Divide and Recycle: Types and Compilation for a Hybrid Synchronous Language *

Albert Benveniste Timothy Bourke
Benoît Caillaud
INRIA Rennes
Firstname.Name@irisa.fr

Marc Pouzet
Univ. Pierre et Marie Curie
LIENS, École normale supérieure
Marc.Pouzet@ens.fr

Abstract

Hybrid modelers such as SIMULINK have become corner stones of embedded systems development. They allow both *discrete* controllers and their *continuous* environments to be expressed *in a single language*. Despite the availability of such tools, there remain a number of issues related to the lack of reproducibility of simulations and to the separation of the continuous part, which has to be exercised by a numerical solver, from the discrete part, which must be guaranteed not to evolve during a step.

Starting from a minimal, yet full-featured, LUSTRE-like synchronous language, this paper presents a conservative extension where data-flow equations can be mixed with ordinary differential equations (ODEs) with possible reset. A type system is proposed to statically distinguish discrete computations from continuous ones and to ensure that signals are used in their proper domains. We propose a semantics based on *non-standard analysis* which gives a synchronous interpretation to the whole language, clarifies the discrete/continuous interaction and the treatment of zero-crossings, and also allows the correctness of the type system to be established.

The extended data-flow language is realized through a source-to-source transformation into a synchronous subset, which can then be compiled using existing tools into routines that are both efficient and bounded in their use of memory. These routines are orchestrated with a single off-the-shelf numerical solver using a simple but precise algorithm which treats causally-related cascades of zero-crossings. We have validated the viability of the approach through experiments with the SUNDIALS library.

Categories and Subject Descriptors C.3 [*SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS*]: Real-time and embedded systems; D.3.2 [*Language Classifications*]: Data-flow languages; D.3.4 [*Processors*]: Code generation, Compilers

General Terms Algorithms, Languages, Theory

Keywords Real-time systems; Hybrid systems; Synchronous languages; Block-diagrams; Compilation; Semantics; Type systems

* This work was supported by the SYNCHRONICS large scale initiative of INRIA.

LCTES'11, April 11–14, 2011, Chicago, Illinois, USA.

1. Introduction

Hybrid system modelers have become, over the last two decades, corner stones of complex embedded system development, with embedded systems involving not only control components or software, but also physical devices. Hybrid system modelers mix discrete time reactive (or dynamical) systems with continuous time ones. Systems like SIMULINK[1] treat explicit (or causal) models made of Ordinary Differential Equations (ODEs), while others, like MODELICA[2] can manage more general implicit (or acausal) models defined by Differential Algebraic Equations (DAEs). In this paper, we focus on modelers for explicit hybrid systems and we refer the reader to [7] for an overview of tools related to hybrid systems modeling and analysis.

Rather than address the formal verification of hybrid systems, which has been studied extensively, this paper treats hybrid system modelers from a programming language perspective, focusing on their typing, semantics, and compilation. Hybrid modelers raise several issues related to the lack of reproducibility of simulations (sensitivity to simulation parameters and to the choice of simulation engine), the interaction between the discrete and the continuous parts, the way cascades of causally-related zero-crossings are handled, and also to the inefficiency of the generated code. Some of these issues are unavoidable because of approximations made by the numerical solver or simply because signals are mathematically not integrable. Others arise from the absence of a strong typing discipline to properly separate the continuous part, which has to be exercised by a numerical solver, from the discrete part, which must be guaranteed not to evolve during a step. Finally, the possible interaction of the code with a numerical variable-step solver usually requires a specific calling convention for every block to separate the computation of outputs from the modification of internal states. In contrast, the compiler of a synchronous language like LUSTRE [10] or SIGNAL [3] generates far better code with, in particular, a single step function that modifies the state in-place [4].

This paper responds to some of the weaknesses described above by considering a hybrid extension of a synchronous language. Our approach involves *dividing* such programs into continuous and discrete parts using a novel type system, and then *recycling* existing tools, viz. compilers and numerical solvers, to execute them.

Contribution and organization of the paper: Starting with a minimal, yet full featured, LUSTRE-like synchronous language, this paper presents a conservative extension that allows data-flow equations to be composed with ordinary differential equations (ODEs) with possible reset.

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

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

A type system is presented that distinguishes, at compile time, *discrete* from *continuous* computations and also ensures that signals are used in their proper domains. The main intuition is that computations are discrete only when activated on a zero-crossing event. They otherwise describe continuous behaviors that must be approximated by a numerical solver. The type system is conservative with respect to that of the basic synchronous language, that is, synchronous programs are typed *discrete*. The synchronous (discrete) subset is compiled in the usual way.

After presenting the type system, we propose a semantics based on non-standard analysis; after an original idea due to Bliudze and Krob [5]. This gives a synchronous interpretation of the whole system where the base clock is both discrete and an infinite sequence of infinitesimals. This interpretation clarifies the treatment of zero-crossings in modelers and allows the correctness of the type system to be established: in particular, the semantics of well-typed programs do not depend on the choice of infinitesimal.

Rather than develop a new compilation method for the extended language, we give a source-to-source transformation that translates from the full language into a synchronous subset. The result can be compiled by an existing synchronous language compiler.

Finally, we show how to interface a compiled routine with a single off-the-shelf numerical solver. A program is simulated by cycling between continuous phases where the state is approximated by the numerical solver, and discrete phases where the consequences of zero-crossing events are computed. Our prototype system is implemented using the SUNDIALS CVODE library [11]. The clear separation of compilation and simulation distinguishes it from other more monolithic tools.

The need for types in hybrid modelers are explained in Section 2, which also shows how an adequate semantic domain can clarify the treatment of zero-crossings, and presents an example to summarize our approach. The syntax, type system, and semantics of the new language are presented in Section 3. The source-to-source transformation is presented in Section 4, and in Section 5 we show how its results, in a certain normal form, can be executed by an interaction between continuous steps, computed in a numerical solver, and discrete steps. Related work is discussed in Section 6, before the paper is concluded in Section 7.

2. Overview

Hybrid modelers allow the composition of parallel subsystems that may work in discrete or continuous time. The following examples use a notation that is formally defined in Section 3: E_1 `and` E_2 is the parallel composition of equations E_1 and E_2, `pre f` is the non-initialized unit delay, and `0.0 -> pre f` is a unit delay with an initial value of 0.0. The meaning of parallel composition is clear when the subsystems are homogeneous, as illustrated by the following two examples (`*.` is floating point multiplication):

```
f = 0.0 -> pre f +. s  and  s = 0.2 *. (x -. pre f)
```

and the initial value problem:

```
der(y') = -9.81 init 0.0  and  der(y) = y' init 10.0
```

The first program can be written in any synchronous language, e.g. LUSTRE. The semantics of f and s are infinite sequences:

$$\forall n \in \mathbb{N}^*, f_n = f_{n-1} + s_n \text{ and } f_0 = 0$$

$$\forall n \in \mathbb{N}, s_n = 0.2 * (x_n - f_{n-1})$$

The second program can be written in any hybrid modeler, e.g. SIMULINK, and its semantics is:

$$\forall t \in \mathbb{R}_+, y'(t) = 0.0 + \int_0^t -9.81\, dt = -9.81\, t$$

$$\forall t \in \mathbb{R}_+, y(t) = 10.0 + \int_0^t y'(t)\, dt = 10.0 - 9.81 \int_0^t t\, dt$$

In both examples, the equations share the same time scale so their parallel composition is precisely defined. But what would it mean to combine two equations with different time domains as in:[3]

```
der(time) = 1.0 init 0.0  and  x = 0.0 fby x +. time
```

or:

```
x = 0.0 fby x +. 1.0  and  der(y) = x init 0.0
```

The two programs cannot be given a clear semantics: in the first one, `time` is a continuous signal such that $\forall t \in \mathbb{R}_+, \texttt{time}(t) = t$ whereas x is discrete. It would be tempting to define the first equation as $\forall n \in \mathbb{N}^*, x_n = x_{n-1} + \texttt{time}(n)$, $x_0 = \texttt{time}(0)$ and the second as $\forall n \in \mathbb{N}^*, x_n = x_{n-1} + 1.0$, $x_0 = 1.0$, and, $\forall t \in \mathbb{R}_+, y(t) = 0.0 + \int_0^t x(t)\, dt$, i.e. to treat $x(t)$ as a piecewise constant function from $\mathbb{R}_+$ to $\mathbb{R}_+$ with $\forall t \in \mathbb{R}_+, x(t) = x_{\lfloor t \rfloor}$, but this would, in both cases, be a mistake, as x is a discrete signal which can be represented as a sequence of values that are not related to any absolute time. That is, x_3 does not define the value produced at absolute time $t = 3$ but only the third value in a sequence. Conversely, for the continuous signal y, it would be meaningless to define its third value. Thus, such compositions should definitely be rejected since there is nothing that defines how discrete (logical) instants are related to (absolute) continuous time. Indeed, the two programs, after being suitably rewritten (see Appendix A), are rejected by SIMULINK, provided it is configured to stop on unspecified data transfers between rates.[4] Explicit conversions from continuous to discrete and back should be added.

An even more unusual behavior can be observed if an integrator is reset. Consider, for example:

```
    der(p) = 1.0 init 0.0
             reset 0.0 every up(p -. 1.0)
and x = 0.0 fby x +. p
and der(time) = 1.0 init 0.0
and z = up(sin (freq *. time))
```

The signal `p` is initialized with value 0 and slope 1, it is reset every time `p` crosses 1 (from negative to positive), and `x` sums the values of `p`. In parallel are two unrelated equations (defining `time` and `z`). For all $t \in \mathbb{R}_+, \texttt{time}(t) = t$ and `z` is true when $\sin(freq \cdot time)$ crosses 0, where *freq* is a constant.[5] This program should be rejected because its behavior depends on the step size chosen by the solver which may be variable and which can be influenced by unrelated blocks running in parallel. No implicit conversion is really meaningful. In the same way, it should not be possible to write an ODE such as $\texttt{der}(y) = x$ `init` 0 in a block only activated at discrete instants.[6] The problem with all of these programs is that they exhibit typing issues: an expression $0\,\texttt{fby}\,x$, referring to the previous value of x, is expected to run at discrete instants, whereas the value of an integral should be computed over continuous instants. So, what, in fact, is a good definition of a

[3] $x\,\texttt{fby}\,y = x\,\texttt{->}\,\texttt{pre}(y)$

[4] The default behavior of SIMULINK is liberal and the two programs are accepted with implicit conversions, that is, $\forall t \in \mathbb{R}_+, x(t) = x_{\lfloor t \rfloor}$. In this case, a warning is printed: *Warning: Illegal rate transition found involving Unit Delay 'ExampleRateTransitionError/Unit Delay'. When using it to transition rates, the input must be connected to the slow sample time and the output must be connected to the fast sample time. The Unit Delay must have a sample time equal to the slow sample time and the slow sample time must be a multiple of the fast sample time. Also, the sample times of all destinations must be the same value. Warning: Using a default value of 0.2 for maximum step size. The simulation step size will be equal to or less than this value.* (SIMULINK version 7.7.0.471, R2008b).

[5] This is evident in the SIMULINK program of Appendix A when comparing values of `p` for different values of `freq` in the unrelated sine wave block.

[6] This ODE is written $y = \frac{1}{s}(x)$ in SIMULINK.

discrete signal? A signal is *discrete* if it is activated on a *discrete clock*, that is so defined:

> 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*.

The first contribution of the paper is a type system for a hybrid synchronous language. The principle is to give a kind $k \in \{\texttt{A}, \texttt{D}, \texttt{C}\}$ to every expression. An expression has kind `D` when it must be activated on a *discrete* clock, `C` when it must be activated on a *continuous* clock, and `A` when it can be activated on any clock. For example, the expressions v `fby` e and `pre`(e) are of kind `D`, and, more generally, so is any (non-combinatorial) LUSTRE program. An expression with kind `D` must be activated on a zero-crossing condition `up`(e). On the contrary, an ODE `der`$(x) = e$ `init` e_0 is of kind `C`. It must be forbidden to put it under a zero-crossing, that is, on a discrete clock, since it must be activated continuously. For example, the following synchronous function is well typed.[7]

```
let node counter(top, tick) = o where
      o = if top then i else 0 fby o + 1
  and i = if tick then 1 else 0
```

Its type signature is `bool` $\times$ `bool` $\xrightarrow{\texttt{D}}$ `int` because the equations defining `o` and `i` are both of kind `D`. The counter can now be connected to continuous time, by, for example, activating it every ten seconds. This is done by defining a continuous signal `time` with slope $1/10$ that is reset whenever `time -. 1.0` crosses zero.[8]

```
let hybrid counter_ten(top, tick) = o where
      der(time) = 1.0 /. 10.0 init 0.0
                  reset 0.0 every zero
  and zero = up(time -. 1.0)
  and o = counter(top, tick) every zero init 0
```

Its type signature is `bool` $\times$ `bool` $\xrightarrow{\texttt{C}}$ `int`. The construction `counter(top, tick) every zero init 0` is an *activation condition*:[9] the function `counter(top, tick)` is called when `zero` is true; otherwise, `o` keeps its previous value. The initial value is 0. As the three equations have kind `C` so does their composition.

Consider another example; a bouncing ball with initial position (x_0, y_0) and initial speed (x'_0, y'_0), written:

```
let hybrid bouncing(x0,y0,x'0,y'0) = (x,y) where
     der(x)  = x' init x0
 and der(x') = 0.0 init x'0
 and der(y)  = y' init y0
 and der(y') = -. g init y'0
               reset -. 0.9 *. last y' every up(-. y)
```

Every time the ball hits the ground (when $-y$ becomes or exceeds zero), its initial speed is reset to `0.9 *. last y'`. The expression `last y'` yields the left limit of signal `y'`. The type signature of this function is: `float` $\times$ `float` $\times$ `float` $\times$ `float` $\xrightarrow{\texttt{C}}$ `float` $\times$ `float`.

After typing such programs, we turn now to considering their compilation and execution, including the incorporation of a numerical solver to approximate continuous computations. Since the language extends an existing synchronous language with continuous time, the interesting question is: can we dispatch the compilation of the discrete part—a well-known problem solved by existing synchronous compilers—and focus solely on the continuous part? In this paper, we present a source-to-source transformation into the discrete subset that removes all continuous computations, i.e., all ODE and zero-crossing expressions. The details are given in Section 4, but as an example, consider the translation of `counter_ten`:

```
let node counter_ten([z], [ltime], (top, tick))
                    = (o, [upz], [time], [dtime])
  where
      time = 0.0 every z default ltime init 0.0
  and dtime = 1.0 /. 10.0
  and o = counter(top, tick) when z init 0
  and upz = time -. 1.0
```

The extra inputs and outputs are for communicating with the numerical solver, which calls (a compiled version of) this function directly. There are two extra inputs: a vector of boolean conditions (`[z]`) for signaling zero-crossings and a vector of continuous state variables (`[ltime]`). There are three extra outputs: a vector of zero-crossing expression values (`[upz]`), a vector of state values (`[time]`), and a vector of derivatives (`[dtime]`). Notice that the state of the function (i.e., the value of delay operators) only changes when the zero-crossing condition is true; otherwise, the function is effectively combinatorial. The discrete function `counter` is left unchanged by the translation. Now, ignoring details of syntax, the function `counter_ten` can be processed by any synchronous compiler, and the generated transition function will satisfy the invariant:

> The discrete state, i.e., the values of delays, will not change if all of the zero-crossing conditions are false.

The numerical solver is responsible for computing the continuous states (`[ltime]`, in the example) and detecting zero-crossing events (`[upz]`). Execution alternates between two phases: (1) a continuous phase where the solver repeatedly computes new values for continuous states and monitors the values of zero-crossing expressions, and (2) a discrete phase where the effect of zero-crossing events is computed. The details are presented in Section 5.

3. A Hybrid Synchronous Language

In this section, we define a single assignment hybrid language, a type system to distinguish discrete computations from continuous ones, and a semantics based on non-standard analysis.

3.1 A single assignment language

We consider the following programming language kernel.

$$
\begin{array}{lcl}
d & ::= & \texttt{let}\ k\ f(p) = e \mid d; d \\
e & ::= & x \mid v \mid op(e) \mid e\ \texttt{fby}\ e \mid \texttt{last}(x) \\
 & & \mid \texttt{up}(e) \mid f(e) \mid (e, e) \mid \texttt{let}\ E\ \texttt{in}\ e \\
p & ::= & (p, p) \mid x \\
h & ::= & e\ \texttt{every}\ e \mid \ldots \mid e\ \texttt{every}\ e \\
E & ::= & x = e \mid \texttt{der}(x) = e\ \texttt{init}\ e\ \texttt{reset}\ h \\
 & & \mid x = h\ \texttt{default}\ e\ \texttt{init}\ e \\
 & & \mid x = h\ \texttt{init}\ e \mid E\ \texttt{and}\ E
\end{array}
$$

A program is a sequence of global declarations (d) of n-ary functions over signals: `let` k $f(p) = e$, where k indicates the kind of the function (in the introductory examples, we wrote `node` for $k = \texttt{D}$ and `hybrid` for $k = \texttt{C}$).

Expressions are composed of variables (x), immediate values (v), the application of an external primitive ($op(e)$), an initialized delay (e `fby` e), the left limit of a signal (`last`(x)), a zero-crossing (`up`(e)), the application of a function to its argument ($f(e)$), a pair (e, e) and a local declaration (`let` E `in` e), which returns the value

[7] It counts the number of `ticks` between two `tops`.

[8] There are obviously far more efficient ways of modeling dedicated periodic clocks in hybrid modelers. But this example does show that timers that trigger blocks can be treated as a particular form of zero-crossing.

[9] Activation conditions appear frequently in SCADE, for example. The equivalent in SIMULINK is a block triggered on a zero-crossing event.

of e where variables used in the expression e can be defined in the set of equations in E.

Patterns, p, comprise variables (x) and pairs of patterns (p, p). Other tuples are a shorthand for nested pairs.

A reset handler, denoted h, is a list of pairs of expressions and zero-crossings (e_1 `every` z_1 | ... | e_n `every` z_n). The zero-crossing expressions $z_1, ..., z_n$ are considered sequentially, that is, if several are enabled at the same instant, only the first in the order has any effect. When z_i is the first enabled zero-crossing, then h will take the corresponding value $Xcrossing(e_i)$, and if no zero-crossings are enabled, it will take the value $NoEvent$.

A set of equations is denoted E, its elements may define a signal x with value e (equation $x = e$); a signal x with derivative e, initial value e_0, and reset according to a handler h (equation $\texttt{der}(x) = e$ `init` e_0 `reset` h),[10] a piecewise-constant signal which evolves at discrete steps according to a handler h (equation $x = h$ `init` e_0), a similar signal (equation $x = h$ `default` e `init` e_0) that also changes at discrete steps but otherwise falls back to a default value of e rather than keeping its previous value, or a parallel composition of two sets of equations. In equations of the form $x = h$ `init` e_0, e_0 defines the (constant) value of x at every instant strictly less than the first occurrence of a zero-crossing in h, otherwise x keeps its previous value unless one of the associated zero-crossings occurs.

Notation shortcuts: We provide the following shortcuts.

1. The conditional operation `if` e_0 `then` e_1 `else` e_2 is a particular case of an imported operator of arity 3.
2. The equation $\texttt{der}(x) = e$ `init` e_0 stands for the equation $\texttt{der}(x) = e$ `init` e_0 `reset` e_0 `every` $\texttt{up}(0.0)$, where resets never occur.
3. The equation $\texttt{der}(x) = e$ `init` e_0 `reset` h can also be written $\texttt{init}(x) = e_0$ `and` $\texttt{der}(x) = e$ `reset` h.

3.2 Static Typing

We argued in the introduction for the need to statically distinguish *discrete* from *continuous* computations. Deciding statically whether a signal is discrete or not is out of reach for a realistic programming language. We thus take a more pragmatic point of view: a signal is typed *discrete* if it is activated on a zero-crossing event, and otherwise it is typed *continuous*.

The intuition behind the type system is to give a type of the form $t_1 \xrightarrow{k} t_2$ to a function f where k is a kind with three possible values. If $k = \texttt{C}$, f can only be used in a block activated on a continuous clock. If $k = \texttt{D}$, f must be activated by a discrete clock. If $k = \texttt{A}$, then f can be used in expressions of any kind, that is, f is a combinatorial function. Kinds can be compared such that for all k, $k \subseteq k$ and $\texttt{A} \subseteq k$. The type language is:

$$\begin{array}{lcl} \sigma & ::= & \forall \beta_1, ..., \beta_n.t \xrightarrow{k} t \\ t & ::= & t \times t \mid \beta \mid bt \\ k & ::= & \texttt{D} \mid \texttt{C} \mid \texttt{A} \\ bt & ::= & \texttt{float} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{zero} \end{array}$$

where σ defines types schemes and $\beta_1, ..., \beta_n$ are type variables. A type t can be a pair ($t \times t$), a type variable (β) or a base type (bt), `zero` stands for the type of a zero-crossing condition whose only value constructor is $\texttt{up}(.)$.

Typing must keep track of both the types of defined nodes (signal functions) and local signals. A global environment G assigns type schemes (σ) to global identifiers and an environment H assigns types to variables. We write $\texttt{last}(x) : t$ when x is of type t and x has an initialized left limit, that is, when the expression $\texttt{last}(x)$ is permitted. If H_1 and H_2 are two environments, $H_1 + H_2$ is the union of the two, provided their domains are disjoint.

$$\begin{array}{lcl} G & ::= & [f_1 : \sigma_1; \ldots; f_n : \sigma_n] \\ H & ::= & [\,] \mid H, x : t \mid H, \texttt{last}(x) : t \end{array}$$

Typing is defined by four predicates:

$$\begin{array}{ll} \text{(TYP-EXP)} & \text{(TYP-ENV)} \\ G, H \vdash_k e : t & G, H \vdash_k E : H' \\ \\ \text{(TYP-PAT)} & \text{(TYP-HANDLER)} \\ \vdash_{pat} p : t, H & G, H \vdash h : t \end{array}$$

The predicate (TYP-EXP) states that under the global environment G and local environment of signals H, expression e has type t and kind k. The predicate (TYP-ENV) states that the equation E produces the type environment H' and has kind k. The predicate (TYP-PAT) defines the type and environment produced by a pattern p. Kinds are not necessary for patterns. The predicate (TYP-HANDLER) states that a reset handler h defines a value of type t.

Programs are typed under an initial global environment G_0 containing, in particular, the type signature for imported primitives. As an example, we give the signature of addition, equality, and the conditional. They are all of kind `A` since they can be executed on either a discrete clock or a continuous clock. The unit delay has kind `D`. The zero-crossing function $\texttt{up}(e)$ must be activated on a continuous clock, and hence its kind is `C`.

$$\begin{array}{lcl} (+) & : & \texttt{int} \times \texttt{int} \xrightarrow{\texttt{A}} \texttt{int} \\ (=) & : & \forall \beta.\beta \times \beta \xrightarrow{\texttt{A}} \texttt{bool} \\ \texttt{if} & : & \forall \beta.\texttt{bool} \times \beta \times \beta \xrightarrow{\texttt{A}} \beta \\ \texttt{pre}(.) & : & \forall \beta.\beta \xrightarrow{\texttt{D}} \beta \\ \texttt{.fby.} & : & \forall \beta.\beta \times \beta \xrightarrow{\texttt{D}} \beta \\ \texttt{up}(.) & : & \texttt{float} \xrightarrow{\texttt{C}} \texttt{zero} \end{array}$$

Generalization and Instantiation: Types schemes (σ) are obtained from types by generalizing their free variables. Because functions are defined globally, type variables can all be generalized; $gen\,(.)$ gives the generalization σ of $t_1 \to t_2$:

$$\begin{array}{lcl} gen(t_1 \xrightarrow{k} t_2) & = & \forall \beta_1, ..., \beta_n.t_1 \xrightarrow{k} t_2 \\ & & \text{if } \{\beta_1, ..., \beta_n\} = FV(t_1 \to t_2) \end{array}$$

A type scheme can be instantiated by replacing type variables by actual types. $Inst(\sigma)$ is the set of all possible instantiations of σ. A type $t_1 \xrightarrow{k} t_2$ can be instantiated by replacing its kind k by any kind k' such that $k \subseteq k'$.

$$\frac{k \subseteq k'}{(t \xrightarrow{k'} t')[t_1/\beta_1, ..., t_n/\beta_n] \in Inst(\forall \beta_1, ..., \beta_n.t \xrightarrow{k} t')}$$

The typing rules are presented in Figure 1.

1. Rule (DER). The derivative of x is well typed if e_1 and e_2 are of type `float` and h is well typed. The overall kind is `C`.
2. Rule (AND). Parallel equations (E_1 `and` E_2) are well typed if E_1 and E_2 are well typed. Both must have the same kind.
3. Rule (EQ). An equation $x = e$ is well typed if the types of x and e are the same. The overall kind k is the one of e.
4. Rules (EQ-DISCRETE) and (EQ-DISCRETE-DEFAULT). An equation $x = h$ `init` e activates x at discrete instants when zero-crossings in the handler h occur, h must be well typed and produce a value of type t which matches the type of e. Since e is not guarded by a zero-crossing, its kind is `C` which is

[10] An integral $1/s(x')$ initially i with reset condition z can be defined `let hybrid integr(x',i,z) = let der(x) = x' init i reset i every z in x`.

also the kind of the equation. For a signal x which evolves at discrete instants, it is meaningful to write $\texttt{last}(x)$ since x is given an initial value; $\texttt{last}(x)$ denotes the value of x at the previous instant of activation. On the contrary, when a default value is given, $\texttt{last}(x)$ is forbidden in order to prevent writing an equation like $x = 0$ `when` z `default` $\texttt{last}(x) + 1$ `init` 0, where the semantics of x would depend on the step of the solver as x would be incremented continuously.

5. Rule (APP). An application $f(e)$ is of type t' if the instantiated type of f is $t \xrightarrow{k} t'$ and if e is of type t and kind k. Consequently, it is not possible to use a function with kind `D` if the overall kind is expected to be `C` (and conversely).
6. Rule (LETIN). A local definition `let` E `in` e is well typed if E is well typed, and if e is well typed in an extended environment. The kind of the result is the kind of the whole.
7. Rule (LAST). The left limit $\texttt{last}(x)$ may only be accessed when the environment is $H, \texttt{last}(x) : t$.
8. Rule (CONST). We illustrate it for an immediate integer constant. Constants can be treated as global functions of arity 0.
9. Rules (VAR) and (VAR-LAST). If x is defined with type t, it can be used with the same type and with any kind.
10. Rule (PAIR). A pair (e_1, e_2) is of type $t_1 \times t_2$ if e_1 is of type t_1 and e_2 is of type t_2. As in the parallel composition of equations, e_1 and e_2 must have the same kind k.
11. Rules (PAT-PAIR) and (PAT-VAR). These rules build an initial environment for patterns (either inputs or outputs).
12. Rule (DEF-NODE). A function definition for f has the type signature $t_1 \xrightarrow{k} t_2$ if its input pattern p has type t_1 and its result q has type t_2 with kind k. The type signature is generalized.
13. Rule (DEF-SEQ). Function definitions are typed sequentially, augmenting the global environment.
14. Rule (HANDLER). A handler h is well typed if every expression e_i is of type t and z_i is of type `zero`; e_i must have kind `D` since it is only activated on a zero-crossing event. This precludes putting a continuous system under a zero-crossing.[11]

PROPERTY 1 (Subtyping). *The following property holds:*

$$G, H \vdash_{\mathtt{A}} e : t \Rightarrow (G, H \vdash_{\mathtt{C}} e : t) \wedge (G, H \vdash_{\mathtt{D}} e : t)$$

Proof : *Direct induction on the proof tree of* $G, H \vdash_{\mathtt{A}} e : t$.

3.3 A sketch of the semantics

There is not space enough to describe the semantics of the language in complete detail. Thus we present only its main principles and an illustrative sketch. The semantics relies on *non-standard analysis*, following our previous proposal [2] where a detailed motivation and mathematical justification are given. As far as this paper is concerned, the approach allows us to give a uniform semantics to the whole language and to establish properties of its compilation.

Cascades of causally-related zero-crossings are a major concern in the compilation of hybrid systems modeling languages [13, 14, 17]. This motivated Lee et al. [13, 14] to consider so-called *super-dense* time, in which several successive instants can occur at a single point of real-time. We contend that the semantic domain provided by non-standard analysis is an elegant alternative. In this section, we provide a glimpse of non-standard analysis and a sketch of how it is used to establish the semantics of our language.

[11] It is exactly the restriction imposed by SIMULINK where integrators are forbidden inside triggered blocks.

The sets $^\star\mathbb{R}$ ***and*** $^\star\mathbb{N}$ ***of non-standard reals and integers:*** We denote by $^\star\mathbb{R}$ and $^\star\mathbb{N}$ the non-standard extensions of $\mathbb{R}$ and $\mathbb{N}$, respectively. Two properties are especially important: $^\star\mathbb{R}$ and $^\star\mathbb{N}$ are both totally ordered, with $\mathbb{R}$ and $\mathbb{N}$ being respective sub-orders. $^\star\mathbb{N}$ contains elements $^\star n$ that are *infinitely large*, in that $^\star n > n$ for any $n \in \mathbb{N}$. $^\star\mathbb{R}$ contains elements ∂ that are *infinitesimal*, in that $0 < \partial < t$ for any $t \in \mathbb{R}_+$, ($\mathbb{R}_+$ are the non-negative reals). The set $BaseClock(\partial) = \{n\partial \mid n \in {}^\star\mathbb{N}\}$ is isomorphic to $^\star\mathbb{N}$ as a total order. Also, for every $t \in \mathbb{R}_+$ and any $\varepsilon > 0$, there exists $t' \in BaseClock(\partial)$ such that $|t' - t| < \varepsilon$, expressing that $BaseClock(\partial)$ is dense in $\mathbb{R}_+$. $BaseClock(\partial)$ is thus a natural candidate for a time index set and ∂ is the corresponding *time basis*. Note that $BaseClock(\partial)$ supports the aforementioned super-dense time. For $t = t_n = n\partial \in BaseClock(\partial)$:

$$\begin{array}{rcll} {}^\bullet t & = & t_{n-1} & \text{is the previous instant} \\ t^\bullet & = & t_{n+1} & \text{is the next instant} \end{array}$$

A *clock* T is a subset of $BaseClock(\partial)$, from which it inherits the total order by restriction. A *signal* s with clock domain T and co-domain V is a total function $s : T \mapsto V$. If T is a clock and b a signal $b : T \mapsto \mathbb{B}$, then T `on` b defines a subset of T comprising those instants where $b(t)$ is true:

$$T \texttt{ on } b = \{t \mid (t \in T) \wedge (b(t) = \texttt{true})\} \tag{1}$$

If $s : T \mapsto {}^\star\mathbb{R}$, we denote the instants when s crosses zero by:

$$T \texttt{ on up}(s) = \{t^\bullet \mid (t \in T) \wedge (s({}^\bullet t) \leq 0) \wedge (s(t) > 0)\} \tag{2}$$

Given a clock T, $base(T)$ returns its base clock.

$$\begin{array}{lcl} base(T \texttt{ on } s) & = & base(T) \\ base(BaseClock(\partial)) & = & \partial \end{array} \tag{3}$$

Discrete versus Continuous: Let s be a signal with clock domain T. It is typed *discrete* ($\mathtt{D}(T)$) either if it has been so declared, or if its clock is the result of a zero-crossing or a sub-clock of a discrete clock. Otherwise it is typed *continuous* ($\mathtt{C}(T)$). We thus have:

1. $\mathtt{C}(BaseClock(\partial))$
2. If $\mathtt{C}(T)$ and $s : T \mapsto {}^\star\mathbb{R}$ then $\mathtt{D}(T \texttt{ on up}(s))$
3. If $\mathtt{D}(T)$ and $s : T \mapsto \mathbb{B}$ then $\mathtt{D}(T \texttt{ on } s)$
4. If $\mathtt{C}(T)$ and $s : T \mapsto \mathbb{B}$ then $\mathtt{C}(T \texttt{ on } s)$

There are two reasons to define "discrete" in this way: 1) it is a syntactic criterion and can thus be statically checked; 2) it generally matches the intuition of a discrete clock. If $f : \mathbb{R}_+ \mapsto \mathbb{R}$ is continuous, then all instants belonging to

$$zero(f) \ =_{\text{def}} \ \{t \in {}^\star\mathbb{R}_+ \mid (f({}^\bullet t) \leq 0) \wedge (f(t) > 0)\}$$

are isolated (i.e., pairwise separated by non-empty intervals), and it is a discrete set in the mathematical sense. Functions like f, from which zero-crossings are constructed, would typically possess such properties. But, of course, for tricky signals, sets of zero-crossings may very well be Zeno. It is simply not possible, however, to statically check whether a clock is mathematically discrete.

The data-flow semantics for the differential equation and the zero-crossing expression are presented in Figure 2:

1. $integr^\#(T)(s)(s_0)(hs)$ defines the integration of the signal s initialized with value s_0 and reset according to hs. Let s' be the result of the integration. At the first instant, $s'(t_0)$ is set to $s_0(t_0)$. Afterward, if no zero-crossings are enabled, the current value $s'(t)$ is equal to the previous value $s'({}^\bullet t)$ plus the product of the previous differential $s({}^\bullet t)$ and the step-size ∂. If a zero-crossing is found, $s'(t)$ takes the value of the handler hs.
2. $up^\#(T)(s)$ is the result of a zero-crossing. It returns the value `true` when the signal s crosses 0 (from negative to positive). The effect is only visible at the next cycle of $BaseClock(\partial)$.

(DER)
$$\frac{G, H \vdash_{\mathtt{c}} e_1 : \mathtt{float} \qquad G, H \vdash_{\mathtt{c}} e_2 : \mathtt{float} \qquad G, H \vdash h : \mathtt{float}}{G, H \vdash_{\mathtt{c}} \mathtt{der}(x) \ = \ e_1 \ \mathtt{init} \ e_2 \ \mathtt{reset} \ h : [\mathtt{last}(x) : \mathtt{float}]}$$

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

(EQ)
$$\frac{G, H \vdash_k e : t}{G, H \vdash_k x \ = \ e : [x : t]}$$

(EQ-DISCRETE)
$$\frac{G, H \vdash h : t \qquad G, H \vdash_{\mathtt{c}} e : t}{G, H \vdash_{\mathtt{c}} x \ = \ h \ \mathtt{init} \ e : [\mathtt{last}(x) : t]}$$

(EQ-DISCRETE-DEFAULT)
$$\frac{G, H \vdash h : t \qquad G, H \vdash_{\mathtt{c}} e : t \qquad G, H \vdash_{\mathtt{c}} e_0 : t}{G, H \vdash_{\mathtt{c}} x \ = \ h \ \mathtt{default} \ e \ \mathtt{init} \ e_0 : [x : t]}$$

(APP)
$$\frac{t \xrightarrow{k} t' \in \mathit{Inst}(G(f)) \qquad G, H \vdash_k e : t}{G, H \vdash_k f(e) : t'}$$

(LETIN)
$$\frac{G, \ H, H_0 \vdash_k E : H_0 \qquad G, \ H, H_0 \vdash_k e : t}{G, H \vdash_k \mathtt{let} \ E \ \mathtt{in} \ e : t}$$

(LAST)
$$G, H + [\mathtt{last}(x) : t] \vdash_k \mathtt{last}(x) : t$$

(CONST)
$$G, H \vdash_k 4 : \mathtt{int}$$

(VAR)
$$G, H + [x : t] \vdash_k x : t$$

(VAR-LAST)
$$G, H + [\mathtt{last}(x) : t] \vdash_k x : t$$

(PAIR)
$$\frac{G, H \vdash_k e_1 : t_1 \qquad H \vdash_k e_2 : t_2}{G, H \vdash_k (e_1, e_2) : t_1 \times t_2}$$

(PAT-PAIR)
$$\frac{\vdash_{pat} p_1 : H_1 \qquad \vdash_{pat} p_2 : H_2}{\vdash_{pat} p_1, p_2 : t_1 \times t_2, H_{p_1} + H_{p_2}}$$

(PAT-VAR)
$$\vdash_{pat} x : t, [x : t]$$

(DEF-NODE)
$$\frac{\vdash_{pat} p : t_1, H_p \qquad G, H_p \vdash_k e : t_2}{G \vdash \mathtt{let} \ k \ f(p) = e : [f : gen_G(t_1 \xrightarrow{k} t_2)]}$$

(DEF-SEQ)
$$\frac{G \vdash d_1 : G_1 \qquad G, G_1 \vdash d_2 : G_2}{G \vdash d_1; d_2 : G_1 + G_2}$$

(HANDLER)
$$\frac{\forall i \in \{1, .., n\} \qquad G, H \vdash_{\mathtt{D}} e_i : t \qquad G, H \vdash_{\mathtt{c}} z_i : \mathtt{zero}}{G, H \vdash e_1 \ \mathtt{every} \ z_1 \mid ... \mid e_n \ \mathtt{every} \ z_n : t}$$

Figure 1. The typing rules

$$\begin{array}{lcll}
\mathit{integr}^{\#}(T)(s)(s_0)(hs)(t) & = & s'(t) & \text{where} \\
s'(t) & = & s_0(t) & \text{if } t = \min(T) \\
s'(t) & = & s'(^{\bullet}t) + \partial s(^{\bullet}t) & \text{if } \mathit{handler}^{\#}(T)(hs)(t) = \mathit{NoEvent} \\
s'(t) & = & v & \text{if } \mathit{handler}^{\#}(T)(hs)(t) = \mathit{Xcrossing}(v) \\
 & & & \\
\mathit{up}^{\#}(T)(s)(t) & = & \mathtt{false} & \text{if } t = \min(T) \\
\mathit{up}^{\#}(T)(s)(t^{\bullet}) & = & \mathtt{true} & \text{if } (s(^{\bullet}t) \leq 0) \wedge (s(t) > 0) \text{ and } (t \in T) \\
\mathit{up}^{\#}(T)(s)(t^{\bullet}) & = & \mathtt{false} & \text{otherwise}
\end{array}$$

Figure 2. The semantics of the differential equation and the zero-crossing expression

4. Compilation

The non-standard semantics is not operational. It serves rather as a reference to establish the correctness of several program transformations. By compilation, we mean the transformation needed to obtain an operational execution of the program. In this paper, we only consider execution with a single numerical solver. There are two problems to address:

1. The compilation of the discrete part, that is, the synchronous subset of the language.
2. The compilation of the continuous part which is to be linked to a black-box numerical solver.

In fact, the discrete part can be compiled by existing synchronous compilers, and we thus need only treat on the continuous part.

We propose a source-to-source transformation of the program into the discrete subset. This transformation removes all continuous computations, i.e., all ODE and zero-crossing expressions. The resulting code can then be compiled to give a transition function that computes one (discrete or continuous) step of the whole system. This function is passed to a numerical solver, along with a list of the continuous variables to be approximated, and the zero-crossings to be observed. Details are given in Section 5.

The translation is defined by four functions $\mathit{Tra}(e)$, $\mathit{TraEq}(E)$, $\mathit{TraDef}(d)$, and $\mathit{TraZ}(h)$. We use the following notation: zv denotes a vector of (incoming) zero-crossing variables $[z_1, \ldots, z_n]$ (each of type `bool`); upv is a vector of (outgoing) zero-crossing expression values $[up_1, \ldots, up_n]$ (each of type `float`); lxv and xv are vectors of continuous state variables, $[lx_1, \ldots, lx_k]$ and $[x_1, \ldots, x_k]$ respectively (each element has type `float`); dxv is a vector of continuous derivatives $[dx_1, \ldots, dx_k]$ (each of type `float`). We write $[z_1, \ldots, z_n] \mathbin{@} [z'_1, \ldots, z'_m]$ for the concatenation of two vectors $[z_1, \ldots, z_n, z'_1, \ldots, z'_m]$. To simplify the presentation, we write $[\,]$ for the empty equation (consider for example $\_ = 0$, where $\_$ denotes a wild card).

$\mathit{Tra}(e)$ defines the normalization of an expression and returns a tuple $\langle e', zv, upv, lxv, xv, dxv, E \rangle$. $\mathit{TraEq}(E)$ normalizes an equation and returns a tuple $\langle zv, upv, lxv, xv, dxv, E' \rangle$. $\mathit{TraDef}(d)$ defines the normalization of a declaration and returns a new declaration d'. $\mathit{TraZ}(h)$ normalizes a zero-crossing handler and returns a tuple $\langle h, zv, upv, lxv, xv, dxv, E \rangle$.

The transformation is defined recursively, and follows two principles: (1) For every zero-crossing computation $\mathtt{up}(e)$, a new input variable z_i and a new output variable up_i are added. Every occurrence of $\mathtt{up}(e)$ is replaced by the variable z_i. An equation of the form $up_i = e$ is added to define the new output. (2) For every

$$
\begin{array}{lcl}
Tra(op(e)) & = & \mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = Tra(e)\ \mathit{in} \\
 & & \langle op(e'), zv, upv, lxv, xv, dxv, E\rangle \\
Tra(y) & = & \langle y, [\,], [\,], [\,], [\,], [\,], [\,]\rangle \\
Tra(v) & = & \langle v, [\,], [\,], [\,], [\,], [\,], [\,]\rangle \\
Tra(\texttt{last}(y)) & = & \langle \texttt{last}(y), [\,], [\,], [\,], [\,], [\,], [\,]\rangle \\
Tra(\texttt{up}(e)) & = & \mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = Tra(e)\ \mathit{in} \\
 & & \langle z, z.zv, u.upv, lxv, xv, dxv, u = e'\ \texttt{and}\ E\rangle \\
 & & \text{where } z \text{ and } u \text{ are fresh variables} \\
Tra(e_1\ \texttt{fby}\ e_2) & = & \mathit{let}\ \langle e'_1, zv_1, upv_1, lxv_1, xv_1, dxv_1, E_1\rangle = Tra(e_1)\ \mathit{in} \\
 & & \mathit{let}\ \langle e'_2, zv_2, upv_2, lxv_2, xv_2, dxv_2, E_2\rangle = Tra(e_2)\ \mathit{in} \\
 & & \langle e'_1\ \texttt{fby}\ e'_2, zv_1 \mathbin{@} zv_2, upv_1 \mathbin{@} upv_2, lxv_1 \mathbin{@} lxv_2, \\
 & & xv_1 \mathbin{@} xv_2, dxv_1 \mathbin{@} dxv_2, E_1\ \texttt{and}\ E_2\rangle \\
Tra((e_1, e_2)) & = & \mathit{let}\ \langle e'_1, zv_1, upv_1, lxv_1, xv_1, dxv_1, E_1\rangle = Tra(e_1)\ \mathit{in} \\
 & & \mathit{let}\ \langle e'_2, zv_2, upv_2, lxv_2, xv_2, dxv_2, E_2\rangle = Tra(e_2)\ \mathit{in} \\
 & & \langle (e'_1, e'_2), zv_1 \mathbin{@} zv_2, upv_1 \mathbin{@} upv_2, lxv_1 \mathbin{@} lxv_2, xv_1 \mathbin{@} xv_2, dxv_1 \mathbin{@} dxv_2, E_1\ \texttt{and}\ E_2\rangle \\
Tra(\texttt{let}\ E\ \texttt{in}\ e) & = & \mathit{let}\ \langle zv_1, upv_1, lxv_1, xv_1, dxv_1, E_1\rangle = TraEq(E)\ \mathit{in} \\
 & & \mathit{let}\ \langle e', zv_2, upv_2, lxv_2, xv_2, dxv_2, E_2\rangle = Tra(e)\ \mathit{in} \\
 & & \langle e', zv_1 \mathbin{@} zv_2, upv_1 \mathbin{@} upv_2, lxv_1 \mathbin{@} lxv_2, xv_1 \mathbin{@} xv_2, dxv_1 \mathbin{@} dxv_2, E_1\ \texttt{and}\ E_2\rangle \\
 & & \text{assuming unique names for variables in } E \\
Tra(f(e)) & = & \mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = Tra(e)\ \mathit{in} \\
 & & \langle f(e'), zv, upv, lxv, xv, dxv, E\rangle \\
 & & \text{if } \mathit{KindOf}(f) \in \{\texttt{A}, \texttt{D}\} \\
Tra(f(e)) & = & \mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = Tra(e)\ \mathit{in} \\
 & & \langle r, z.zv, up.upv, lx.lxv, x.xv, dx.dxv, (r, up, x, dx) = f(z, lx, e')\ \texttt{and}\ E\rangle \\
 & & \text{if } \mathit{KindOf}(f) = \texttt{C} \text{ and where } r, z, up, lx, x, \text{ and } dx \text{ are fresh variables} \\
TraEq(x = e) & = & \mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = Tra(e)\ \mathit{in} \\
 & & \langle zv, upv, lxv, xv, dxv, x = e'\ \texttt{and}\ E\rangle \\
TraEq(\texttt{der}(x) = e\ \texttt{init}\ e_0\ \texttt{reset}\ h) & = & \mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = Tra(e)\ \mathit{in} \\
 & & \mathit{let}\ \langle e'_0, zv_0, upv_0, lxv_0, xv_0, dxv_0, E_0\rangle = Tra(e_0)\ \mathit{in} \\
 & & \mathit{let}\ \langle h', zv_h, upv_h, lxv_h, xv_h, dxv_h, E_h\rangle = TraZ(h)\ \mathit{in} \\
 & & \langle zv \mathbin{@} zv_0 \mathbin{@} zv_h, upv \mathbin{@} upv_0 \mathbin{@} upv_h, \\
 & & lx.(lxv \mathbin{@} lxv_0 \mathbin{@} lxv_h), x.(xv \mathbin{@} xv_0 \mathbin{@} xv_h), dx.(dxv \mathbin{@} dxv_0 \mathbin{@} dxv_h), \\
 & & (x = h'\ \texttt{default}\ lx\ \texttt{init}\ e'_0)\ \texttt{and}\ (dx = e')\ \texttt{and}\ E\ \texttt{and}\ E_0\ \texttt{and}\ E_h\rangle \\
 & & \text{where } lx \text{ and } dx \text{ are fresh variables} \\
TraEq(x = h\ \texttt{init}\ e) & = & \mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = Tra(e)\ \mathit{in} \\
 & & \mathit{let}\ \langle h', zv_h, upv_h, lxv_h, xv_h, dxv_h, E_h\rangle = TraZ(h)\ \mathit{in} \\
 & & \langle zv \mathbin{@} zv_h, upv \mathbin{@} upv_h, lxv \mathbin{@} lxv_h, xv \mathbin{@} xv_h, dxv \mathbin{@} dxv_h, \\
 & & (x = h'\ \texttt{init}\ e')\ \texttt{and}\ E\ \texttt{and}\ E_h\rangle \\
TraEq(x = h\ \texttt{default}\ e\ \texttt{init}\ e_0) & = & \mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = Tra(e)\ \mathit{in} \\
 & & \mathit{let}\ \langle e'_0, zv_0, upv_0, lxv_0, xv_0, dxv_0, E_0\rangle = Tra(e_0)\ \mathit{in} \\
 & & \mathit{let}\ \langle h', zv_h, upv_h, lxv_h, xv_h, dxv_h, E_h\rangle = TraZ(h)\ \mathit{in} \\
 & & \langle zv \mathbin{@} zv_0 \mathbin{@} zv_h, upv \mathbin{@} upv_0 \mathbin{@} upv_h, lxv \mathbin{@} lxv_0 \mathbin{@} lxv_h, xv \mathbin{@} xv_0 \mathbin{@} xv_h, \\
 & & dxv \mathbin{@} dxv_0 \mathbin{@} dxv_h, (x = h'\ \texttt{default}\ e'\ \texttt{init}\ e'_0)\ \texttt{and}\ E\ \texttt{and}\ E_0\ \texttt{and}\ E_h\rangle \\
TraEq(E_1\ \texttt{and}\ E_2) & = & \mathit{let}\ \langle zv_1, upv_1, lxv_1, xv_1, dxv_1, E'_1\rangle = TraEq(E_1)\ \mathit{in} \\
 & & \mathit{let}\ \langle zv_2, upv_2, lxv_2, xv_2, dxv_2, E'_2\rangle = TraEq(E_2)\ \mathit{in} \\
 & & \langle zv_1 \mathbin{@} zv_2, upv_1 \mathbin{@} upv_2, lxv_1 \mathbin{@} lxv_2, xv_1 \mathbin{@} xv_2, dxv_1 \mathbin{@} dxv_2, E'_1\ \texttt{and}\ E'_2\rangle \\
TraZ(e_1\ \texttt{every}\ z_1 \mid \ldots \mid e_n\ \texttt{every}\ z_n) & = & \mathit{let}\ \langle z'_i, zv_i, upv_i, lxv_i, xv_i, dxv_i, E_i\rangle = TraEq(z_i)\ \mathit{in} \\
 & & \langle e_1\ \texttt{every}\ z'_1 \mid \cdots \mid e_n\ \texttt{every}\ z'_n, zv_1 \cdots \mathbin{@} zv_n, upv_1 \cdots \mathbin{@} upv_n, \\
 & & lxv_1 \cdots \mathbin{@} lxv_n, xv_1 \cdots \mathbin{@} xv_n, dxv_1 \cdots \mathbin{@} dxv_n, E_1 \cdots \texttt{and}\ E_n\rangle \\
TraDef(\texttt{let}\ k\ f(y) = e) & = & \mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = Tra(e)\ \mathit{in} \\
 & & \texttt{let D}\ f(zv, lxv, y) = \texttt{let}\ E\ \texttt{in}\ (e', upv, xv, dxv) \\
 & & \text{if } k = \texttt{C} \\
TraDef(\texttt{let}\ k\ f(y) = e) & = & \texttt{let}\ k\ f(y) = e \text{ if } k \in \{\texttt{A}, \texttt{D}\}
\end{array}
$$

Figure 3. The source-to-source transformation

ODE $\texttt{der}(x) = e\ \texttt{init}\ e_0$, a new input variable lx and two new output variables x and dx are added.

For every function definition f, $KindOf(f)$ defines its kind $k \in \{\texttt{A}, \texttt{D}, \texttt{C}\}$. The definition of the translation is shown in Figure 3.

1. The application of an operation $op(e)$ to an expression is translated into an operation that gathers all the information about e.
2. A variable y, a constant v, and $\texttt{last}(y)$ are left unchanged.
3. For an expression such as $e_1\,\texttt{fby}\,e_2$ or (e_1, e_2), we gather information from e_1 and e_2: and then take the unions of the various components.
4. Local definitions $\texttt{let}\ E\ \texttt{in}\ e$ are effectively flattened. This is sound, provided variables are renamed appropriately, because continuous expressions cannot have side-effects and may thus be reordered without changing the meaning of a program.
5. A zero-crossing expression $\texttt{up}(e)$ is translated into a fresh variable z, which is added to the list of input boolean variables, the equation $u = e$ is added to the set of equations, and u is added to the list of output variables.
6. There are two cases to consider for the application $f(e)$. If f is combinatorial (kind A) or discrete (kind D) then the function call is unchanged. If, however, f is a function involving continuous computations, then it has been compiled into a function expecting two extra arguments and returning three extra outputs. We thus replace the function call $f(e)$ by an equation of the form $(r, upz, x, dx) = f(z, lx, e)$ where r is the result of calling the function, upz is the vector of zero-crossing values to observe, lx is the vector of updated states, and dx is the vector of derivatives. Each new variable is added to the appropriate vector.
7. For each derivative $\texttt{der}(x) = e\ \texttt{init}\ e_0\ \texttt{reset}\ h$ is added a new variable lx to receive the (left-limit) state value calculated by the solver, and two new equations that define a state value x (which is identical to lx in continuous steps but which may be reset in discrete steps) and a derivative value dx. Note that, in a later stage of translation, all occurrences of $\texttt{last}(x)$ must be replaced by the corresponding lx.
8. Other equations define variables which are modified at discrete instants only. This does not introduce any extra variables.
9. In reset handlers $e_i\ \texttt{every}\ z_i$, the expressions e_i are discrete and thus not translated.
10. A function definition with kind C is translated into a discrete function extended with two new inputs and three new outputs.
11. Functions with $k \in \{\texttt{A}, \texttt{D}\}$ are left unchanged.

5. Interfacing with a numerical solver

The compilation scheme introduced in the previous section has been implemented using the SUNDIALS library (version 2.4.0).

5.1 Ocaml interface to Sundials

SUNDIALS [11] is a collection of numerical solvers for nonlinear, differential, and algebraic equations. One of which, called CVODE, solves ODE initial value problems. We have implemented an Ocaml interface to the CVODE variable-step solver,[12] which provides a convenient programming environment that simplifies development—for instance, the solver is configured through algebraic data types rather than multiple function calls, error conditions are signaled by exceptions rather than return codes, and callback routines can be higher-order closures. Vectors are passed to and from CVODE as Ocaml BigArrays to minimize copying.

[12] It lacks parallel nvectors, but is otherwise comprehensive.

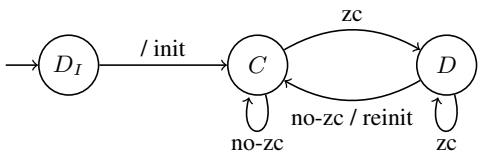

Figure 4. Basic structure of the simulation algorithm

5.2 The simulation algorithm

The CVODE solver approximates continuous states and detects zero-crossings, but some extra routines are needed for initialization and discrete transitions. The simplicity of these routines is a direct consequence of the properties guaranteed by the type system of Section 3.2, and the structure of functions produced by the compilation algorithm described in Section 4. While we pay attention to the details of the solver, ultimately we treat it, with all its sophisticated internal numerical algorithms, simply as a black box for approximating continuous signals and detecting interesting events with reasonable accuracy.

The step function. The simulation algorithm takes the top-level function resulting from the transformation and compilation of a hybrid data-flow program, and combines it with calls to the Ocaml CVODE interface to produce another function that simulates the system from $t = 0$ for a given or unbounded period.

The nodes of the original program are transformed exactly as described in Section 4. Some extra routines are needed to map to and from arrays of the appropriate size, but the details are relatively straightforward. The transformed nodes are readily compiled into Ocaml functions by compilers like the one of Lucid Synchrone [19]. The final result is a single function that need only take four arrays as arguments, since inputs and outputs are typically handled by imperative calls from within the (discrete) functions of a data-flow program.

Simulation phases. The basic structure of the simulation algorithm is shown in Figure 4. The algorithm begins in an initial discrete phase, D_I, where the step function is called to initialize both its internal state and an array of continuous states. The latter are used as initial values to create a session with the CVODE library. After which the algorithm alternates between a continuous phase, C, and a discrete phase, D.

The D_I and D phases differ from each other in four main ways. First, several CVODE *initialization* functions are called on leaving the former, while only the *reinitialization* function is called on leaving the latter. Second, none of the zero-crossings can be true in the call to the step function from D_I, whereas at least one will be true in the call from D. Third, the call to the step function from D_I updates all discrete and continuous states with their initial values, while from D it may only update a subset of the states. In both cases the step function is called in exactly the same way; the difference in behavior is due only to its internal state; this kind of initialization reaction is typical for a synchronous language kernel. Fourth, no new zero-crossings can be generated in D_I because there are no last values against which to compare the zero-crossing values.

Note that starting the algorithm in a discrete phase is the only way of having a discrete reaction at $t = 0$, since the continuous solver ignores zero-crossings after initialization (and reinitialization) until the values of the corresponding expressions are not zero.

The C phase is entered after initialization. In this phase, the numeric solver is repeatedly executed to approximate the evolution of continuous states and to search for zero-crossings. The solver in turn makes two types of callbacks: one for the values of continuous

state differentials, and the other for the values of zero-crossing expressions.[13] The step function is called in both cases.

For differential calls, none of the zero-crossings will be enabled, and as such the type system guarantees that the continuous state array will never be modified by the step function (though it may be read, of course). This is essential: in the continuous phase, continuous states may only be changed indirectly through their corresponding differential values. Moreover, neither will the internal discrete states of a well-typed program change in such reactions. It is for this reason that the step function can be called repeatedly by the solver in the continuous phase without having to shuffle and cache arrays of discrete variables.[14]

Similar statements can be made about the calls to evaluate zero-crossing expressions. In fact, the step function updates both the values of the differential and the zero-crossing value arrays in both situations. This has no effect on the results of a simulation, and it is, in any case, only a slight inefficiency easily remedied by the addition of extra boolean flags (which would be, essentially, activation clocks for the differential and zero-crossing value variables).

The algorithm cycles in the continuous phase, constantly increasing the value of the simulation time, until the solver reports that one or more zero-crossings have been found. The algorithm then enters the D phase. At least one zero-crossing will be enabled when the step function is called from the D phase, so a discrete computation will occur, and also possibly a discrete state change.

Note that while a step function may be activated several times within a C phase, only the continuous states, which are managed by the solver, may change. Within D phases, however, the step function is executed exactly once for each set of triggering events. Thus it is safe for it to change internal variable values, and even, in more complete versions of the language, to perform imperative actions (like sampling inputs values, or logging data for graphing).

Cascaded zero-crossings Different possibilities in the treatment of cascaded zero-crossings within the D phase give rise to different semantic models.

In the *delta-delay model* employed in this paper, zero-crossings that occur within the step function during the D phase are not detected immediately. Rather, after calling the step function, new values for the zero-crossing array are calculated by comparing the elements of a previous zero-crossing value array with those most recently calculated by the step function. If no new zero-crossings are detected, the algorithm returns to the C phase. Otherwise it executes another discrete step; any number of D phases may thus occur consecutively; indeed, they may continue to occur indefinitely.

An alternative *instant-cascade model* has, in some ways, a stronger affinity with the usual approach taken in synchronous languages, and also with the approach taken in SIMULINK. In this model, new zero-crossings are detected immediately within the step function. This complicates the transformation algorithm, which must insert extra equations to calculate the status of zero-crossings by comparing the previous and current values of zero-crossing expressions. But afterward the synchronous language compiler inherently sorts the equations to ensure that any new zero-crossing events generated within a step are properly detected within the same step. Programs with cyclic dependencies, on zero-crossing events or otherwise, are rejected during compilation. There is thus no need to check for new zero-crossings after a D phase—no two D phases ever occur without an intervening C phase—the algorithm simply returns to the C-phase.

Finally, note that, in all of these models, no simulation time passes during a sequence of D phases: a fundamental characteristic of synchronous languages!

[13] Or, (rising) 'root functions' to use the SUNDIALS terminology.

[14] As is required, for example, by SIMULINK s-functions.

6. Discussion and Related Work

While there exists much research on the theoretical properties, and the analysis of hybrid systems, e.g. by model checking, there are fewer studies on hybrid systems modelers from a programming language perspective. We focus in this section on approaches for expressing and executing causal hybrid models. Besides early work for a hybrid extension of SIGNAL [1], where neither static typing nor non-standard analysis were considered, there are four main bodies of closely related work, namely recent approaches by P.J. Mosterman and colleagues at The Mathworks [8, 15], Fran and FRP [9, 18], the Ptolemy project [13, 14], and Scicos [17].

Mosterman et al. The work of P.J. Mosterman and colleagues at The Mathworks [8, 15] attempts to establish SIMULINK on a sound semantic basis. They show, for instance, how (a restricted class of) variable step solvers can be given a functional *stream* semantics. The class of solvers is first restricted to those relying on *explicit schemes,* as *implicit* ones cannot be put in explicit functional form. While this indeed provides a hybrid systems modeler with a stream semantics, the semantics is very complex since it explicits the discretization method—in particular, changing the latter changes the semantics. Moreover, both the discrete and continuous parts are treated together making the semantics unnecessarily complex.

Block-diagram simulation tools such as SIMULINK already provide a form of static typing. It is integrated into the mechanism for inferring the rate at which a signal has to be computed, and works through forward and backward propagation. Nonetheless, the way the verification of rates is done by the compiler is imprecise [20]. We advocate making strong typing an integral part of the language specification. Furthermore, while SIMULINK associates kinds to signals, we claim that associating them to block structures gives a simpler system for languages based on *activation conditions*.

Fran and FRP While Fran [9] focuses on interactive animations, it can also be used to simulate physical phenomenon expressed as ODEs. FRP [18] is descended from Fran; its principle innovation being a shift to arrows-based combinators. Both Fran and FRP are embedded in Haskell, and they are thus very expressive languages. We propose a more austere first-order language, which better suits our aims of applying serious compilation techniques to generate embedded code and of precisely understanding and treating effects like cascaded zero-crossings. Furthermore, the expressivity of Fran and FRP comes at the cost of significant memory leaks. In contrast, our proposal, being a Lustre-like language with a clock calculus, can guarantee bounds on the amount of memory required.

Another important distinction is that we have focused on achieving very precise numerical approximations by exploiting a state-of-the-art numerical solver. The approximation of integrals in Fran is handled internally by a fourth-order Runge-Kutta algorithm applied locally, that is signal by signal without a global view of rates of change, errors, and tolerances. While integral approximation is externalized in Fran, only simple techniques are used (at least in *Yampa*): a fixed step solver with rectangular rules to approximate integrals, without even the predicate event (zero-crossing) detection algorithm of Fran. Continuous evolutions in Fran and FRP are effectively simulated by discrete techniques.

The work of the Ptolemy group E.A. Lee and H. Zheng [13, 14] use the *tagged signals model* [12] as the semantic support for the CT Domain in Ptolemy II.[15] Events are tagged with an extended time index from the set $\mathbb{R}_+ \times \mathbb{N}$ and associated lexicographic order, which the authors call *super-dense time*. We can avoid using super-dense time because our non-standard index set is both discrete and dense. In particular, the existence of previous ${}^\bullet t$ and next $t^\bullet$

[15] http://ptolemy.eecs.berkeley.edu/ptolemyII

instants replaces the multi-dimensional instants $(t, 0)$ and $(t, 1)$. Furthermore, the approach in Ptolemy II is made complicated by issues of smoothness, Lipschitzness, existence and uniqueness of solutions, Zenoness, etc. (see [13, §6] on "Ideal Solver Semantics" and [14, §7] on "Continuous Time Models"). In contrast, these issues play no role in our non-standard semantics, and thus no role in our compilation scheme; they may, however, manifest at runtime.

Scicos and the work of R. Nikoukhah Scicos[16] is freely available software developed by R. Nikoukhah at INRIA [6, 16]. Its principles derive from the Signal language, and include an attempt to cleanly separate continuous from discrete dynamics, but some issues are yet unsolved. Cascaded zero-crossings are discussed [17], and the "synchronous interpretation" of them as truly simultaneous is rejected in favor of a micro-step interpretation, where simultaneous zero-crossings interleave non-deterministically. We prefer a synchronous interpretation where programmers explicitly describe responses to multiple zero-crossings. Then non-determinism arises solely in numerical solvers, and not in the semantics. Note that our work employs micro-step sequencing of causally-related cascaded zero-crossings, but in a completely deterministic manner.

7. Conclusion

This paper has presented an extension of a synchronous language similar to LUSTRE that allows the combination of stream equations and ODEs with reset. The language has a static type system that separates the continuous parts, which have to be exercised by a numerical solver, from the discrete parts, which are guaranteed not to evolve during intermediate solver steps.

This strong separation makes it possible to reduce the compilation process to a source-to-source pre-processing step which adds extra inputs and outputs for continuous signals. This, together with the guarantee that the discrete state will only change in response to zero-crossing events, means that existing compilation techniques can be applied directly. The efficient code thus generated is readily combined with a single black box numerical solver.

References

[1] A. Benveniste. Compositional and uniform modelling of hybrid systems. *IEEE Trans. on Automatic Control*, 43(4):579–584, April 1998.

[2] A. Benveniste, B. Caillaud, and M. Pouzet. The fundamentals of hybrid systems modelers. In *49th IEEE Int. Conf. on Decision and Control (CDC)*, Atlanta, Georgia, USA, December 2010.

[3] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[4] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation of synchronous data-flow languages. In *ACM Int. Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

[5] S. Bliudze and D. Krob. Modelling of complex systems: Systems as dataflow machines. *Fundamenta Informaticae*, 91(2):251–274, 2009.

[6] S.L. Campbell, J.-Ph. Chancelier, and R. Nikoukhah. *Modeling and Simulation in Scilab/Scicos*. Springer, 2006.

[7] L.P. Carloni, R. Passerone, A. Pinto, and A.L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1(1/2), 2006.

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

[9] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. of the ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'97)*, pages 263–273, Amsterdam, The Netherlands, August 1997.

[16] http://www-rocq.inria.fr/scicos/

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

[11] A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, and C.S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. on Mathematical Software*, 31(3):363–396, September 2005.

[12] E.A. Lee and A.L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[13] E.A. Lee and H. Zheng. Operational semantics of hybrid systems. In *Hybrid Systems: Computation and Control (HSCC)*, volume 3414 of *LNCS*, pages 25–53, 2005.

[14] E.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.

[15] P.J. Mosterman, J. Zander, G. Hamon, and B. Denckla. Towards computational hybrid system semantics for time-based block diagrams. In *3rd IFAC Conf. on Analysis and Design of Hybrid Systems (ADHS'09)*, pages 376–385, Zaragoza, Spain, September 2009.

[16] M. Najafi and R. Nikoukhah. Implementation of hybrid automata in scicos. In *IEEE Multi-conference on Systems and Control*, 2007.

[17] R. Nikoukhah. Hybrid dynamics in modelica: Should all events be considered synchronous? In *First Int. Workshop on Equation-Based Object Oriented Languages and Tools (EOOLT 2007)*, pages 37–48, Berlin, Germany, 2007.

[18] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Haskell'02: Proc. of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64, Pittsburgh, Pennsylvania, January 2002.

[19] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at: `www.lri.fr/~pouzet/lucid-synchrone`.

[20] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time simulink to lustre. *ACM Trans. on Embedded Computing Systems*, 4(4):779–818, 2005.

A. Examples in Simulink

Figure 5 gives the translation in Simulink of the introductory examples. All clocks have been set to "inherited" (-1). Run with Simulink version 7.7.0.471, R2008b, changing the frequency (Sine Wave block) influences the value of `p`.

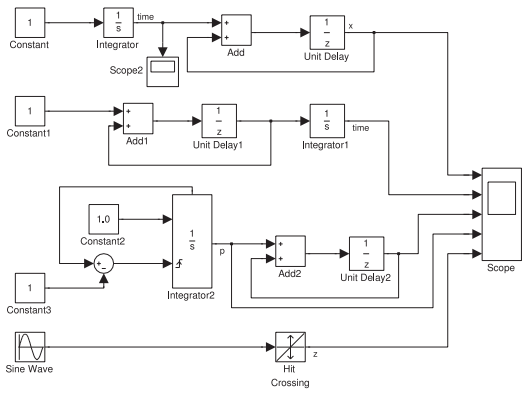

Figure 5. A Program with a Rate Transition Error