$$\frac{C \mid G, H \vdash_{\mathtt{C}} e : ct}{C \mid G, H \vdash_{\mathtt{C}} \mathtt{init}\ x = e : [ct/x]}\ (\textsc{init}) \qquad \frac{C \mid G, H \vdash_{\mathtt{D}} e : ct_1 \qquad C \vdash ct_2 < ct_1}{C \mid G, H \vdash_{\mathtt{D}} \mathtt{next}\ x = e : [ct_2/x]}\ (\textsc{next})$$

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

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

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

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

$$\frac{C \mid G, H \vdash_k E : H' \qquad C \vdash H' < H \qquad \forall i \in \{1, 2\} : \ C \mid G, H \vdash_k x_i : ct_i}{\vdash \mathtt{let}\, k\, f(x_1) = x_2\, \mathtt{where}\, E : [Gen(C)(ct_1 \xrightarrow{k} ct_2)/f]}\ (\textsc{def})$$

Fig. 6: A Lustre-like Causality Analysis

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

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

Rule (EQ). An equation $p = e$ defines an environment $[ct/p]$ if p 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 (DER). An integrator has a similar role as 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 such that $ct_2 < ct_1$.

Rule (PRESENT). The present statement returns an environment $H_1 * H_2$. The first handler is activated during discrete steps and the second one has kind `C`.

Rule (IF). This rule is the same as that of the present statement except that the handlers and condition must all be of kind k.

Rule (DEF). For a function f with parameter x_1 and result x_2, 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.

Example The following program is a classic synchronous (thus discrete-time) program. Calling the forward Euler integrator `integr` below, the function `heat` is valid since `temp` does not depend instantaneously on `gain - temp`. `step` is a global constant.

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

let node heat(temp0, gain) = temp where
  rec temp = integr(temp0, gain - temp))
```

The causality signatures are:

```
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:

```
let cycle() = (x, y) where rec y = x + 1 and x = y + 2
```

Indeed, taken $x : \alpha_x$ and $y : \alpha_y$, the first equation is correct if both $C \vdash \alpha_x < \alpha_y$ and $C \vdash \alpha_y < \alpha_x$. This means that C must contain $\{\alpha_x < \alpha_y, \alpha_y < \alpha_x\}$ which is cyclic. This one is correct:

```
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    : {'b < 'a }.'a -C-> 'b
val loop : 'a -C-> 'a
```

Yet, the type system is that of a synchronous language and the restriction of $\texttt{last}(x)$ to appear only in a discrete context is quite restrictive. We do now a little better.

5.2 A Schizophrenic Causality Analysis

To properly analyse what the causality rules should be when relaxing the use of $\texttt{last}(x)$, let us turn to the non-standard semantics. In non-standard semantics, $\texttt{last}(x)$ is the previous value of x.

$$\texttt{last}(x)(t) = x({}^{\bullet}t)$$

When x is left-continuous, it coincides with the left limit since $\texttt{last}(x)(t) \approx x(t)$. Otherwise, it is the previously computed value of x. This means, in particular, that if a sequence of zero-crossings appears consecutively, termed a *zero-crossing cascade*, the current value of $\texttt{last}(x)$ may change at every instant. Should we consider that $\texttt{last}(x)$ breaks a causality cycle on x when x is continuous? No, since the equation $x = \texttt{last}(x) + 1$ with x initialized to 0 has no standard part. Thus, $\texttt{last}(x)$ only breaks an algebraic loop during a discrete step, that is, when x may not be left-continuous. This is intuitive operationally if we consider the way simulation is performed in a hybrid modeler. A modeler alternates discrete steps (where side effects and state changes can occur) and integration steps. Consider a boolean variable d, true during discrete steps and otherwise false. The implementation of $\texttt{last}(x)$ is simply:

$$\texttt{last}(x) = \texttt{if}\, d\, \texttt{then}\, \texttt{pre}(x)\, \texttt{else}\, x$$

During a discrete step, $\texttt{last}(x)$ is a discrete register, otherwise it is the identify function. The dependence information associated to $\texttt{last}(x)$ is thus conditional. The principle is to associate a pair $ct_1 + ct_2$ to every expression e such that *(a)* during discrete steps, e only depends on ct_1, and, *(b)* during integration steps, e only depends on ct_2 . The type language is modified accordingly.

$$\begin{array}{rcl} \sigma & ::= & \forall \alpha_1, ..., \alpha_n : C.\ ct \to ct \\ ct & ::= & ct \times ct \mid \alpha + \alpha \mid \alpha \end{array}$$

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 $(ct \times ct)$, a compound of two type variables $(\alpha + \alpha)$ or a type variable (α).

The operation $\alpha_1 + \alpha_2$ is lifted to types. We consider the equality of types modulo the following equation:

$$(ct_1 \times ct_2) + (ct'_1 \times ct'_2) = (ct_1 + ct'_1) \times (ct_2 + ct'_2)$$

The typing rules for causality are also defined for typing environments. The structure of G and H is preserved and constraints are extended to account for types $ct_1 + ct_2$ in a straightforward manner.

$$\text{(SUM)}\quad \frac{C \vdash ct_1 < ct'_1 \qquad C \vdash ct_2 < ct'_2}{C \vdash ct_1 + ct_2 < ct'_1 + ct'_2}$$

Similarly, instantiation and generalization are unchanged. But we replace the rule (LAST) of the system in Figure 6 with this one:

$$\text{(LAST)}\quad \frac{C \vdash ct'_1 < ct_1}{C, H + [x : ct_1 + ct_2] \vdash \texttt{last}(x) : ct'_1 + ct_2}$$

That is, $\texttt{last}(x)$ does not depend on x during a discrete step, but it does during a continuous step.

Example $\texttt{last}(x)$ does not necessarily break causality loops:

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

If $z : \alpha_z + \beta_z$ and $y : \alpha_y + \beta_y$, then $\texttt{last}(z) : \alpha'_z + \beta_z$. We also have: $\texttt{last}(z) + y : \alpha_y + \beta$ with $\beta_y, \beta_z < \beta$. There is a causality cycle since $\beta_y, \beta_z < \beta \not< \beta_z$. This program is also rejected:

```
let hybrid f(z) = y where
  der y = 1.0 init -1.0 reset up(z) -> -1.0

let hybrid loop() = y where rec y = f(y)
```

Indeed, `f` takes the type signature $\forall \alpha_1, \alpha_2 : \{\alpha_3 < \alpha_2\}.(\alpha_1 + \alpha_2) \to \alpha_1 + \alpha_3$. The body of `loop` is then typed. Let $y : \alpha_y + \beta_y$. Thus the function application $f(y)$ is given the type $\alpha_y + \beta'_y$. The equation $y = f(y)$ is well typed if $\alpha_y + \beta'_y < \alpha_y + \beta_y$, that is, $\alpha_y < \alpha_y$ and $\beta'_y < \beta_y$. This does not hold since $\alpha_y \not< \alpha_y$.

What should the signature of a zero-crossing be? As we discussed in Section 2, numeric solvers can monitor certain signals for zero-crossings during integration. These signals need not be continuous. Nonetheless, classical methods for zero-crossing detection such as the 'Illinois' method [9] are both faster and more accurate for continuous signals. We can split $\texttt{up}(\cdot)$ into two operations according to the value of d:

$$\texttt{up}(x) = \texttt{if}\ d\ \texttt{then}\ \texttt{upd}(x)\ \texttt{else}\ \texttt{upc}(x)$$

$\texttt{up}(x)$ is the disjoint union of two event detections and the two events cannot happen at the same time:

(*a*) $\texttt{upc}(x)$ detects zero-crossings during integration. It is directly managed outside of the program by the numeric solver.

(*b*) $\texttt{upd}(x)$ detects a zero-crossing when x has a discontinuity.

The implementation of $\texttt{upd}(x)$ can be programmed in a synchronous manner. It is false at the very first instant and true when x goes from negative to positive while the current time (represented by a global signal t) does not change:

$$\texttt{upd}(x) = \texttt{false -> } (\texttt{last}(t) = t) \wedge (\texttt{last}(x) \leq 0) \wedge (x > 0)$$

The signature for zero-crossing detection is thus:

$$\begin{array}{rcl} \texttt{upc}(\cdot) & : & \forall \alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\}.\alpha_1 \to \alpha_2 \\ \texttt{upd}(\cdot) & : & \forall \alpha_1, \alpha_2, \alpha_3 : \{\alpha_3 < \alpha_2\}.(\alpha_1 + \alpha_2) \to (\alpha_1 + \alpha_3) \end{array}$$

As a consequence, $\texttt{up}(\texttt{last}(x))$ does not depend on x.

Example If $\texttt{up}(x)$ instantaneously depends on x during discrete time, then:

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

let hybrid loop() returns y where
  rec y = f(last y) init 0.0
```

`f` gets the following signature:

```
val f : { 'c < 'b }. ('a + 'b) -> ('a + 'c)
```

Avoiding unbounded cascades of discrete zero-crossings By distinguishing the detection of zero-crossings at discrete transitions from their detection during integration, it is possible to identify unbounded cascades of discrete zero-crossings. This situation occurs when a discrete transition fires a new discrete transition for the next instant, and so on. Consider replacing the signature of $\texttt{upd}(\cdot)$ by:

$$\texttt{upd}(\cdot) \quad : \quad \forall\alpha.\ \alpha + \alpha \to \alpha + \alpha$$

The following equation is now rejected:

```
der x = 1.0 init -1.0 reset upd(last x) -> -1.0
```

Indeed, $x : \alpha_1 + \alpha_2$, thus $\texttt{last}(x) : \alpha_3 + \alpha_2$, thus $\texttt{upd}(\texttt{last}(x)) : \alpha_4 + \alpha_4$ with $\alpha_2, \alpha_3 < \alpha_4$. As $\alpha_4 < \alpha_1$ but $\alpha_4 \not< \alpha_2$, the equation is invalid. Replacing $\texttt{upd}(\cdot)$ by $\texttt{upc}(\cdot)$ gives a valid equation:

```
der x = 1.0 init -1.0 reset upc(last x) -> -1.0
```

Indeed: $x : \alpha_1 + \alpha_2$, thus $\texttt{last}(x) : \alpha_3 + \alpha_2$, thus $\texttt{upc}(\texttt{last}(x)) : \alpha_4 + \alpha_5$. As $\alpha_4 < \alpha_1$ and $\alpha_5 < \alpha_2$, the program is valid.

The new signature for $\texttt{upd}(\cdot)$ means that it is no longer considered to break feedback loops as would a zero-crossing performed during integration. Consequently, defining $\texttt{up}(x)$ as the disjunction of $\texttt{upd}(x)$ and $\texttt{upc}(x)$ means that it cannot break loops either, and that it is a potential source of unbounded discrete cascades.

6 The Main Theorem

We can now state the main result of this paper. Well-typed programs have a standardization, that is, all signals are continuous during integration. This theorem requires assumptions on primitive operators and imported functions, as the following example shows.

6.1 A Nonsmooth Model

A detailed presentation of this example can be found online ♣. In a nutshell, it consists of several modules. The first two are an integrator and a time base with a parameterized initial value t_0:

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

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

Then a function producing a quasi-Dirac (Dirac with width > 0). It yields a function $\text{dirac}(d, t)$ such that $\int_{-\infty}^{+\infty} \text{dirac}(d, t)dt = 1$ for every constant $d > 0$.

```
let dirac(d, t) = 1 / 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 centered on $t = 0$. This can be achieved by integrating a pulse of magnitude 1, but of infinitesimal width. Such a pulse can be produced using a variable that is reset twice by the successive occurrences, separated by a ∂, of two zero-crossings:

```
let hybrid doublecrossing(t) = (x + 1.0) / 2.0 where
  rec init x = -1.0
  and present up(t) then next x = 1.0 else
      present up(x) then next x = -1.0 else der x = 0.0

let hybrid infinitesimal(t1,t) =
  integrator(0.0, doublecrossing(t1))
```

The first zero-crossing in `doublecrossing` occurs when `t` crosses zero and causes an immediate reset of `x` from -1 to $+1$, this in turn triggers an immediate zero-crossing on `x` and a reset of `x` back to -1. The input of the integrator is thus one for one ∂-step; the output of the integrator, initially 0, becomes ∂ at time $t_1 + \partial$.

The main program is the following, where $t_0 < t_1 < t_2$:

```
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 its standardization has a discontinuity at t_2 though no zero-crossing occurs. This is because `dirac` standardizes to a Dirac mass.

6.2 Discussion

In the previous example, the problem arises with the function `dirac`. Indeed, $\frac{d}{d^2+t^2}$ is not defined when $t = 0$ and $d = 0$. However, it is defined everywhere when $d{\neq}0$. In particular, it is defined for $d = \partial > 0$. The solution seems clear: *if a standard function $f(x)$ of a real variable x is such that $f(x_0) = \perp$, then our non-standard semantics must enforce $f(x) = \perp$ for any $x \approx x_0$.* Applying this to the function $d \mapsto \frac{d}{d^2+t^2}$ where $t = 0$ is fixed gives $\frac{\partial}{\partial^2+t^2} = \perp$. This precludes the possibility of generating a Dirac mass as above. This trick 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 $x, y \in {}^\star\mathbb{R}$,

$$x \approx y \text{ implies } f(x) \approx f(y). \tag{2}$$

Recall that the microcontinuity of f implies the uniform continuity of $st(f){:}\mathbb{R}{\mapsto}\mathbb{R}$ [19]. Denote $[t_0, t_1]_{\mathbb{T}} = \{t \in \mathbb{T} | t_0 \leq t \leq t_1\}$, 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:

$$\begin{cases} op(\bot) = \bot \\ \forall v, op(v) \neq \bot \text{ implies } op(v) \text{ finite} \\ \forall u, v, u \approx v \text{ and } op(u) \not\approx op(v) \text{ implies } op(u) = \bot \end{cases}$$

Assumption 2: Environment G is assumed to satisfy the following assumption, 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:

$$\left[\forall t \in K, \begin{cases} fst(G(f)(u)(t)) \neq \bot \text{ and} \\ snd(G(f)(u)(t)) = \texttt{false} \end{cases}\right]$$

$$\Downarrow$$

$$\left[\begin{array}{l} \forall t \in K, fst(G(f)(u)(t)) \text{ finite, and} \\ \forall t, t' \in K, t \approx t' \text{ implies} \\ fst(G(f)(u)(t)) \approx fst(G(f)(u)(t')) \end{array}\right]$$

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: $1/\epsilon = \bot$ for any infinitesimal ϵ is the only solution.
- Consequently, the function $\text{sgn}(x) = x/\sqrt{x^2}$ returning the sign of x must satisfy $\text{sgn}(\epsilon) = \text{sgn}(-\epsilon) = \text{sgn}(0) = \bot$, for any infinitesimal ϵ.

Theorem 1: Under Assumptions 1 and 2, every causally correct equation E (wrt. typing rules of Sections 5.1 or 5.2) satisfies Invariants 1 and 2 and is standardizable.

This theorem 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:

$$\left[\forall t \in K, \begin{cases} fst({}^\star[\![e]\!]^\rho_G(T)(t)) \neq \bot \text{ and} \\ snd({}^\star[\![e]\!]^\rho_G(T)(t)) = \texttt{false} \end{cases}\right]$$

$$\Downarrow$$

$$\left[\begin{array}{l} \forall t \in K, fst({}^\star[\![e]\!]^\rho_G(T)(t)) \text{ finite, and} \\ \forall t, t' \in K, t \approx t' \text{ implies} \\ fst({}^\star[\![e]\!]^\rho_G(T)(t)) \approx fst({}^\star[\![e]\!]^\rho_G(T)(t')) \end{array}\right]$$

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 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.
- ${}^{\star}[\![\texttt{last}(x)]\!]^{\rho}_{G}(T)(t)$: Two case must be distinguished, depending on the typing rules used to assert the causal correctness of the expression. If the Lustre-like causality analysis applies (Section 5.1), then $\texttt{last}(x)$ expressions are of kind `D` which is ruled out by assumption. Assume the Schizophrenic causality analysis applies (Section 5.2). Recall no zero-crossing occurs on K. Hence:

  $$ {}^{\star}[\![\texttt{last}(x)]\!]^{\rho}_{G}(T)(t) \approx {}^{\star}[\![x]\!]^{\rho}_{G}(T)(t) = \rho(x)(t), \texttt{false} $$

  Which is by assumption a defined, finite and microcontinuous function everywhere on K.

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 ${}^{\star}[\![(e_i)]\!]^{\rho}_{G}(T)(t), i = 1, 2$ are defined and microcontinuous.
- Consider the application of operator op on expression e. Two cases must be distinguished: First case, op is of kind `D`, in which case expression $op(e)$ has the same kind, which is ruled out by assumption. Second case, op is of kind `A`. 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.
- $e_1 \,\texttt{fby}\, e_2$ expressions have kind `D`. Thus they can only appear in expression of the same kind.
- In `A` or `C` expressions of the form $f(e)$, f and e can not be of the kind `D`. Therefore Assumption 2 applies to function f and the induction hypothesis applies to 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. It is, therefore, microcontinuous over K.

- ${}^\star[\![\texttt{up}(e)]\!]^\rho_G(T)(t)$ defined and produces no zero crossing for all $t \in K$ implies that it is constant and therefore microcontinuous over K.

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

Given a bounded interval $T = [t_0, t_1]_{\mathbb{T}}$, define the following nonstandard dynamical system on T:

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

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 $x(t)$ 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 wlog 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 $t_1 = t', t_2 = t$ yields the existence of $t'', t' \leq t'' \leq t$ such that $|f(t'', x(t''))|$ is also infinite. □

The corollary of this lemma, is that under Assumptions 1 and 2, the semantics of equation $\texttt{der}\, x = e$ is smooth provided that expression e is defined and triggers no zero-crossing:

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-crossing on K, then ρ' is microcontinuous on K.

Proof: Assume ρ' defined and z' false on K. Assume 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, ${}^\star[\![e]\!]^{\rho' + \rho}_G(T)(t')$ is infinite, which contradicts Lemma 1. □

Lemma 3: 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 the semantics of E, a causally correct equation of kind `C`, is defined and produces no zero-crossing on K, then it is finite and microcontinuous on K:

$$\left[\forall x, \forall t \in K, \left\{ \begin{array}{r} fst(^\star[\![E]\!]^\rho_G(T))(x)(t) \neq \bot \ and \\ snd(^\star[\![E]\!]^\rho_G(T))(t) = \texttt{false} \end{array} \right.\right]$$

$$\Downarrow$$

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

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 $fst(^\star[\![x = e]\!]^\rho_G(T)) = fst(^\star[\![e]\!]^\rho_G(T))$ is a direct consequence of Lemma 1.
- Equations `init` $x = e$ and `next` $x = e$ can not be typed with the sort `C`. Hence these cases are ruled out.
- Consider a causally correct equation of kind `C` and of the form `der` $x = e$. That $fst(^\star[\![\texttt{der}\, x = e]\!]^\rho_G(T))$ is finite and microcontinuous is a direct consequence of Corollary 1.

We now proceed with compositions of equations. Assume that the lemma is valid for equations E_1 and E_2. We shall prove 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`. Finiteness and microcontinuity of $fst(^\star[\![E]\!]^\rho_G(T)) = fst(^\star[\![E_1]\!]^\rho_G(T)) + fst(^\star[\![E_2]\!]^\rho_G(T))$ is a consequence of the induction hypothesis.
- Consider equation $E =$ `present` e `then` E_1 `else` E_2 and assume it to be causally correct. Since $snd(^\star[\![E]\!]^\rho_G(T))$ is equal to `false` at every $t \in K$, $fst(^\star[\![E]\!]^\rho_G(T)) = fst(^\star[\![E_2]\!]^\rho_G(T))$ is finite and microcontinuous by induction hypothesis.
- Consider a causally correct equation $E =$ `if` e `then` E_1 `else` E_2 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. Wlog, assume the expression evaluates to true. Hence, $fst(^\star[\![E]\!]^\rho_G(T)) = fst(^\star[\![E_1]\!]^\rho_G(T))$ is finite and microcontinuous by induction hypothesis. □

7 Discussion and Related Work

The present work continues that of Benveniste et al. [5], exploiting this time the use of non-standard semantics to establish causality relations in a hybrid program. The objective is the design and implementation of a synchronous language conservatively extended with ODEs. Synchronous hybrid programs are compiled into sequential code linked with an off-the-shelf numeric solver. The proposed causality analysis gives a sufficient condition for the program to be statically scheduled. Moreover, it ensures that all discrete changes — calling an external function performing side effects or a synchronous function — are performed only at instants of zero-crossing. This avoids using a computationally expensive *run-to-completion* mechanism to stop a cascade of discrete transitions.

The present work is related to Ptolemy [11] and the use of synchronous language concepts to define the semantics of hybrid modelers [17]. We follow the same path, replacing super-dense semantics by non-standard semantics that we found more helpful to explain causality constraints and generalize solutions adopted in synchronous compilers. The presented material has been implemented in ZÉLUS, a synchronous language extended with ODEs [7]. It is more single-minded than Ptolemy but provides a compiler producing sequential code whereas Ptolemy provides an interpreter.

Causality has been extensively studied for the synchronous languages SIGNAL [1] and ESTEREL [6]. Instead of imposing that every feedback loop crosses 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 feed with a constant input. In the present work, we adapted the simpler causality of LUSTRE based on a precedence relation in order to focus on specific issues raised when mixing discrete and continuous-time signals. Schneider et al. [2] have considered the causality problem for a hybrid extension of Quartz, a variant of ESTEREL, with ODEs. Yet, they did not address issues due to the interaction of discrete and continuous behaviors.

Regarding existing tools like SIMULINK, we believe that the data-flow interpretation of signals is useful to explain causality constraints and what the compiler should do. SIMULINK also provides a static analysis to restrict the use of the *state port* and/or reject weird interations between discrete and continuous 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. E.g., the signature $\alpha_1 \times \alpha_1 \to \alpha_1 \times \alpha_2$ states that the first output depends on the two inputs and the second output does not depend on any input.

8 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 symmetric role of a unit delay for continuous signals as the previous value is continuously close to the current value.

We introduced the construction $\texttt{last}(x)$ 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.

References

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

[2] Kerstin Bauer and Klaus Schneider. From synchronous programs to symbolic representations of hybrid systems. In *HSCC*, pages 41–50, 2010.

[3] A. Benveniste, P. Caspi, R. Lublinerman, and S. Tripakis. Actors without directors: a kahnian view of heterogeneous systems. Technical report, Verimag, Centre Équation, 38610 Gières, September 2008. Extended version of HSCC'10.

[4] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'11)*, Chicago, USA, April 2011.

[5] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Non-Standard Semantics of Hybrid Systems Modelers. *Journal of Computer and System Sciences (JCSS)*, 78(3):877–910, May 2012. Special issue in honor of Amir Pnueli.

[6] Gérard Berry. The constructive semantics of pure Esterel. 1999.

[7] Timothy Bourke and Marc Pouzet. Zélus, a Synchronous Language with ODEs. In *International Conference on Hybrid Systems: Computation and Control (HSCC 2013)*, Philadelphia, USA, April 8–11 2013. ACM.

[8] L.P. Carloni, R. Passerone, A. Pinto, and A.L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations & Trends in Electronic Design Automation*, vol. 1, 2006.

[9] Germund Dahlquist and Åke Björck. *Numerical Methods in Scientific Computing: Volume 1*. SIAM, 2008.

[10] B. Denckla and P.J. Mosterman. Stream- and state-based semantics of hierarchy in block diagrams. In *17th IFAC World Congress*, pages 7955–7960, South Korea, 2008.

[11] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proc. IEEE*, 91(1):127–144, January 2003.

[12] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc 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'12)*, Beijing, June 12-13 2012. ACM.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[14] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[15] Nicolas Halbwachs and Florence Maraninchi. On the symbolic analysis of combinatorial loops in circuits and synchronous programs. In *Euromicro 95*, Como, Italy, September 1995.

[16] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

[17] Edward A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Proc. of the 7th ACM & IEEE Int. Conf. on Embedded software (EMSOFT)*, pages 114–123, 2007.

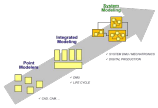
[18] Edward A. Lee and Haiyang Zheng. Operational semantics of hybrid systems. In *Hybrid Systems: Computation and Control (HSCC)*, volume 3414, Zurich, Switzerland, March, 9-11 2005. LNCS.

[19] T. Lindstrom. An invitation to non standard analysis. In N. Cutland, editor, *Nonstandard analysis and its applications*. Cambridge Univ. Press, 1988.

[20] Oded Maler, Zouar Manna, and Amir Pnueli. From Timed to Hybrid Systems. In *Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 447–484. Springer, 1992.

[21] The Mathworks, Natick, MA, U.S.A. *Simulink 7—Reference*, 7.6 edition, September 2010.

[22] The Mathworks, Natick, MA, U.S.A. *Simulink 7—User's Guide*, 7.5 edition, March 2010.



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

[24] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006.