A Synchronous Look at the Simulink Standard Library

Timothy Bourke, Francois Carcenac, Jean-Louis Colaço, Bruno Pagano,
Cédric Pasteur, Marc Pouzet

A Synchronous Look at the Simulink Standard Library*

TIMOTHY BOURKE, INRIA Paris and École normale supérieure
FRANCOIS CARCENAC, JEAN-LOUIS COLAÇO, BRUNO PAGANO, and CÉDRIC PASTEUR, ANSYS/Esterel-Technologies
MARC POUZET, Université Pierre et Marie Curie, École normale supérieure, and INRIA Paris

Hybrid systems modelers like Simulink come with a rich collection of discrete-time and continuous-time blocks. Most blocks are not defined in terms of more elementary ones—and some cannot be—but are instead written in imperative code and explained informally in a reference manual. This raises the question of defining a minimal set of orthogonal programming constructs such that most blocks can be *programmed* directly and thereby given a specification that is mathematically precise, and whose compiled version performs comparably to handwritten code.

In this paper, we show that a fairly large set of blocks of a standard library like the one provided by Simulink can be programmed in a precise, purely functional language using stream equations, hierarchical automata, Ordinary Differential Equations (ODEs), and deterministic synchronous parallel composition. Some blocks cannot be expressed in our setting as they mix discrete-time and continuous-time signals in unprincipled ways that are statically forbidden by the type checker.

The experiment is conducted in Zélus, a synchronous language that conservatively extends Lustre with ODEs to program systems that mix discrete-time and continuous-time signals.

Additional Key Words and Phrases: Hybrid systems, Synchronous languages, Block diagrams

ACM Reference format:
Timothy Bourke, Francois Carcenac, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. 2017. A Synchronous Look at the Simulink Standard Library. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (August 2017), 23 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Hybrid systems modelers [12] like Simulink[1] are used to write executable models of dynamical systems that include control software and physical plants. Models are not just for simulation: they are used for test-case generation, formal verification, and the generation of embedded code. It is important to ensure that different interpretations of a model are consistent, so that, paraphrasing Gérard Berry, “what you simulate is what you execute” [8].

One characteristic of existing hybrid systems modeling languages is the multiplicity of language constructs. Models mix stream equations, Ordinary Differential Equations (ODEs), zero-crossing events, hierarchical automata, and also side effects, loops, different forms of modular composition (for example, subsystem blocks), and calls to external functions produced by other tools; and all of

*This article was presented in the International Conference on Embedded Software 2017, October 15–20, Seoul, Korea and appears as part of the ESWEEK-TECS special issue.

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

1539-9087/2017/8-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

these features can be composed in parallel. While this expressiveness is unquestionably useful for writing real applications, the undisciplined composition of language constructs leads to fragile and unsafe models [28] that are difficult to understand modularly and debug because the behavior of a block can change dramatically when other unconnected blocks are added in parallel. Problems can arise, for instance, when a continuous-time signal is used where a discrete-time signal is expected and vice versa [6], when a block explicitly refers to the internal time steps of the solver—the so-called *major steps* of SIMULINK solvers, —when a call is made during integration to a function that unexpectedly performs a side effect, or when two parallel blocks write to the same shared variable. Some of these situations are detected statically and trigger errors or warnings, but not all of them. The static detection is not imposed as a strong type discipline: some unsafe models, in the sense of [28], pass the static check while some safe ones do not. Furthermore, there is no formal definition of exactly what constitutes a safe model.

Moreover, real models use large subsets of a standard library of discrete-time and continuous-time blocks. Most of these blocks are not *programmed* from more elementary and mathematically defined primitive constructs but are implemented directly in imperative code and described informally. This makes it difficult to understand models and to verify them. It also invites the question:

Which orthogonal constructs should a minimal language for hybrid systems provide, so that a comprehensive library of classical discrete and continuous control blocks can be programmed in a mathematically precise way, and so that the program is the specification and can be compiled to give performance comparable to handwritten code?

Synchronous languages [7] provide a partial answer to this question: they focus on discrete-time signals and systems only, that is, the software parts of a hybrid system. Input, output, and local signals are abstracted as sequences of values that advance synchronously, and systems are functions over sequences. These languages are mathematically defined, their compilers ensure strong safety properties by rejecting, for example, programs that are not deterministic, and they are translated into sequential code that executes in bounded time and space. Such features contribute to the fact that SCADE Suite[2] is now the reference for critical software development in civil avionics. Moreover, early research [28] into translating a *safe* discrete subset of SIMULINK into LUSTRE [18] has been transferred into academic[3] and industrial[4] tools. A synchronous language like SCADE can be used to program a library of discrete-time control blocks, thereby reducing the number of built-in primitives and simplifying the analysis of models. That said, synchronous languages do not permit the adequate modeling or efficient simulation of systems that mix discrete and continuous-time signals.

Earlier work [6] proposes a language for hybrid systems that reuses synchronous language principles and an existing compiler infrastructure. The language was initially an extension of LUSTRE with ODEs and zero-crossing events [5], but was later extended with hierarchical automata [4], and implemented in a compiler producing statically scheduled code paired with an off-the-shelf numerical ODE solver [10]. These results are the foundation of ZÉLUS [11][5] and the industrial prototype SCADE HYBRID [10]. The latter is a conservative extension of SCADE 6 (discrete models are compiled as usual) with new constructs for mixing discrete-time and continuous-time signals. Programs must follow a strict typing discipline that forbids both using a discrete-time signal where a continuous-time one is expected and vice versa, and explicitly referring to the major steps of the solver: every side-effect and state change must be aligned with a zero-crossing [5]. Algebraic loops,

[2] http://www.esterel-technologies.com/products/scade-suite/

[3] https://github.com/coco-team/cocoSim/wiki/CoCoSim

[4] http://www.esterel-technologies.com/products/scade-suite/gateways/scade-suite-gateway-simulink

[5] http://zelus.di.ens.fr

which may lead to systems with zero or many solutions, are statically rejected [3]. It is reasonable to ask: what are the practical consequences of these design choices for writing real applications? How do they respond to our original question? Are the language and its type discipline expressive enough to define a comprehensive library of safe discrete-time and continuous-time blocks? Which blocks cannot be defined?

Contributions. We show that a large set of typical control blocks, like those provided in the Simulink standard library, can be *programmed* in a precise, purely functional style using stream equations, hierarchical automata, ODEs, and synchronous composition. Some continuous-time blocks cannot be adequately defined because they mix discrete and continuous time in unprincipled ways. Specifically, the memory, derivative, rate limiter, and time and transport delay blocks can only be applied to discrete-time signals. These blocks explicitly rely on the internal steps of the solver and do not respect the type discipline described above.

The aim here is not to define a translator from Simulink to Zélus or Scade Hybrid, nor to prove that we generate identical output traces for 'equivalent' blocks (though we did compare simulation results on examples). Writing a translator would only partially address the questions we pose. Some standard library blocks of Simulink are not defined as the composition of simpler ones but are implemented in sequential code to which an automatic translation would not apply; some others are considered unsafe in Zélus and Scade Hybrid and cannot be represented. Finally, blocks are only described informally in the documentation making a proof of equivalence difficult to establish.

Experiments are performed in Zélus and Scade Hybrid. For simplicity and lack of space, we only present Zélus programs. We describe the blocks that pose interesting problems and highlight the strengths and limitations of the language, and identify some open questions on language extensions. Complete definitions in Zélus and Scade Hybrid are indicated by '♣'s which point to http://zelus.di.ens.fr/emsoft2017/. This experiment presents, for the first time, new language and compilation features in Zélus, namely arrays defined by a SISAL-like [16] looping construct and higher-order functions with static expansion prior to code generation, together with associated extensions to the modular type inference and causality analyses.

Structure. Section 2 presents a short introduction to Zélus. Section 3 gives the definition of some of the main blocks of the standard library in both discrete and continuous time. We identify those that cannot be defined in Zélus and Scade Hybrid. Section 4 discusses the approach and related work. We conclude in Section 5.

2 ZÉLUS = LUSTRE + ODES + ZERO-CROSSINGS

In Zélus, a discrete-time signal is an infinite sequence of values, termed a *stream*. Streams are defined by mutually recursive equations and advance synchronously. The semantic model is that of Lustre. An equation $x = a$, where a is an expression, means that $(x_n = a_n)$, for all $n \in \mathbb{N}$. The parallel composition of two equations is synchronous, that is, $x = a$ **and** $y = b$, where a and b are expressions, means $(x_n = a_n) \land (y_n = b_n)$, for all $n \in \mathbb{N}$. For example, if $x = (x_i)_{i \in \mathbb{N}}$ is a stream, the stream function filter given below computes a value $y = (y_i)_{i \in \mathbb{N}}$ from x by applying feedback through a unit-delay (**fby**).[6] ♣

```
let node filter(x) = y where
  rec y = 0.2 *. x +. 0.8 *. (0.0 fby y)
```

[6] The +. and *. operators are, respectively, floating-point addition and multiplication.

The semantics of this program is given by,

$$\begin{aligned}
\text{for } n > 0, \quad y_n &= (0.2x + 0.8(0\,\textbf{fby}\,y))_n \\
&= 0.2x_n + 0.8(0\,\textbf{fby}\,y)_n \\
&= 0.2x_n + 0.8y_{n-1} \\
\text{and,} \quad y_0 &= 0.2x_0.
\end{aligned}$$

The keyword **node** indicates that filter is a stateful discrete-time function: it takes an input stream and returns an output stream. The current output value may depend on both the current and past input values. Functions may be composed arbitrarily provided there are no instantaneous feedback loops.[7] In the above function, time is logical, no mention is made of the physical time that elapses between successive values.

The following program defines the global constant g and the function bounce that computes the continuous-time position y, the speed y' and an event z for a ball that falls from an initial position yi with an initial velocity yi'. It bounces on the ground every time y crosses zero. ♣

```
let g = 9.81
let hybrid bounce(yi, yi') = (y, y', z) where
  rec der y' = −. g init yi' reset z −> −0.8 *. last y'
  and der y = y' init yi
  and z = up(−. y)
```

The keyword **hybrid** indicates a continuous-time function. The signals y and y' are defined by their derivatives, with y' being reset whenever z is true. The expression up(−. y) define the zero-crossing event l. It detects when −. y crosses zero from a strictly negative to a strictly positive value.

Continuous-time signals are given an ideal interpretation as sequences of values separated by infinitesimal gaps [6]. We follow Suenaga, Sekine, and Hasuo [27], in terming such signals *hyperstreams.* If $\partial \in {}^{\star}\mathbb{R}$ is an infinitesimally small real number, a signal ${}^{\star}yi = ({}^{\star}yi_n)_{n \in {}^{\star}\mathbb{N}}$ defines a value at every instant $n\partial \in {}^{\star}\mathbb{R}$. The base clock is defined as $BaseClock(\partial) = \{n\partial \mid n \in {}^{\star}\mathbb{N}\}$.

For the example, ${}^{\star}yi = ({}^{\star}yi_n)_{n \in {}^{\star}\mathbb{N}}$ and ${}^{\star}yi' = ({}^{\star}yi'_n)_{n \in {}^{\star}\mathbb{N}}$, ${}^{\star}y = ({}^{\star}y_n)_{n \in {}^{\star}\mathbb{N}}$, and ${}^{\star}z = ({}^{\star}z_n)_{n \in \mathbb{N}}$ must satisfy the constraints

$$\begin{aligned}
{}^{\star}y'_0 &= {}^{\star}yi'_0 \\
{}^{\star}y_0 &= {}^{\star}yi_0 \\
\text{and, for } n > 0, \quad {}^{\star}y'_n &= \mathit{if}\ {}^{\star}z_n\ \mathit{then} -0.8(\textbf{last}(y'))_n\ \mathit{else}\ {}^{\star}y'_{n-1} - g\partial \\
{}^{\star}y_n &= {}^{\star}y_{n-1} + {}^{\star}y'_{n-1}\partial \\
{}^{\star}z_n &= \textbf{up}(-y)_n \\
\textbf{up}(-y)_n &= y_{n-2} < 0 \wedge y_{n-1} > 0 \\
\textbf{last}(y')_n &= {}^{\star}y'_{n-1},
\end{aligned}$$

where **last**(y) is the previous value of y. When y is left continuous, it is the left limit of y. The expression **up**(-y) is true when y crosses zero from a strictly positive to a strictly negative value. Let $l, h \in {}^{\star}\mathbb{N}$ be two instants where ${}^{\star}z_l = \mathsf{true}$ and ${}^{\star}z_h = \mathsf{true}$ with ${}^{\star}z_n = \mathsf{false}$ in between ($l < n < h$).

[7] For example, replacing 0 **fby** y by y in filter results in a compile-time error.

Developing the difference equations gives

$$\begin{aligned} {}^\star y'_n &= -0.8{}^\star y'_{l-1} - (n-l+1)g\partial \\ {}^\star y_n &= {}^\star y_l + (\textstyle\sum_{j=l}^{n} {}^\star y'_j)\partial \\ &= {}^\star y_l + \textstyle\sum_{j=l}^{n}(-0.8{}^\star y'_{l-1} + (j-l+1)g\partial)\partial \\ &= {}^\star y_l - 0.8(n-l+1){}^\star y'_{l-1}\partial + ((n-l)(n-l+1)/2)g\partial^2, \end{aligned}$$

where ${}^\star y_n$ is the value of y at instant $n\partial$. The standardization of such signals yields a definition in super-dense time [22, 23]: $t_l, t_h, t \in \mathbb{R}$ such that $l\partial \approx t_l$, $h\partial \approx t_h$ and $n\partial \approx t$. At time t_l, y' has two values (before and after the reset). Let v_l be the value before the reset. The standardization of ${}^\star y$ and ${}^\star y'$ for $t \in [t_l, t_h[$ is

$$\begin{aligned} y'(t) &= 0.8v_l - (t-t_l)g \\ y(t) &= y(t_l, 0) + 0.8(t-t_l)v_l - ((t-t_l)^2/2)g. \end{aligned}$$

The static type system of Zélus enforces a strict separation between streams and hyperstreams. A stream, where there is no notion of duration between successive samples, cannot be used where a hyperstream is expected, and vice versa. A stream can be made a hyperstream by computing it at some zero-crossing events. For example, the sum computes the stream of natural numbers and count_bounces counts the number of ball bounces. ♣

```
let node sum() = o where rec o = (0 fby o) + 1

let hybrid count_bounces(yi, yi') = s where
  rec (y, y', z) = bounce(yi, yi')
  and s = present z -> sum() init 0

let hybrid count_bounces_and_stop(epsilon, yi, yi') = (y, s) where
  rec automaton
    | Bounce ->
        (* z and y' are local to the state *)
        local z, y' in
        do (y, y', z) = bounce(yi, yi')
        and s = present z -> sum() init 0
        until z on (y' < epsilon) then Stop
    | Stop ->
        (* s is unchanged, i.e., implicitly, s = last s *)
        do y = 0.0 done
    end
```

In count_bounces, s is a piece-wise constant signal that is initialized to 0 and updated whenever z is true. This program is valid because the discrete-time function sum in equation **present** z -> sum() **init** 0 is computed at zero-crossing instants and define a (piece-wise constant) hyperstream. Removing the **present** construct would result in a typing error based on the rule: when declaring a function with some kind k (for example, **node** or **hybrid**), all computations must be of the same kind. It is thus possible to compose either stream equations in parallel or hyperstream equations in parallel but not to mix them.

The function count_bounces_and_stop is a variation of count_bounces. It is defined by a two state automaton: in the initial state Bounce, it counts the number of bounces until z **on** (y' < epsilon), that is, when there is both a contact and the speed of the ball is less than epsilon. When this occurs, the next state is Stop in which the position y remains constant at 0.0. The absence of an explicit definition for s in this state means that its value remains constant.

The Zélus compiler first computes type signatures for the functions in a given program. Then, it computes causality signatures which express the input/output dependencies of each function [3]. Finally, it applies an initialization analysis [13] to check that functions do not depend on uninitialized state variables. Only if all static checks succeed does the compiler produce sequential code. For a (discrete) node, it generates a procedure to compute a single step as would any synchronous language compiler. For a hybrid function, it generates three procedures: one that calculates derivatives; another that computes zero-crossing expressions (that is, all expressions passed as arguments to up(.)); and another that is called whenever zero-crossings occur. The first two are passed directly to an ODE solver [10].

Zélus is founded on three principles: an ideal interpretation of time where signals are streams that progress synchronously; systems as functions where the only composition operation is function application, and; hierarchical automata as syntactic sugar. Programs that are not well typed or that cannot be compiled to statically scheduled code are rejected by type-based static analyses. This choice has both positive and negative consequences when defining a library of discrete-time and continuous-time control blocks.

3 THE SIMULINK STANDARD LIBRARY

The Simulink library constitutes the standard toolkit for designing control systems. It contains four kinds of blocks: (1) *combinatorial blocks* whose output values at an instant depend only on the input values at that instant; (2) *discrete-time blocks* that transform discrete-time signals into discrete-time signals; (3) *continuous-time blocks* that transform continuous-time signals into continuous-time signals, and; (4) *discontinuous-time blocks* that convert signals from one class to another or introduce discontinuities.

3.1 Combinatorial blocks

Combinatorial blocks are functions that transform signals into signals and whose outputs at an instant n depend only on the inputs at that instant. Many combinatorial functions, like + and −, are typically implemented in the language targeted by the compiler. ♣

Combinatorial functions can also be defined by users. The compiler checks that their definitions only rely on other combinatorial functions. For example, the function adder computes the one-bit sum by composing two half adders:

```
let half(a, b) = (s, co) where
  rec s = if a then not b else b
  and co = a & b

let adder(c, a, b) = (s, co) where
  rec (s1, c1) = half(a, b)
  and (s, c2) = half(c, s1)
  and co = c1 or c2
```

The compiler responds to this program with

```
val half : bool × bool → bool × bool
val adder : bool × bool × bool → bool × bool
```

Combinatorial function definitions are compiled directly into the target language (OCaml for Zélus or C for Scade Hybrid). The function type -> is shorthand for -A->, where the kind A stands for ‘any’ and indicates that the function is combinatorial. Such functions can be used on both discrete-time and continuous-time signals.

Lookup Tables. These are a typical example of combinatorial functions that are not programmed directly but are rather imported from the target language. The size of the arrays used by the tables must be known statically and is included in the typing constraints of Zélus. Simulink provides lookup tables for one-dimensional functions (1-D Lookup Table; $y = F_1(x)$), two-dimensional ones (2-D Lookup Table; $y = F_2(x_1, x_2)$), and n-dimensional ones (n-D Lookup Table; $y = F(x_1, \ldots, x_n)$). Given l, an integer, and t, an array of length l, lut1D$(l)(t)(x)$ calculates an approximation to $F_1(x)$. Given l_1 and l_2, two integers, t_1, an array of length l_1, and t_2, an array of length l_2, the function lut2D$(l_1)(l_2)(t_1, t_2)(x_1, x_2)$ calculates an approximation to $F_2(x_1, x_2)$. The type signatures are declared in an interface file and the implementation is defined directly in the target language. ♣

$$\mathbf{val}\ \ lut1D\ :\ (l:\ int) \xrightarrow{S} float[l] \xrightarrow{S} float \rightarrow float$$

$$\mathbf{val}\ \ lut2D\ :\ (l1:\ int) \xrightarrow{S} (l2:\ int) \xrightarrow{S} float[l1][l2] \xrightarrow{S} float \times float \rightarrow float$$

The type signature of lut1D indicates that the values of the first two arguments, l and t, must be known at compile time and that the array has l elements. The type signature of lut2D is similar but indicates that the first three values are required at compile time and that the array has two dimensions. The 'S' in the type t_1 –S–> t_2 of a function f signifies that any application $f(e)$ of f must be computable at compile time. The static typing system is not expressive enough to program an n-dimensional lookup table.

Arrays and For Loops. Zélus programs can import arbitrarily complex external functions to manipulate arrays. The language also includes some structured SISAL-inspired [16] constructs that provide a similar expressiveness to the array iterators [24] of Scade. For example, the following function computes the point-wise sum of two arrays of size l: ♣

```
let vsum(l)(x, y) = z where rec
  forall i in 0 .. (l − 1), xi in x, yi in y, zi out z do
    zi = xi + yi
   done
```

The compiler responds to this definition with

$$\mathbf{val}\ \ vsum\ :\ (l\_8 : int) \xrightarrow{S} int[l\_8] \times int[l\_8] \rightarrow int[l\_8]$$

Conceptually, the forall construct composes l copies of the loop body in parallel where they execute synchronously. The index i ranges from 0 to $l - 1$ inclusively. In iteration i, xi stands for $x(i)$, yi for $y(i)$, and zi for $z(i)$. The equation $zi = xi + yi$ thus stands for $z(i) = x(i) + y(i)$ with $i \in [0, l - 1]$, and $z(i)_n = x(i)_n + y(i)_n$ for all instants $n \in \boldsymbol{N}$.

It is possible to accumulate intermediate results. For example, vvproduct$(l)(x, y)$ is the scalar product of two vectors x and y. ♣

```
let vvproduct(l)(x, y) = acc where rec
  forall i in 0 .. (l − 1), xi in x, yi in y do
    acc = (xi *. yi) +. last acc
   initialize
    last acc = 0.0
   done
```

The equation acc = (xi *. yi) +. **last** acc in the loop body stands for

$$\begin{aligned} acc(i) &= (x(i) * y(i)) + acc(i-1) && \text{for } i \in [0, l-1] \\ acc(-1) &= 0, \end{aligned}$$

and so, for all instants $n \in \boldsymbol{N}$ and $i \in [0, l-1]$,

$$\begin{aligned} acc(i)_n &= (x(i)_n * y(i)_n) + acc(i-1)_n \\ acc(-1)_n &= 0. \end{aligned}$$

In iteration i, **last** acc denotes the value computed in iteration $i-1$, that is, $(\textbf{last}\ \mathrm{acc})(i) = acc(i-1)$. Yet, acc is not implemented as an array, but rather as a local variable; outside the loop, the variable acc refers to $acc(l-1)$.

3.2 Discrete-time Blocks

As in any synchronous language, a discrete-time signal represents an infinite stream and discrete systems are functions on synchronous streams. While many discrete-time blocks can be programmed directly in Lustre, others benefit from newer functionality, namely arrays and static higher-order functions. In Zélus, sophisticated features like the generic PID controller are not provided as blocks with many configuration options. Rather, specialized versions are built as (function) compositions of simpler elements. Moreover, we will show that, in Zélus, many continuous blocks are direct extensions of their discrete counterparts.

Unit delay. The z^{-1} operator shifts its input. It is written **pre** x in Lustre and Zélus, which also have operators for initialization, x −> y, and initialized delay, x **fby** y. If $x = (x_i)_{i \in \boldsymbol{N}}$ and $y = (y_i)_{i \in \boldsymbol{N}}$:

(1) $\forall i \in \boldsymbol{N}^*,\ (\textbf{pre}(x))_i = x_{i-1}$ and $(\textbf{pre}(x))_0 = nil$.
(2) $\forall i \in \boldsymbol{N}^*,\ (x\,\textbf{fby}\,y)_i = y_{i-1}$ and $(x\,\textbf{fby}\,y)_0 = x_0$.
(3) $\forall i \in \boldsymbol{N}^*,\ (x \mathbin{\texttt{->}} y)_i = y_i$ and $(x \mathbin{\texttt{->}} y)_0 = x_0$.

where nil denotes the undefined value (a model is only valid when its result does not depend on nil). The Simulink difference and discrete derivative blocks are defined in Zélus as ♣

```
let node diff u = y where rec y = 0.0 −> u −. pre(u)
let node derivative(h)(k, u) = x' where rec x' = diff (k *. u /. h)
```

The compiler responds to these declarations with

$$\begin{aligned} &\textbf{val}\ \ \mathit{diff}\ :\ \mathit{float} \xrightarrow{\mathsf{D}} \mathit{float} \\ &\textbf{val}\ \ \mathit{derivative}\ :\ \mathit{float} \xrightarrow{\mathsf{S}} \mathit{float} \times \mathit{float} \xrightarrow{\mathsf{D}} \mathit{float} \end{aligned}$$

The **node** keyword indicates a discrete-time function: at every instant n, the current output may depend on both the current input and an internal state. The type t_1 −D−> t_2 signifies a stateful function from a stream of type t_1 to a stream of type t_2.

Edge Front Detectors. Several different trigger types are used in standard library blocks. Trigger conditions may occur at *rising* edges, *falling* edges, or *either* type, and also during so called *level* and *level hold* pulses.[8] For boolean signals, the definitions in Zélus are straightforward. ♣

```
(* Resets the state when the reset signal has a rising edge. *)
let node rising_bool(x) = y where
  rec y = false −> not (pre x) & x

let node falling_bool(x) = rising_bool(not x)
let node either_bool(x) = rising_bool(x) or falling_bool(x)
```

These definitions are generalized to floating-point signals. ♣

[8] See https://mathworks.com/help/simulink/slref/discretetimeintegrator.html, with its proviso that *for the discrete-time integrator block, all trigger detections are based on signals with positive values.*

```
let node rising_float(x) = r where
  rec r = (false fby (x < 0.0)) & (x >= 0.0)

let node falling_float(x) = r where
  rec r = (false fby (x > 0.0)) & (x <= 0.0)
```

The explicit comparisons to zero in these definitions are numerically problematic [25], and more complex edge detection functions may sometimes be required.

Reset and Tapped Delays. Both Simulink and Scade provide unit delays of arbitrary length in basic and sliding window forms. They are implemented externally in time complexity $O(1)$ using circular arrays. The delay of length k can be defined directly in Zélus with a forall loop that contains internal state: ♣

```
let node delay_k(k)(v, u) = o where
  rec
    forall i in 0 .. (k − 1) do
      o = v fby (last o)
    initialize
      last o = u
    done
```

Then delay_k(k)(v, u) is the composition of k initialized delays, that is, v **fby** (v **fby** (. . . v **fby** x)). The semantics of the output o is

$$\begin{aligned} o(i) &= v \textbf{ fby } o(i-1) \qquad \text{for all } i \in [0, k-1] \\ o(-1) &= u. \end{aligned}$$

In other words, forall $n \in N$, $i \in [0, k-1]$,

$$\begin{aligned} o(i)_n &= \mathit{if}\ n = 0\ \mathit{then}\ v_0\ \mathit{else}\ o(i-1)_{n-1} \\ o(-1)_n &= u_n. \end{aligned}$$

This implementation has time and space complexity $O(k)$. Consider now the following alternative definition.

```
let node delay_k(k)(v, u) = o where
  rec
    init w = Arrays.const k v
  and
    w = { last w with (i) = u } (* functional update *)
  and
    o = w.((i + 1) mod k)
  and
    i = 0 −> (pre i + 1) mod k
```

It allocates an array w of size k initialized with v. The expression { **last** w **with** (i) = u } defines a new array which is equal to **last** w, the previous value of w, except at index i where it is equal to u. A naive realization of functional update, which also exists in Scade, has a complexity in $O(k)$, but it can be optimized to a single assignment in $O(1)$ because w and **last** w can be stored in the same location [17]. We have not yet implemented this compiler optimization.

The classical sliding window, or tapped delay, is defined below. ♣

```
let node window(k)(v, u) = t where rec
  forall i in 0 .. (k − 1), ti out t do
    acc = v fby (last acc) and ti = last acc
  initialize
    last acc = u
  done
```

It yields $t = [|t_0; \ldots ; t_{k-1}|]$, where for all $n \in \boldsymbol{N}$, $i \in [1, k-1]$,

$$\begin{aligned} acc(i)(n) &= \mathit{if}\ \ n = 0\ \ \mathit{then}\ v_0\ \ \mathit{else}\ acc(i-1)_{n-1} \\ acc(-1)(n) &= u_n \\ t(i)_n &= acc(i-1)_n. \end{aligned}$$

This function can also be programmed using functional updates. The two generalized delay blocks illustrate the trade-off between generating efficient code and minimizing primitive constructs. The k-delay and sliding window blocks are primitives in Scade as they can be efficiently implemented by circular buffers. The less efficient versions that use more basic primitives are still useful for giving a reference semantics and for proving properties. They could replace the custom primitives if the compiler provided more sophisticated optimizations.

Simulink provides various forms of delays with or without resets. In Zélus, the delay reset on a rising edge of r is defined:

```
let node resettable_delay(l)(v, r, u) = o where rec
  reset
    o = delay_k(l)(v)(u)
  every rising_bool(r)
```

This definition illustrates the modularity of Zélus: resettable delays are programmed by composing existing blocks.

Linear Filters. Finite (FIR) and Infinite (IIR) Impulse Response filters are typical examples of unit delay programming in synchronous languages. They are defined by the coefficients of two polynomials in z^{-1}. If U is the input and Y the output, an FIR filter has the form

$$\frac{Y(z)}{U(z)} = \frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{1},$$

and an IIR filter has the form

$$\frac{Y(z)}{U(z)} = \frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{a_0 + a_1 z^{-1} + \cdots + a_N z^{-N}},$$

where M and N are the numbers of coefficients in, respectively, the feedforward and feedback paths.

We give the direct form of these filters with a floating-point input and unit delays initialized to zero. ♣

```
let node fir(m)(b)(u) = y where
  rec t = window(m)(0.0, u) and y = vvproduct(m)(b, t)

let node iir_direct1(n)(m)(b)(a)(u) = y where
  rec y1 = fir(m)(b)(u) and y = y1 −. y2 and y2 = fir(n−1)(a)(0.0 fby y)
```

Discrete Transfer Function. The implementation of a discrete transfer function in Lustre is classical [18]. Unlike Lustre, Zélus allows the expression of a node parameterized over arrays of n numerator and m denominator coefficients. ♣ Its type signature indicates that the first four arguments must be compile-time constants.

$$\textbf{\textit{val}}\ \ \mathit{ztransfer}\ :\ (n\!:\ \mathit{int}) \xrightarrow{S} (m\!:\ \mathit{int}) \xrightarrow{S} \mathit{float}[n] \xrightarrow{S} \mathit{float}[m] \rightarrow \mathit{float} \xrightarrow{D} \mathit{float}$$

Discrete Time Integrators. The Simulink library provides Forward Euler, Backward Euler, and Trapezoidal integrator forms. ♣ Given an input $u = (u_n)_{n \in \mathbb{N}}$, the length of a step T, and a gain K,

Forward Euler: $y_n = y_{n-1} + K * T * u_{n-1}$,
Backward Euler: $y_n = y_{n-1} + K * T * u_n$,
Trapezoidal method: $y_n = y_{n-1} + K * \frac{T}{2} * u_{n-1} + K * \frac{T}{2} * u_n$.

The transfer functions are, respectively, $\frac{T}{z-1}$, $\frac{T\,z}{z-1}$, and $\frac{T}{2}\frac{z+1}{z-1}$. Our definitions translate the recurrence equations directly, rather than call ztransfer with array coefficients. We write x0 for the initial integrator values.

```
(* [output(0) = x0(0)]   and  [output(n) = output(n–1) + (k * t) * u(n–1)] *)
let node forward_euler(t)(k, x0, u) = output where
  rec output = x0 fby (output +. (k *. t) *. u)

(* [output(0) = x0(0)]   and   [output(n) = output(n–1) + i * t * u(n)] *)
let node backward_euler(t)(k, x0, u) = output where
  rec output = x0 -> pre(output) +. k *. t *. u

(* [output(n) = y(n)]   [y(n) = x(n) + (k * t / 2) * u(n)]
 * [x(0) = x0(0)]  and [x(n) = y(n–1) + (k * t / 2) * u(n–1)] *)
let node trapezoidal_fixed(t)(k, x0, u) = output where
   rec x = x0 fby (y +. gain *. u)
   and y = x +. gain *. u
   and gain = k *. t /. 2.0
   and output = y

(* [output(n) = y(n)] and [x(0) = x0(0)] and [x(n) = y(n–1)]
 * [pu(0) = u(0)] and [pu(n) = u(n–1)] and [y(n) = x(n) + (k * t / 2) * (u(n) + pu(n))] *)
let node trapezoidal_variable_step(t)(k, x0, u) = output where
  rec x = x0 fby y
  and y = x +. gain *. (u +. u fby u)
  and gain = k *. t /. 2.0
  and output = y
```

These functions are readily extended to account for upper and lower saturation limits, and reset and output ports. For example, the definition of the extended forward Euler integrator is as follows.

```
type saturation = Upper | Between | Lower

let node forward_euler_complete(t)(upper, lower, res, k, x0, u) = (output, stport, saturation) where
   rec stport = x0 fby (output +. k *. t *. u)
   and v = if res then x0 else stport
   and (output, saturation) =
     if v < lower then lower, Lower else if v > upper then upper, Upper else v, Between
```

The first line declares an enumerated type with three elements (replacing Simulink's 1/0/-1 encoding). The stport output is the Simulink *state port* which yields the value of the integrator to avoid algebraic loops through the reset or initial condition ports.

Discrete Time PID. The transfer function of a proportional-integral-derivative (PID) controller is

$$C_{par}(z) = P + Ia(z) + D\left(\frac{N}{1 + Nb(z)}\right),$$

where P is the proportional gain, I is the integral gain, D is the derivative gain, N sets the location of the pole in the derivative filter, $a(z)$ is the integrator method, and $b(z)$ is the filter method. If dint(h)(k, x0, u) integrates signal k · u with initial value x0 and sampling time h, and dfilter(n)(h)(d, u) computes the filtered derivative of k · u, the parallel PID is defined as follows. ♣

```
let node pid_par(h)(n)(p, i, d, u) = c where
 rec c_p = p *. u
 and i_p = dint(h)(i, 0.0, u)
 and c_d = dfilter(n)(h)(d, u)
 and c = c_p +. i_p +. c_d
```

When there is no filtering, dfilter is simply the derivative:

```
let node dfilter(n)(h)(k, u) = derivative(h)(k, u)
```

Otherwise, it can be replaced by a low-pass filter [2], with filter coefficient n and sample time h:

```
let node dfilter(n)(h)(k, u) = udot where
 rec udot = n *. (k *. u −. f)
 and f = dint(h)(n, 0.0, udot)
```

The signal f must not depend instantaneously on udot, otherwise dfilter cannot be statically scheduled. This precludes instantiating dint with, for example, the Backward Euler or Trapezoidal methods whose current outputs depend instantaneously on their current inputs. This situation is also statically rejected by Simulink.

The PID comes in various forms as it is parameterized by the integration and filter functions, and the latter is itself parameterized by a possibly different integration function. The Simulink graphical interface provides options to specify all possible combinations. Zélus, on the other hand, allows functions to be passed as arguments. We can thus define a generic PID once and for all, as a higher-order function, and instantiate it in different ways, for example, with a forward or backward Euler integration function.

```
let node generic_pid(dint)(dfilter)(h)(p, i, d, u) = c where
  rec c_p = p *. u
  and i_p = run (dint h)(i, 0.0, u)
  and c_d = run (dfilter h)(d, u)
  and c = c_p +. i_p +. c_d

let node pid_forward_no_filter(h)(p, i, d, u) = generic_pid(forward_euler)(derivative)(h)(p, i, d, u)

let node pid_backward_no_filter(h)(p, i, d, u) = generic_pid(backward_euler)(derivative)(h)(p, i, d, u)
```

The keyword **run** in the application **run** (dint h)(i, 0.0, u) tells the type system that dint h is a stateful function. The compiler responds to this definition with:

$$
\begin{array}{l}
\mathbf{val}\ \mathit{generic\_pid}\ :\\
\quad ('a \xrightarrow{S} 'b \times \mathit{float} \times \mathit{float} \xrightarrow{D} \mathit{float}) \xrightarrow{S}\\
\qquad ('a \xrightarrow{S} 'c \times \mathit{float} \xrightarrow{D} \mathit{float}) \xrightarrow{S}\\
\qquad\quad 'a \xrightarrow{S} \mathit{float} \times 'b \times 'c \times \mathit{float} \xrightarrow{D} \mathit{float}\\
\mathbf{val}\ \mathit{pid\_forward\_no\_filter}\ :\\
\quad \mathit{float} \xrightarrow{S} \mathit{float} \times \mathit{float} \times \mathit{float} \times \mathit{float} \xrightarrow{D} \mathit{float}\\
\mathbf{val}\ \mathit{pid\_backward\_no\_filter}\ :\\
\quad \mathit{float} \xrightarrow{S} \mathit{float} \times \mathit{float} \times \mathit{float} \times \mathit{float} \xrightarrow{D} \mathit{float}
\end{array}
$$

The filter function can also be written in a more generic manner:

```
let node generic_filter(dint)(n)(h)(k, u) = udot where
 rec udot = n *. (k *. u −. f)
 and f = run (dint h)(n, 0.0, udot)
```

```
let node filter_forward_100(k, u) = generic_filter(forward_euler)(100.0)(0.1)(k, u)
```

The compiler responds to these definitions with:

$$
\begin{array}{l}
\textbf{val}\ \mathit{generic\_filter}\ : \\
\quad (\mathit{'a} \xrightarrow{S} \mathit{float} \times \mathit{float} \times \mathit{float} \xrightarrow{D} \mathit{float}) \xrightarrow{S} \\
\qquad \mathit{float} \xrightarrow{S} \mathit{'a} \xrightarrow{S} \mathit{float} \times \mathit{float} \xrightarrow{D} \mathit{float} \\
\textbf{val}\ \mathit{filter\_forward\_100}\ :\ \mathit{float} \times \mathit{float} \xrightarrow{D} \mathit{float}
\end{array}
$$

All computations of kind S are evaluated *at compile time* and thus induce no run-time overhead. Static analyses like static typing and causality analysis are performed before static expansion. Then, the compiler applies a source-to-source transformation into code without any further static computations. In the example above, filter_forward_100 is statically expanded into a simple, Lustre-like definition.

Discrete State-Space Representation. The discrete state-space representation of a linear system is

$$
\begin{aligned}
x(n+1) &= Ax(n) + Bu(n) \\
y(n) &= Cx(n) + Du(n),
\end{aligned}
$$

where Ax is matrix-vector multiplication and + is vector addition. Given const(l)(x0), returning a constant vector of n x0 values, mvproduct(l)(m)(b, u), the matrix-vector multiplication of b by u, and sum(l)(x, y), the sum of the two vectors x and y of size l ♣, we can define ♣

```
let node discrete_state_space(l)(m)(r)(x0)(a)(b)(c)(d)(u) = y where
 rec x = const(l)(x0) fby sum(l)(mvproduct(l)(l)(a,x), mvproduct(l)(m)(b,u))
 and y = sum(r)(mvproduct(r)(l)(c,x), mvproduct(r)(m)(d,u))
```

The compiler computes the type signature:

$$
\begin{array}{l}
\textbf{val}\ \mathit{discrete\_state\_space}\ : \\
\quad (\mathit{l\_12} : \mathit{int}) \xrightarrow{S} (\mathit{m\_13} : \mathit{int}) \xrightarrow{S} \\
\qquad (\mathit{r\_14} : \mathit{int}) \xrightarrow{S} \mathit{float} \xrightarrow{S} \mathit{float}[\mathit{l\_12}][\mathit{l\_12}] \xrightarrow{S} \\
\qquad\quad \mathit{float}[\mathit{m\_13}][\mathit{l\_12}] \xrightarrow{S} \mathit{float}[\mathit{l\_12}][\mathit{r\_14}] \xrightarrow{S} \\
\qquad\qquad \mathit{float}[\mathit{m\_13}][\mathit{r\_14}] \xrightarrow{S} \mathit{float}[\mathit{m\_13}] \xrightarrow{D} \mathit{float}[\mathit{r\_14}]
\end{array}
$$

which returns an output y that depends instantaneously on its input u. It is thus more constraining than the Simulink state-space block which provides a special case for when d is all zeros. In this situation, y does not depend instantaneously on its input and can be used in feedback loops. The causality analysis in Zélus (and Scade) does not depend on the values of expressions and is applied before static expansion. Thus it is not possible to have an instantaneous feedback loop between an output of discrete_state_space even if d is all zeros. An additional function must instead be provided for this special case.

Applying static analysis before expansion allows for more precise diagnostic messages and earlier detection of bugs. The disadvantage is that some programs are rejected even though their expansions would not be. Our approach is more demanding on the compiler, since the static analyses must handle higher-order features, but we think that it fits more naturally into the overall philosophy of synchronous languages.

Other discrete blocks are defined similarly. We have not, however, considered an important Simulink feature: it inherits its arithmetic operations from Matlab and they are thus not only overloaded for booleans, integers, and floats, but also for arrays and matrices. Implicit coercions abound; the edge front detectors, for instance, can be applied to numerical values and the integrators and PIDs to arrays of floating-point values. Zélus, like Scade, follows a different approach. It is based on the simple type inference system of ML [26] with no implicit coercion nor overloaded operators.

This discipline avoids ambiguities in interpreting the model and makes compiler correctness simpler to tackle.

3.3 Continuous-time Blocks

Continuous Time Integrators. These blocks have the same input/output interface as their discrete-time counterparts. The simplest version integrates an input signal u with initial position x0. ♣

```
let hybrid int(k, x0, u) = x where
  rec der x = k *. u init x0
```

The keyword **hybrid** indicates that int is a function from continuous-time signals to continuous-time signals and that x is a continuous state variable defined by its instantaneous derivative and initial value. Vector integration is defined as follows.

```
let hybrid vint(n)(x0, u) = x where
  rec forall i in 0 .. (n - 1), x0i in x0, ui in u, xi out x do
          der xi = ui init x0i
       done
```

Simulink allows the state variable of an integrator to be reset to x0 whenever an event res occurs and also provides a state port for breaking algebraic loops. We express the former feature in Zélus with a modular reset and the latter with **last** x, the left limit of x.

```
(* Integration with initial value, reset and state port *)
let hybrid reset_int(k, x0, res, u) = (x, last x) where rec
  reset
    der x = k *. u init x0
  every res
```

Integrators can also be made to saturate at lower and upper limits.

```
let hybrid limit_int(k, y0, upper, lower, r, u) = (y, sat) where rec
  reset
    init y = y0
   and
    automaton
     | BetweenState ->
         (* regular mode. Integrate the signal *)
         do der y = k *. u and sat = Between
         unless up(last y -. upper) then UpperState
         else up(-. (last y -. lower)) then LowerState
     | UpperState ->
         (* when the input [u] is negative *)
         do y = upper and sat = Upper
         unless up(-. u) then BetweenState
     | LowerState ->
         (* when the input [u] is positive *)
         do y = lower and sat = Lower
         unless up(u) then BetweenState
    end
  every r
```

Derivative and Filtered Derivative. The derivative cannot be applied to a continuous-time signal. The following definition is statically rejected by the compiler

```
let hybrid derivative(h)(x) = 0.0 -> (x -. pre(x)) /. h
```

with the error message:

```
File "prog.zls", line 1, characters 30−55:
>let hybrid derivative(h)(x) = 0.0 → (x −. pre(x)) /. h
>                              ^^^^^^^^^^^^^^^^^^^^^^^^^^
Type error: this is a discrete expression and is expected
to be continuous.
```

It must instead be replaced with a linear filter [2]:

```
let hybrid filter(n)(k, u) = udot where
  rec udot = n *. (u −. f) and f = int(k, 0.0, udot)
```

where n is the filter coefficient. This definition is similar to the discrete-time one of Section 3.2; the discrete-time integration is replaced by a continuous-time one.

Continuous Time PID. The continuous-time PID is defined as ♣

```
let hybrid pid_par(int)(filter)(p, i, d, u) = c where
  rec c_p = p *. u
  and i_p = run int(i, 0.0, u)
  and c_d = run filter(d, u)
  and c = c_p +. i_p +. c_d

(* Instantiate the PID with a continuous−time integration function and linear filter. *)
let hybrid pid(n)(p, i, d, u) = pid_par(int)(filter(n))(p, i, d, u)
```

The compiler computes the type signatures:

$$
\begin{array}{l}
\mathtt{val\ pid\_par\ :} \\
\quad \mathtt{('a \times float \times float} \xrightarrow{\mathsf{C}} \mathtt{float)} \xrightarrow{\mathsf{S}} \\
\qquad \mathtt{('b \times float} \xrightarrow{\mathsf{C}} \mathtt{float)} \xrightarrow{\mathsf{S}} \mathtt{float \times 'a \times 'b \times float} \xrightarrow{\mathsf{C}} \mathtt{float} \\
\mathtt{val\ pid\ :\ float} \xrightarrow{\mathsf{S}} \mathtt{float \times float \times float \times float} \xrightarrow{\mathsf{C}} \mathtt{float}
\end{array}
$$

Variants of the continuous PID controller are defined similarly.

The previous definition highlights a limitation of the Zélus type system. It currently forbids writing a single generic PID function that could be instantiated both in discrete-time and continuous-time. That is, the current type system does not permit polymorphism on kinds ($\mathsf{A}, \mathsf{D}, \mathsf{C}$).

Second Order Integrator Block. This block is interesting because it raises an unexpected modularity issue.[9] The normal behavior for the second order integration block is:

$$
\begin{array}{ll}
\dot{x} = y' & \qquad\qquad x(t_0) = x0 \\
\dot{x'} = u & \qquad\qquad x'(t_0) = x0',
\end{array}
$$

where u is the input, $x0$ is the initial position, and $x0'$ is the initial velocity. It is tempting to define this block as the composition of two integrators. This is not possible in Simulink when the position or velocity is limited as the resulting system would have an instantaneous feedback loop. Quoting the documentation:[10]

> *When x is less [resp. greater] than or equal to its lower [resp. upper] limit, the value of x is held at its lower [resp. upper] limit and dx/dt is set to zero. When x is in between its lower and upper limits, both states follow the trajectory given by the second-order ODE.*

In Zélus, this can be achieved by composing two first-order integrators.

[9] See the blog by Guy Rouleau on "How to model a hard stop in Simulink" at: http://blogs.mathworks.com/simulink/2014/01/22/how-to-model-a-hard-stop-in-simulink/.

[10] https://mathworks.com/help/simulink/slref/secondorderintegrator.html

```
let hybrid limit_int2 (xlower, xupper, xlower', xupper', xres, xres', x0, x0', u) = (x, x', xstatus, xstatus')
  where rec
    (x', xstatus') = limit_int(1.0, x0', xlower', xupper', xres', fu)
  and
    (x, xstatus) = limit_int(1.0, x0, xlower, xupper, xres, x')
  and
    fu = match xstatus with | Between −> u | Upper | Lower −> 0.0
```

To ensure the invariant that x' is the derivative of x, the filtered input fu is set to zero when x is constant, that is, when xstatus is either Above or Below. Despite the apparent cyclic dependency between the two calls to the function int, there is no instantaneous loop: fu depends on xstatus which does not depend instantaneously on x'.

State Space. The continuous state-space block resembles the discrete-time one, the unit delay is replaced by an integrator: ♣

$$\begin{aligned} \dot{x} &= Ax + Bu & \qquad x(t_0) &= x0 \\ y &= Cx + Du. \end{aligned}$$

Its definition is:

```
let hybrid state_space(n)(m)(r)(x0)(a)(b)(c)(d)(u) = y where
  rec
    x = vint(n)(const(n)(x0), sum(n)(mvproduct(n)(n)(a, x),  mvproduct(n)(m)(b, u)))
  and
    y = sum(r)(mvproduct(r)(n)(c, x),  mvproduct(r)(m)(d, u))
```

The compiler computes the type signature:

$$\begin{array}{l} \textit{val}\ \ \textit{state\_space}\ : \\ \quad (n\_21 : int) \xrightarrow{S} (m\_20 : int) \xrightarrow{S} (r\_22 : int) \xrightarrow{S} float \xrightarrow{S} \\ \qquad\qquad float[n\_21][n\_21] \xrightarrow{S} float[m\_20][n\_21] \xrightarrow{S} \\ \qquad\qquad\qquad float[n\_21][r\_22] \xrightarrow{S} float[m\_20][r\_22] \xrightarrow{S} float[m\_20] \xrightarrow{C} float[r\_22] \end{array}$$

This definition suffers the same limitation as the discrete-time version: even if d is all zeros, y appears to depend instantaneously on u. Again, a specialized version is required to handle this case.

3.4 Discontinuous Blocks

The discontinuous module includes blocks for coulomb and viscous friction, quantization, saturation, relays, hit crossings, backlashes, deadzones, rate limiting, and wrapping to zero.

Those blocks are simple to define in discrete-time ♣ but they raise several difficulties when applied to continuous-time inputs. We discuss them in this section. ♣

Coulomb and Viscous Friction. When the gain and offset are scalars (floating-point signals), the definition is:

```
let hybrid coulomb(gain, offset, x) = y where
  rec y = sgn(x) *. (gain *. abs_float (x) +. offset)
```

The function sgn(x) returns -1 if x is strictly negative, 1 if it is strictly positive, and 0 otherwise. Since it introduces discontinuities, it must be programmed using zero-crossing detection. ♣ SIMULINK provides several forms of the coulomb block. For example, x can be a scalar value and offset a vector.

```
let hybrid coulomb_scalar_input(n)(gain, offset, x) = y where
  rec
    g = gain *. abs_float (x)
  and
```

```
    forall i in 0 .. (n − 1), offseti in offset, yi out y do
      yi = sgn(x) *. (g +. offseti)
    done
```

or, the input and the offset can both be vectors.

```
let hybrid coulomb_vectors(n)(gain, offset, x) = y where
  rec forall i in 0 .. (n − 1), xi in x, offseti in offset, yi out y do
      yi = sgn(xi) *. (gain *. abs_float (xi) +. offseti)
    done
```

Quantization. The quantizer block produces a staircase function:

```
let hybrid quantizer(q, u) = n where
  rec init n = q *. floor(u /. q) and nq = n *. q
  and present up(u −. nq −. q) −> do n = last n +. 1.0 done
            | up(nq −. u) −> do n = last n −. 1.0 done
```

Again, since floor is discontinuous, zero-crossings are required to program quantization in Zélus.

Saturation. The saturation block returns the value of an input if it is within given bounds, and otherwise the upper or lower bound:

```
let hybrid saturation(upper, lower, u) = min (max lower u) upper
```

No zero-crossing is needed here.

Relay. The relay block switches between two values and can be defined as a two-state automaton. In the initial mode On, it returns the value von unless the input u becomes lower than soff; in mode Off, it returns voff unless it becomes greater than son.

```
let hybrid relay(son, soff, von, voff, u) = r where
  rec automaton
  | On −> do r = von unless up(soff −. u) then Off
  | Off −> do r = voff unless up(u −. son) then On
  end
```

Hit crossing. This block detects when a signal crosses zero. It has three main variants: negative to positive only, positive to negative only, or either direction. It is tempting to first try a simple version that returns true when u is positive and false otherwise:

```
let hybrid positive(u) = (u >= 0.0)
```

But the Zélus compiler complains:

```
File "prog.zls", line 1, characters 25−35:
>let hybrid positive(u) = (u >= 0.0)
>                         ^^^^^^^^^^
Type error: this is a stateless discrete expression and
is expected to be continuous.
```

Zélus forbids any comparison on a continuous-time signal which could lead to a boolean change during integration. Zero-crossing detection must be used instead. In this second attempt, the output is true when u changes from negative to positive, false when it changes from positive to negative, and unchanged otherwise.

```
let hybrid positive(u) = present up(u) −> true | up(−. u) −> false init false
```

Unfortunately, this solution is problematic: **up**(u) detects a *true* zero-crossing, that is, a change in sign during integration as provided by SUNDIALS [21, §2.4] and based on the Illinois method [14]. It does not detect a signal that was strictly negative and then remains at zero or, conversely, a signal that rests at zero and then becomes strictly positive.[11] SIMULINK, on the other hand, incorporates these two cases into the zero-crossing definition and implementation. Though it is of clear practical interest and could be implemented in ZÉLUS, its mathematical definition is unclear and its implementation can lead to fragile models.[12] An alternative is to use a threshold.

```
type pos = Positive | Negative | Between

let hybrid compare(epsilon, x) = pos where rec
  automaton
  | BetweenState  -> do pos = Between
                     unless up(x -. epsilon) then PositiveState
                     else up(-. epsilon -. x) then NegativeState
  | PositiveState -> do pos = Positive unless up(-. x) then NegativeState
  | NegativeState -> do pos = Negative unless up(x) then PositiveState
  end
```

Dead Zone. This block[13] returns 0.0 when its input u is between a lower limit ll and an upper limit ul and otherwise shifts the result.

```
let hybrid dead_zone(ll, ul, u) = y where
 rec automaton
   | BetweenState ->
     do y = 0.0 unless up(u -. ul) then PositiveState
     else up(ll -. u) then NegativeState
   | PositiveState ->
     do y = u -. ul unless up(ul -. u) then BetweenState
   | NegativeState ->
     do y = u -. ll unless up(u -. ll) then BetweenState
   end
```

Backlash. This block models play between two elements. Quoting the documentation:[14]

A system with play can be in one of three modes: (1) Disengaged — In this mode, the input does not drive the output and the output remains constant. (2) Engaged in a positive direction — In this mode, the input is increasing (has a positive slope) and the output is equal to the input minus half the deadband width. (3) Engaged in a negative direction — In this mode, the input is decreasing (has a negative slope) and the output is equal to the input plus half the deadband width.

Let u be the input, width the width of the play, and y0 the initial output position. The function backlash needs to test for a change in the derivative of u, but this cannot be modeled directly since it would require the computation of the derivative. We instead require that the derivative u' of u be passed explicitly.

```
let hybrid backlash (width, y0, (u, u')) = y where
 rec half_width = width /. 2.0
 and init y = y0
 and automaton
   | Disengaged ->
     do unless up(u -. (last y +. half_width)) then Engaged_positive
     else up( -.u +. (last y -. half_width)) then Engaged_negative
```

[11] In such situations, SUNDIALS raises an error.

[12] Consider a quadratic signal x whose root is at time $t = 1$ and simultaneous with another event, which means that the simulation stops at that instant. SIMULINK finds a zero-crossing for x at $t = 1$ which is not seen if the other event is absent.

[13] https://mathworks.com/help/simulink/slref/deadzone.html

[14] https://mathworks.com/help/simulink/slref/backlash.html

```
| Engaged_positive ->
  do y = u -. half_width unless up(-. u') then Disengaged
| Engaged_negative ->
  do y = u +. half_width unless up(u') then Disengaged
end
```

Either u' is passed to the block, approximated with a sampled version of the derivative, or replaced by a linear filter.

3.5 Undefinable Continuous-time Blocks

Several blocks cannot be defined as continuous-time blocks in Zélus, even though they are easily defined in discrete-time ♣, as they explicitly rely on the internal steps performed by the solver: namely, the memory block, the derivative (and thus the modular form of backlash), the rate delimiter, and the transport delay. Any definition of these blocks inevitably refers to the mechanics of simulation. We do not know how to formalize their meaning solely in terms of their input and output traces. This obstructs compositional reasoning: their precise behavior can be influenced by any other block in the same model.

The Memory Block. In discrete time, the memory block is the unit delay. In continuous time, it is the value of the input signal at the previous major step and constant in minor steps.[15] It is used for breaking algebraic loops but gives rise to fragile models [6]. The Simulink reference manual advises not to use this block with certain solvers or if its inputs change 'during simulation'. On the contrary, our **last**$(\cdot)$ operator is the left limit of its input signal and does not explicitly refer to the simulation engine. Its use is constrained by the causality analysis [3]. It is less flexible than the memory block but more predictable.

The Derivative and Rate Delimiter. The derivative block computes an approximation of the derivative that explicitly relies on major time steps.[16] Neither Zélus nor Simulink performs symbolic computation on signals to compute their derivatives. The rate delimiter[17] is another example that explicitly refers to the internal steps of the simulation engine. These two blocks can easily be defined from the memory block and they have the same weakness. Their semantics is not defined solely in terms of their inputs but depends on the whole model.

Time and Transport Delay. The *time* and *transport delays* delay an input signal by some amount. They come in various forms but all of them rely on the internal discrete steps of the solver. They thus cannot be defined directly in Zélus. The ideal definition for a signal $u : \mathbb{R} \to V$ is simply:

$$\text{transport\_delay}(i)(\tau)(x)(t) = x(t - \tau) \qquad \text{if } t \geq \tau, \text{ and } i \text{ otherwise.}$$

But this definition is not operational since only a finite set of samples of x are known during simulation—typically those computed at the minor and major steps of the solver. A buffer is used in the actual implementation to store the values that are computed at the major steps of the simulation; values at times greater than τ are interpolated linearly, and those at times less than τ are extrapolated.

For the four 'undefinable' continuous functions, it is possible to define well-typed versions that compute approximations at discrete instants. For example, the following function delays its input x by one step according to the input event z. It returns the value that the input x had at the previous

[15] https://mathworks.com/help/simulink/slref/memory.html

[16] https://mathworks.com/help/simulink/slref/derivative.html

[17] https://mathworks.com/help/simulink/slref/ratelimiter.html

instant where z was true and remains constant between such instants. The output is initialized with i.

```
let hybrid delay(i)(z, x) = y where
  rec y = present z -> (i fby x) init i
```

The type signature is

$$\textbf{val}\ \ delay\ :\ \ 'a \xrightarrow{S} zero \times\ \ 'a \xrightarrow{C}\ 'a$$

The input i and x, the output y have the same type 'a. z is of type zero. E.g., this block can be instantiated by giving for z a periodic timer. E.g., delay(0)(**period** (0.1), x) models a discrete-time delay of 0.1 second. It defines the piece-wise constant integer signal y such that for all $t \in [0, 0.1[$, $y(t) = 0$ and for all $n \in \mathbb{N}^*$, $y(0.1(n + 1) + t) = x(0.1n)$.

3.6 Importing and Exporting Functions

Practical hybrid systems models must be able to import and run external stateful functions, for example, functions written in C or generated from another language. Simulink s-functions[18] and the FMI standard[19] are typical representations for stateful functions. Ptolemy[20] allows discrete-time and continuous-time stateful functions to be imported from these two representations. Scade allows discrete-time stateful functions to be imported directly from C.

Since Simulink, Zélus, and Scade Hybrid only consider causal models, they cannot express DAEs but only ODEs. Scade Hybrid allows the import and export of stateful functions with both discrete-time and continuous-time state variables and complies with the FMI/FMU standard [10]. FMUs for co-simulation are imported as nodes and FMIs for model-exchange are imported as hybrid functions. It is thus possible to program models in Scade Hybrid that compose (1) several FMI and/or FMUs that can be discrete-time, continuous-time, or hybrid, some being produced by languages based on DAEs (Modelica, Simplorer) together with (2) a software controller written in SCADE. The static checks performed by Scade Hybrid—like the type checking, causality, and initialization analyses—are applied as per usual on the overall model. These checks reject some bad combinations[21] that would be silently accepted otherwise and lead to non-deterministic/non-reproducible simulation results.

Zélus provides a more limited capability that allows the importing of stateful functions directly written in OCaml.

4 DISCUSSION AND RELATED WORK

Our work is related to several projects and tools. The Ptolemy environment already contains a comprehensive set of control blocks. A reference semantics [29] shows that every (elementary or composed) block can be defined as four atomic transition functions. For any given composition, a *director* defines the overall four transition functions from those of the individual blocks. Zélus follows a different but complementary approach to Ptolemy by defining and implementing a hybrid systems modeler using synchronous language principles only, and by adopting the point-of-view of Lustre where function application is used to compose programs and static constraints to reject programs. The semantics and compilation techniques of Zélus are adapted directly from those of existing synchronous language compilers.

A wide range of tools target the formal verification of hybrid systems, see [1] for references. The input languages of these tools are non-deterministic and designed for expressing properties of

[18] https://mathworks.com/help/simulink/create-cc-s-functions.html

[19] https://www.fmi-standard.org

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

[21] An example is an instantaneous feed-back loop on a discrete-time variable.

models rather than for programming executable models that mix control software and a continuous-time model of the environment. The formal verification of synchronous programs, and of Scade in particular, has been extensively studied [7]. The formal verification of Zélus programs has not yet been specifically addressed.

Several works address the semantics of hybrid system modelers, and, in particular, the simulation engine of Simulink [9] and the subset of discrete time blocks [19, 20]. We already noted the work of Caspi et al. [28] where Lustre is applied to define a semantics for a subset of Simulink. The present paper follows the same path initiated by Caspi, in the sense that the synchronous definition of a block, whether in discrete-time or continuous-time, gives a formal and executable semantics.

A comprehensive definition of discrete-time blocks has been completed for Scade in parallel to this work [15]. Several of the continuous blocks have been programmed in Scade Hybrid too and we continue defining blocks with Zélus.

The experiment described in this paper revealed several limitations in the previous version of Zélus and motivated several language extensions. For-loops, arrays, static parameters, and higher-order functions were added to the compiler to make the definition of blocks more generic. Loops *à la* SISAL [16] appear to integrate well into the data-flow style of Zélus. Higher-order functions are an original feature of Zélus that require significant extensions to static type inference, and the causality and initialization analyses which were limited to first-order functions in the previous version of the compiler. These analyses are applied on source programs prior to static expansion. To generate code, we choose the simple solution of "templates" which consists in expanding static computations, in particular, for higher-order function applications. The definition of a function of type t_1 −S−> t_2 does not generate code directly but is specialized as many times as it is applied with different values.

Several questions remain to be addressed. We are not yet satisfied with the treatment of zero-crossings on signals that reach zero or depart from zero without changing sign immediately. The current definition of **up**(x) in Zélus and Scade Hybrid only detects when x goes from a strictly negative to a strictly positive value during integration but not when signal reaches or leaves zero. This conforms with the mathematical definition and implementation of zero-crossings in the Illinois method [14], but is probably too constraining for real models. We have experimented in Scade Hybrid with a type system that distinguishes signals based on whether they are constant, smooth (C^∞), or discontinuous during integration. The type system statically ensures that zero-crossing detection is only used in the cases described above.

Finally, some blocks that are defined in terms of major steps cannot be written in the current version of Zélus without introducing a zero-crossing. An example is the "wrapping state" mechanism of the integrator[22] that limits a continuous state variable y, forcing it to stay between lower and upper bounds. This wrapping is not done on a zero-crossing, that is, by testing that the state variable has reached a bound, nor during integration, but rather at a major steps, that is, the disjunction of all the events in the system. It is thus normal for y to exceed the given bounds during integration. If z denotes the disjunction of all events, we have:

```
let hybrid f(lower)(upper)(x0, z, u) = y where
  rec der y = u init x0
          reset z -> last y -. (upper -. lower) *. (floor ((last y -. lower) /. (upper -. lower)))
```

It is tempting to simply provide access to major steps of the simulation engine, for example, by providing a function:

```
val major_step : unit → zero
```

[22] https://mathworks.com/help/simulink/slref/integrator.html

but using such a function without any additional constraints would make models unsafe and their semantics difficult to define modularly.

Zélus and Scade Hybrid conservatively extend a synchronous language in the sense that a stream function (a 'node') is compiled as usual into a sequential step function that runs in bounded time and space. Typical implementations execute this function periodically on a real-time clock whose value must be greater than the worst-case execution time (WCET). For a hybrid model, the compiler generates code for the simulation that is paired with an ODE solver. The very same generated code for stream functions—in particular, the software part—is kept and so, the code used for the simulation is the one used on the target platform. Yet, in the simulated model, a stream function produces its output in zero time whereas its implementation on the target platform definitely takes time. To make the simulation faithful to what is executed on the platform, execution times and transmission delays must also be expressed in the model. For a hybrid model, the simulation is off-line: it uses a variable-step ODE solver and the actual duration between two successive zero-crossings is unknown. As a consequence, an on-line (real-time) simulation, with a different solver and an inevitable approximation of zero-crossing occurrences would give different results. One possible way to address this problem would be to apply a source-to-source transformation to turn hybrid models, or subsets of them, into synchronous stream functions, by internalizing the integration scheme and replacing zero-crossings with edge front detectors.

5 CONCLUSION

We show that it is possible to directly program many of the standard blocks used to model discrete-time and continuous-time control systems and their environments. We do this by combining a minimal set of primitives and the main features of a synchronous language extended with ODEs and zero-crossings.

We took the Simulink standard library as a reference for our experiment as it is representative of typical control blocks and is challenging in term of expressiveness, semantics, type checking, and code generation. Provided with the ability to program most blocks, users are less dependent on an existing library and can extend it according to their needs. This approach also gives an executable and reference semantics to several blocks.

Certain applications are currently simulated in one tool and then manually reprogrammed in a synchronous language for certified compilation. Enriching synchronous languages with hybrid blocks for testing and implementing control software potentially removes an important gap in the development chain.

Acknowledgments. This work was inspired by a discussion with Mary Sheeran who posed the question "would it be possible to make a functional language version of Simulink?".

REFERENCES

[1] Rajeev Alur. 2011. Formal Verification of Hybrid Systems. In *Proceedings of the 9th ACM International Conference on Embedded Software (EMSOFT).* ACM, Taipei, Taiwan, 273–278.

[2] Karl J. Åström and Richard M. Murray. 2008. *Feedback Systems: An Introduction for Scientists and Engineers.* Princeton University Press, NJ, USA.

[3] Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet. 2014. A Type-based Analysis of Causality Loops in Hybrid Systems Modelers. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control (HSCC).* ACM, Berlin, Germany, 71–82.

[4] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. 2011. A Hybrid Synchronous Language with Hierarchical Automata: Static Typing and Translation to Synchronous Code. In *Proceedings of the 9th ACM International Conference on Embedded Software (EMSOFT).* ACM, Taipei, Taiwan, 137–147.

[5] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. 2011. Divide and recycle: types and compilation for a hybrid synchronous language. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers,*

Tools and Theory for Embedded Systems (LCTES). ACM, Chicago, USA, 61–70.

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

[7] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (Jan. 2003), 64–83.

[8] Gérard Berry. 1989. Real time programming: Special purpose or general purpose languages. *Information Processing* 89 (1989), 11–17.

[9] Olivier Bouissou and Alexandre Chapoutot. 2012. An operational semantics for Simulink's simulation engine. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*. ACM, Beijing, China, 129–138.

[10] Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. 2015. A Synchronous-based Code Generator For Explicit Hybrid Systems Languages. In *Proceedings of the 24th International Conference on Compiler Construction (CC) (Lecture Notes in Computer Science)*. Springer, London, UK, 69–88.

[11] Timothy Bourke and Marc Pouzet. 2013. Zélus, a Synchronous Language with ODEs. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control (HSCC)*. ACM, Philadelphia, USA, 113–118.

[12] Luca Carloni, Maria D. Di Benedetto, Alessandro Pinto, and Alberto Sangiovanni-Vincentelli. 2004. *Modeling Techniques, Programming Languages, Design Toolsets and Interchange Formats for Hybrid Systems.* Technical Report. IST-2001-38314 WPHS, Columbus Project.

[13] Jean-Louis Colaço and Marc Pouzet. 2004. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)* 6, 3 (August 2004), 245–255.

[14] M. Dowell and P. Jarratt. 1972. A modified regula falsi method for computing the root of an equation. *BIT Numerical Mathematics* 11, 2 (June 1972), 168–174.

[15] Esterel Technologies SAS 2016. *Gateway Guidelines for Simulink.* Esterel Technologies SAS.

[16] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. 1990. A report on the Sisal language project. *J. Parallel and Distrib. Comput.* 10 (1990), 349–366.

[17] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. 2012. A Modular Memory Optimization for Synchronous Data-Flow Languages. Application to Arrays in a Lustre Compiler. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*. ACM, Beijing, 51–60.

[18] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Dataflow Programming Language LUSTRE. *Proc. IEEE* 79, 9 (September 1991), 1305–1320.

[19] Grégoire Hamon. 2005. A denotational semantics for Stateflow. In *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT)*. ACM, Jersey City, NJ, USA, 164–172.

[20] Gregoire Hamon and John Rushby. 2004. An Operational Semantics for Stateflow. In *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE) (Lecture Notes in Computer Science)*, Vol. 2984. Springer, Barcelona, Spain, 229–243.

[21] Alan C. Hindmarsh and Radu Serban. 2016. *User Documentation for CVODE v2.9.0* (v2.9.0 ed.). Lawrence Livermore National Laboratory, Livermore, CA, USA.

[22] Edward A. Lee and Haiyang Zheng. 2005. Operational Semantics of Hybrid Systems. In *Proceedings of the 8th International Conference on Hybrid Systems: Computation and Control (HSCC) (Lecture Notes in Computer Science)*, Vol. 3414. Springer, Zurich, Switzerland, 25–53.

[23] Oded Maler, Zouar Manna, and Amir Pnueli. 1992. From Timed to Hybrid Systems. In *Proceedings of the REX Workshop Real-Time: Theory in Practice (Lecture Notes in Computer Science)*, Vol. 600. Springer, Mook, The Netherlands, 447–484.

[24] Lionel Morel. 2007. Array Iterators in Lustre: From a Language Extension to Its Exploitation in Validation. *EURASIP Journal on Embedded Systems* 2007, 1 (2007), article 059130.

[25] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. 2010. *Handbook of Floating-Point Arithmetic.* Birkhäuser, Basel, Switzerland.

[26] B. C. Pierce. 2002. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA.

[27] Kohei Suenaga, Hiroyoshi Sekine, and Ichiro Hasuo. 2013. Hyperstream Processing Systems: Nonstandard Modeling of Continuous-time Signals. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 417–430.

[28] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. 2005. Translating Discrete-Time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems* 4 (2005), 779–818. Issue 4. Special Issue on Embedded Software.

[29] Stavros Tripakis, Christos Stergiou, Chris Shaver, and Edward A. Lee. 2013. A modular formal semantics for Ptolemy. *Mathematical Structures in Computer Science* 23, 4 (008 2013), 834–881.