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

Albert Benveniste
INRIA, Rennes

Timothy Bourke
INRIA and ENS/DI, Paris

Benoit Caillaud
INRIA, Rennes

Bruno Pagano
Esterel-Technologies, Toulouse

Marc Pouzet
UPMC, ENS/DI and INRIA, Paris

ABSTRACT

Explicit hybrid systems modelers like *Simulink/Stateflow* allow for programming both discrete- and continuous-time behaviors with complex interactions between them. A key issue 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. This operator is used to break causality loops and permits a uniform treatment of discrete and continuous state variables. The semantics 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 dependences between signals. In well-typed programs, signals are provably continuous during integration provided that imported external functions are continuous.

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

Categories and Subject Descriptors

I.6.2 [**Simulation and Modeling**]: Simulation Languages

Keywords

Hybrid systems; Synchronous programming languages; Type systems; Block diagrams; Static analysis.

*The full version of this paper with proofs and examples can be found at: `http://zelus.di.ens.fr/hscc2014/`. This work has been partially funded by the Sys2Soft, *Briques Génériques du Logiciel Embarqué*, *Investissements d'Avenir* French national project.

HSCC'14, April 15–17, 2014, Berlin, Germany.

http://dx.doi.org/10.1145/2562059.2562125.

1. CAUSALITY AND SCHEDULING

Tools for modeling hybrid systems [7] 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 and proving them correct [15, 9].

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 features 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 [20, 2-34] pose problems of well-definedness and compilation. They can lead to mathematically unsound models. They can 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 studied and implemented since the mid-1980s in synchronous data-flow language compilers [12, 13, 1]. The classical and simplest solution is to reject cycles (feed-back 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:

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

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. 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 [11] 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;
```

[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 sometimes written as 0 `fby`$\cdot$ ('0 followed by'), or in Simulink: $\frac{1}{z}$.

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 $k(t) = y(t) + 1$.[5] This program is causal simply because it is possible to generate a sequential function $\mathit{derivative}(y) = \texttt{let}\ z = 10 - 0.1 * y\ \texttt{in}\ z$ returning the current derivative of y and initial value 4 for y so that a numeric solver [8] 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 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 ∂ [4]. 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:

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

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:

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

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 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 [4], it is infinitely close to the previous value of y, and written $ly(n) \approx y(n-1)$. When y is defined by its derivative, `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 one above ♣.[7] According to the documentation [19, 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 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 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 x - 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:

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

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

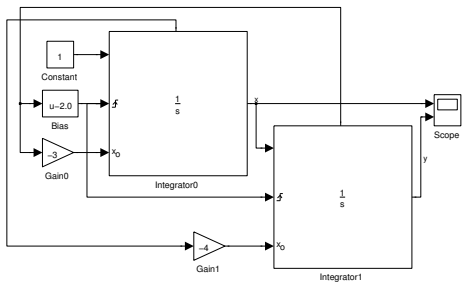
Yet, can you guess the behavior of the model and explain why the trajectories computed by SIMULINK are wrong?

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 $(= 2 \cdot -3)$

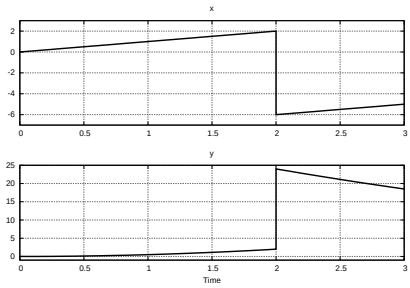
[5] `der` $y = e$ `init` v_0 stands for $y = \frac{1}{s}(e)$ inititialized to v_0 in SIMULINK.

[6] The ♣'s link to `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:
$(y, ly) = \frac{1}{s}(0.0, \texttt{up}(ly - 1.0), 1.0)$.



(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)

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 [19, 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.[8] 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. Providing a means to statically detect and explain causality issues in a hybrid model is a key motivation of this paper.

Any loop in Simulink, whether of discrete- or continuous-time signals, can be broken by inserting the so-called memory block [19, 2-831].[9] 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 [3].[10] 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 using the unsafe operator $\texttt{mem}(x)$, we could better 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 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`. 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 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.

Contribution and organization of the paper: This paper presents the causality problem for a core language

[8] The same issue exists in release R2014a.

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

[10] 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."

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 [4] 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 [21]. 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 [6], 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 LANGUAGE + 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 includes data-flow equations, ordinary differential equations and control structures. Its syntax is:

$$d ::= \texttt{let}\, x = e \mid \texttt{let}\, k\, f(p) = e\, \texttt{where}\, E \mid d; d$$

$$e ::= x \mid v \mid op(e) \mid e\, \texttt{fby}\, e \mid \texttt{last}(x) \mid f(e) \mid (e, e) \mid \texttt{up}(e)$$

$$p ::= (p, p) \mid x$$

$$E ::= () \mid x = e \mid \texttt{init}\, x = e \mid \texttt{next}\, x = e \mid \texttt{der}\, x = e \mid E\ \texttt{and}\ E \mid \texttt{local}\, x\, \texttt{in}\, E \mid \texttt{if}\, e\, \texttt{then}\, E\, \texttt{else}\, E \mid \texttt{present}\, e\, \texttt{then}\, E\, \texttt{else}\, E$$

$$k ::= \texttt{D} \mid \texttt{C} \mid \texttt{A}$$

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, 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 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 ($\texttt{init}\ x = e$); 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: every variable has a unique definition at every instant.

3. NON-STANDARD SEMANTICS

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 (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 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 will thus be 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. $S(V) = \mathbb{T} \mapsto V_\perp$ 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. *The clock of a signal* x *is* $clock(x) = \{t \in \mathbb{T} \mid x(t) \neq \bot\}$.

Sampling: Let $\texttt{bool} = \{\texttt{false}, \texttt{true}\}$ and $x : T \mapsto \texttt{bool}_\bot$. 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 $x : T \mapsto {}^\star\mathbb{R}_\bot$ is $up(x) : T \mapsto \texttt{bool}_\bot$. 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 $t \notin T$, $up(x)(T)(t) = \bot$. In the definition below $<$ is the total order on ${}^\star\mathbb{R}$.

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

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. ${}^\star V_\bot = {}^\star V + \{\bot\}$ with $\bot$ 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 $\rho_2(x)(t) = \bot$, $\rho_2(x)(t)$ if $\rho_1(x)(t) = \bot$, and $\bot$ 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 $t \in T$, $\rho(x)(t) = \rho_1(x)(t)$ 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 $x \in Dom(\rho_2)$; finally, $\rho(x)(t) = \rho(x)({}^\bullet clock(x)(t))$ otherwise.

Expressions: Expressions are interpreted as signals and node definitions as functions from signals to signals. For expressions, we define ${}^\star[\![e]\!]^\rho_G(T)(t)$ to give at every instant $t \in T$ 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

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

Figure 3: The non-standard semantics of expressions

$op(e)$ 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 does. The initial value of a delay e_1 `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 `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 does. Finally, the semantics of $\texttt{up}(e)$ is given by operator $up(.)$, 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 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 *or* $z_2 : T \mapsto \texttt{bool}_\bot$ and $\forall t \in T.(z_1 \text{ or } z_2)(t) = z_1(t) \vee z_2(t)$ if $z_1(t) \neq \bot$ and $z_2(t) \neq \bot$, and otherwise, $(z_1 \text{ or } z_2)(t) = \bot$.

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

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

Figure 4: The non-standard semantics of equations

$x(t) \neq y(t)$ 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* [14].

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:

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

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

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

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

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

and with

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

Yet, 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[11]

```
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 [18], 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* [18, 15] 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:

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

that is, the set of all instants infinitely close to t. T is totally ordered and hence so is $set(T)(t)$. 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

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

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

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

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

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

(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

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

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

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

which corresponds to a Zeno behavior.

Our approach differs slightly from [15], 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. 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 (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 all discontinuities are aligned on zero-crossings, that is, 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.

4. A LUSTRE-LIKE CAUSALITY

Programs are statically typed. We adopt, for our language, the type system presented in [3]. Well-typed programs may still exhibit causality issues, that is, the definition of a signal at instant t may instantaneously depend on itself. 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 [12], LUCID SYNCHRONE [22] and SCADE 6.[12] 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.

The analysis gives sufficient conditions for invariants 1 and 2. We adopt the convention quoted below [3, 4]. 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`(.).

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 [1, 4] 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'.

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

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:

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

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

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

$$\text{(PAIR)}\quad \frac{C \vdash ct_1 < ct_1' \qquad C \vdash ct_2 < ct_2'}{C \vdash ct_1 \times ct_2 < ct_1' \times ct_2'}$$

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

Figure 5: Constraints between types

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 α_1 and also later, within the execution of the same reaction, at α_2. $<$ 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.

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

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.

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

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

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 $C \mid G, H \vdash x : \alpha$, $C \mid G, H \vdash y : \alpha$. Thus $C \mid G, H \vdash x + y : \alpha$. $\texttt{pre}(x)$ does not depend on x. For $\texttt{up}(x)$, two policies can be considered:

$$\texttt{up}(\cdot) : \forall \alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\}.\alpha_1 \xrightarrow{\texttt{C}} \alpha_2 \quad \texttt{up}(\cdot) : \forall \alpha_1 : \alpha_1 \xrightarrow{\texttt{C}} \alpha_1$$

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

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:

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

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

Rule (APP). An application $f(e)$ has 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). $\texttt{last}(x)$ is the previous value of x. In this system, we only allow $\texttt{last}(x)$ to appear during a discrete step (of kind $\texttt{D}$).

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 (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.

Rules (PRESENT) and (IF). The present statement returns an environment $H_1 * H_2$. The first handler is activated during discrete steps and the second one has kind $\texttt{C}$. The rule for conditionals is the same except that the handlers and condition must all be of kind k.

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.

Example: The following program is a classic synchronous (thus discrete-time) program written in the concrete syntax of ZÉLUS. 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 fby (x + x' * step)

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

The causality signatures are:

```
val integr : {'a < 'b}.'a * 'b -C-> 'a
val heat   : {'a < 'b}.'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 following two are correct, `der` playing the role of a delay:

$$\text{(VAR)}\ \frac{}{C \mid G, H + x : ct \vdash_k x : ct} \qquad \text{(CONST)}\ \frac{}{C \mid G, H \vdash_k v : ct} \qquad \text{(APP)}\ \frac{C, ct_1 \xrightarrow{k} ct_2 \in Inst(G(f)) \qquad C \mid G, H \vdash_k e : ct_1}{C \mid G, H \vdash_k f(e) : ct_2}$$

$$\text{(LAST)}\ \frac{C \vdash ct_2 < ct_1}{C \mid G, H + x : ct_1 \vdash_{\mathtt{D}} \mathtt{last}(x) : ct_2} \qquad \text{(EQ)}\ \frac{C \mid G, H \vdash_k x : ct \qquad C \mid G, H \vdash_k e : ct}{C \mid G, H \vdash_k x = e : [ct/x]} \qquad \text{(DER)}\ \frac{C \mid G, H \vdash_{\mathtt{C}} e : ct_1 \qquad C \vdash ct_2 < ct_1}{C \mid G, H \vdash_{\mathtt{C}} \mathtt{der}\, x = e : [ct_2/x]}$$

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

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

$$\text{(AND)}\ \frac{C \mid G, H \vdash_k E_1 : H_1 \qquad C \mid G, H \vdash_k E_2 : H_2}{C \mid G, H \vdash_k E_1\ \mathtt{and}\ E_2 : H_1 * H_2} \qquad \text{(LOCAL)}\ \frac{C \mid G, H + [x : ct_1] \vdash_k E : H' + [x : ct_2] \qquad C \vdash ct_2 < ct_1}{C \mid G, H \vdash_k \mathtt{local}\ x\ \mathtt{in}\ E : H'}$$

$$\text{(PAIR)}\ \frac{\forall i \in \{1,2\} :\ C \mid G, H \vdash_k e_i : ct_i}{C \mid G, H \vdash_k (e_1, e_2) : ct_1 \times ct_2} \qquad \text{(DEF)}\ \frac{C \mid G, H \vdash_k E : H' \qquad C \vdash H' < H \qquad C \mid G, H \vdash_k p : ct_1 \qquad C \mid G, H \vdash_k e : ct_2}{\vdash \mathtt{let}\ k\ f(p) = e\ \mathtt{where}\ E : [Gen(C)(ct_1 \xrightarrow{k} ct_2)/f]}$$

Figure 6: A Lustre-like Causality Analysis

```
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
```

In the present system, $\mathtt{last}(x)$ is restricted to 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
```

If up(.) is considered to instantaneously depend on its input, loop is 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 has signature $\forall\alpha.\alpha \to \alpha$. For loop to be well-typed, we would need to be able to state an equation $\alpha_y < \alpha_y$ where $y : \alpha_y$.

Type simplication: The ZÉLUS compiler implements a simplification algorithm to eliminate useless constraints. It follows the algorithm [23] to partition type variables according to *Input-Output* (IO) relations. Moreover, as causality analysis is performed after typing, some relations can be removed. E.g., the actual signature of the unit delay is simply:

$$\mathtt{pre}(\cdot) \quad : \quad \forall\alpha_1, \alpha_2 : \alpha_1 \xrightarrow{\mathtt{D}} \alpha_2$$

The State Port: The present causality analysis restricts the use of $\mathtt{last}(x)$ to appear only under a discrete context. An extension is to allow $\mathtt{last}(x)$ to appear in a continuous context provided x is a continuous state variable, i.e., it is defined by an equation $\mathtt{der}\, x = e$. Indeed, during integration $\mathtt{last}(x)$ and x are infinitely close to each other (${}^\star x(n-1) \approx {}^\star x(n)$). The ZÉLUS compiler implements this minor extension.

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 Model ♣: It comprises several modules (written in ZÉLUS syntax). 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 a width strictly greater than 0). It yields a function $\mathrm{dirac}(d,t)$ such that $\int_{-\infty}^{+\infty} \mathrm{dirac}(d,t)dt = 1$ 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 $\mathrm{dirac}(d,t)$ 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 do next x = 1.0 done else
      present up(x) then do next x = -1.0 done
                    else do der x = 0.0 done

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.

Discussion: In the previous example, the problem arises with the function `dirac`, that is not defined when $t = 0$ and $d = 0$. However, it is defined everywhere when $d \neq 0$. In particular, it is defined for $d = \partial$. The solution seems clear: *if a standard function $f(x)$ of a real variable x is such that $f(x_0) = \bot$, then the semantics must enforce $f(x) = \bot$ 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 $\frac{\partial}{\partial^2+t^2} = \bot$. 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 $st(f)$:$\mathbb{R}\mapsto\mathbb{R}$ [17]. 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:*

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

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:*

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

Assumption 1 has several implications on the definitions of the usual operators. For the square root function: $\sqrt{\epsilon} = \sqrt{-\epsilon} = \sqrt{0}$, for all $\epsilon \approx 0$, 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.

Theorem 1. *Under Assumptions 1 and 2, the semantics of every causally correct equation E (wrt. 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:*

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

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

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

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.*

The corollary of this lemma, is that under Assumptions 1 and 2, the semantics of `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.*

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:*

$$\left[\forall x, \forall t \in K, \begin{cases} fst({}^\star[\![E]\!]^\rho_G(T))(x)(t) \neq \bot \text{ and} \\ snd({}^\star[\![E]\!]^\rho_G(T))(t) = \texttt{false} \end{cases}\right]$$
$$\Downarrow$$
$$\left[\begin{array}{l} \forall x, (\forall t \in K, fst({}^\star[\![E]\!]^\rho_G(T))(x)(t) \textit{ finite, and} \\ \forall t, t' \in K, t \approx t' \textit{ implies} \\ fst({}^\star[\![E]\!]^\rho_G(T))(x)(t) \approx fst({}^\star[\![E]\!]^\rho_G(T))(x)(t')) \end{array}\right]$$

6. DISCUSSION AND RELATED WORK

The present work continues that of Benveniste et al. [4], 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.

The present work is related to Ptolemy [10] and the use of synchronous language concepts to define the semantics of hybrid modelers [16]. 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 is implemented in ZÉLUS, a synchronous language extended with ODEs [6]. It is more single-minded than Ptolemy but ZÉLUS programs are turned into sequential code whereas Ptolemy only provides an interpreter.

Causality has been extensively studied for the synchronous languages SIGNAL [1] and ESTEREL [5]. 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 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 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. But, they did not address issues due to 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.

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 $\texttt{last}(x)$ 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.

8. 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] K. Bauer and K. Schneider. From synchronous programs to symbolic representations of hybrid systems. In *HSCC*, pages 41–50, 2010.

[3] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *LCTES*, Chicago, USA, Apr. 2011. ACM.

[4] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. Non-Standard Semantics of Hybrid Systems Modelers. *Journal of Computer and System Sciences (JCSS)*, 78(3):877–910, May 2012.

[5] G. Berry. The constructive semantics of pure Esterel. 1999.

[6] T. Bourke and M. Pouzet. Zélus, a Synchronous Language with ODEs. In *HSCC*, Philadelphia, USA, April 8–11 2013. ACM.

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

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

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

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

[11] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A Modular Memory Optimization for Synchronous Data-Flow Languages. Application to Arrays in a Lustre Compiler. In *LCTES*, Beijing, June 12-13 2012.

[12] 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.

[13] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *PLILP*, LNCS, Passau (Germany), August 1991.

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

[15] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In *HSCC*, Zurich, Switzerland, Mar. 2005.

[16] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, 2007.

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

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

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

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

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

[22] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, Apr. 2006.

[23] M. Pouzet and P. Raymond. Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation. In *EMSOFT*, Grenoble, France, Oct. 2009. ACM.