Zélus: A Synchronous Language with ODEs

Tool Paper

Timothy Bourke
NICTA, Sydney
INRIA Paris-Rocquencourt
DI, École normale supérieure
Timothy.Bourke@ens.fr

Marc Pouzet
Univ. Pierre et Marie Curie
DI, École normale supérieure
INRIA Paris-Rocquencourt
Marc.Pouzet@ens.fr

ABSTRACT

ZÉLUS is a new programming language for modeling systems that mix discrete logical time and continuous time behaviors. From a user's perspective, its main originality is to extend an existing LUSTRE-like synchronous language with Ordinary Differential Equations (ODEs). The extension is conservative: any synchronous program expressed as data-flow equations and hierarchical automata can be composed arbitrarily with ODEs in *the same source code*.

A dedicated type system and causality analysis ensure that all discrete changes are aligned with zero-crossing events so that no side effects or discontinuities occur during integration. Programs are statically scheduled and translated into sequential code that, by construction, runs in bounded time and space. Compilation is effected by source-to-source translation into a small synchronous subset which is processed by a standard synchronous compiler architecture. The resultant code is paired with an off-the-shelf numeric solver.

We show that it is possible to build a modeler for explicit hybrid systems *à la Simulink/Stateflow* on top of an existing synchronous language, using it both as a semantic basis and as a target for code generation.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Language Classifications—*Data-flow languages*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Continuous, Discrete Event*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers*

General Terms

Algorithms, Languages

Keywords

Hybrid systems; Hybrid automata; Synchronous languages; Block diagrams; Type systems;

HSCC'13, April 8–11, 2013, Philadelphia, Pennsylvania, USA.

1. INTRODUCTION

Hybrid systems modelers are used not only in the high-level design and simulation of complex embedded systems, but also as development platforms in which the same source is used for formal verification, testing, simulation, and the generation of target executables. The quintessential example is the SIMULINK/STATEFLOW suite,[1] but there are also LABVIEW,[2] MODELICA,[3] and several others [6].

These tools are distinguished from the formal model of hybrid automata [14] by their focus on modular programming and simulation of both physical models and their controllers. In this context, reproducible, efficient simulations and the ability to generate statically scheduled code are essential features. As a major consequence, programming constructs are typically deterministic whereas hybrid automata are essentially non-deterministic and oriented toward specification and formal verification through the over-approximation of piecewise continuous behaviors.

The underlying mathematical model of hybrid modelers is the synchronous parallel composition of stream equations, differential equations, and hierarchical automata. But even with a carefully chosen numeric solver, actual simulations only ever approximate the ideal behaviors of such models. A formal semantics exists for discrete subsets [13, 20], but mixes of discrete and continuous-time signals often have unpredictable and mathematically weird behaviors [2, 3]. For instance, a continuous STATEFLOW state may be triggered repeatedly if a transition guard remains true, and, although transitions are instantaneous, the amount of simulation time that elapses between triggerings is non-zero and depends on when the solver decides to stop, which in turn is influenced by simulation parameters, global numerical error, and the occurrence of other zero-crossings. The behavior of a model may even change if it is placed in parallel with an independent block, for example one that tests the zero-crossings of a sinusoid signal. In this case, changing the sinusoid frequency may radically change the output of the other model. This time leak is not due to numerical artifacts but to a more fundamental reason: discrete time is not logical but global, it exposes the internal steps of the simulation engine.

Synchronous languages [5] differ from this approach by providing a logical notion of time, independent of a physical implementation. For example, the LUSTRE equations

$$\mathsf{x} = 0 \rightarrow \mathsf{pre}\ \mathsf{y} \quad \text{and} \quad \mathsf{y} = \mathsf{if}\ \mathsf{c}\ \mathsf{then}\ \mathsf{x} + 1\ \mathsf{else}\ \mathsf{z}$$

[1] http://www.mathworks.com/products/simulink/
[2] http://www.ni.com/labview/
[3] http://www.modelica.org/

define the two sequences $(x_n)_{n\in I\!N}$ and $(y_n)_{n\in I\!N}$ which are computed sequentially with $x_0 = 0$, $x_n = y_{n-1}$, and for all $n \in I\!N$, $y_n = \text{if } c_n \text{ then } x_n + 1 \text{ else } z_n$.[4] Time is logical, that is, nothing can be inferred about the actual time that passes between instants i and $i+1$. Synchronous programs see the environment as a source of input and output sequences and ignore intervening gaps. They are thus only suitable for the design of discrete controllers. In contrast, a model expressed with Ordinary Differential Equations (ODEs) or Differential Algebraic Equations (DAEs) continues to evolve during such gaps and a variable-step numeric solver is necessary to approximate continuous trajectories efficiently and faithfully.

So, what is the best way to combine the precision of synchronous languages for programming discrete components with the extra expressiveness afforded by ODEs approximated by numeric algorithms? Any combination must be *conservative*: the behavior of a synchronous program should not change if ODEs are placed in parallel, it should have the same logical-time semantics and compile to the same code; in particular to avoid inconsistencies between simulation and execution. Furthermore, discrete computations and side effects should not occur during numerical approximation.

In order to avoid the aforementioned time leak and to have a clear separation between discrete and continuous-time signals, we proposed the following convention, quoting [3]:

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

This means that any synchronous program can be paired with ODEs as long as it is activated on a discrete clock. The definition is sufficiently general to model, for example, a discrete controller activated on a timer (periodic or not) or a deterministic hybrid automata with dynamic conditions.

Previously, we proposed the basis of a LUSTRE-like language extended with ODEs and following the above discipline. We defined the semantics of a minimal language using non-standard analysis [4], and proposed a type system that ensures the absence of time leaks, and a compilation method [3]. We later added hierarchical automata [2]. These techniques have been implemented in a new language, called ZÉLUS, which allows arbitrary combinations of data-flow equations, hierarchical automata, and ODEs. A type system and causality analysis statically ensure that discrete computations are aligned with zero-crossings. Compilation is by source-to-source translation into synchronous code which is then compiled to sequential code and paired with an off-the-shelf numeric solver. This paper describes key aspects of the language and implementation. We use synchronous programming as both a semantical foundation, where we are strongly influenced by the work of Lee et al. [17] and Mosterman et al. [11, 19], and as a target for code generation.

2. AN OVERVIEW OF ZÉLUS

ZÉLUS[5] is a first-order synchronous dataflow programming language extended with resettable ODEs and hierarchical automata. Rather than define the abstract syntax (available in [2, §3.1]), we show its main features through examples.

[4]The unit delay initialized to 0, $0 \rightarrow \text{pre}(.)$, is $\frac{1}{z}$ in SIMULINK.

[5]Available at `http://www.di.ens.fr/~pouzet/zelus`.

A ZÉLUS program is a sequence of definitions. The following declares a discrete function that counts the occurrences of a Boolean v and detects when there have been n:

```
let node after (n, v) = (c = n) where
  rec  c = 0 → pre(min(tick, n))
  and tick = if v then c + 1 else c
```

where $\text{pre}(\cdot)$ is the non-initialized unit delay, $. \rightarrow .$ is the initialization operator of LUSTRE, min computes the minimum of its arguments and if/then/else is the mux operator that executes both branches and takes the value of one. The semantics in terms of infinite sequences is

$$\forall i \in I\!N^*, c_i = min(tick_{i-1}, n_{i-1}) \text{ and } c_0 = 0$$
$$\forall i \in I\!N, tick_i = \text{if } v_i \text{ then } c_i + 1 \text{ else } c_i,$$
$$\forall i \in I\!N, (after(n, v))_i = (c_i = n_i)$$

and the compiler infers the signature:

```
val after : int × bool ⇒ bool
```

This node can be used in a two state automaton,

```
let node blink (n, m, v) = x where
  automaton
  | On → do x = true until (after(n, v)) then Off
  | Off → do x = false until (after(m, v)) then On
```

which returns a value for x that alternates between true for n occurrences of v and false for m occurrences of v. The keyword until stands for a *weak* preemption, that is, x equals true when after(n, v) is true and becomes false the following instant. The semantics and compilation of automata, defined in [8], is that of SCADE 6[6] and LUCID SYNCHRONE.[7] The blinking behavior can be reset on a boolean condition r by nesting it inside a one state automaton that tests r, which we write using the reset/every syntactic sugar:

```
let node blink_reset (r, n, m, v) = x where
  reset
    automaton
    | On  → do x = true until (after(n, v)) then Off
    | Off  → do x = false until (after(m, v)) then On
  every r
```

The type signatures inferred by the compiler are:

```
val blink : int × int × bool ⇒ bool
val blink_reset : bool × int × int × bool ⇒ bool
```

Up to syntactic details, these ZÉLUS programs could have been written *as is* in SCADE 6 or LUCID SYNCHRONE.

But ZÉLUS goes beyond discrete dataflow programming and allows the definition of continuous-time variables. For instance, consider a boom turning on a fixed pivot. The boom's angle can be expressed as a differential equation with initial value i and derivative v using the der keyword:

```
  der angle = v init i
```

Its (ideal) value at model time t is:

$$\text{angle}(t) = i(0) + \int_0^t v(x)\, dx,$$

It is compiled into a continuous state variable whose value is approximated by a numeric solver as described in §3.2.

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

[7]`http://www.di.ens.fr/~pouzet/lucid-synchrone`

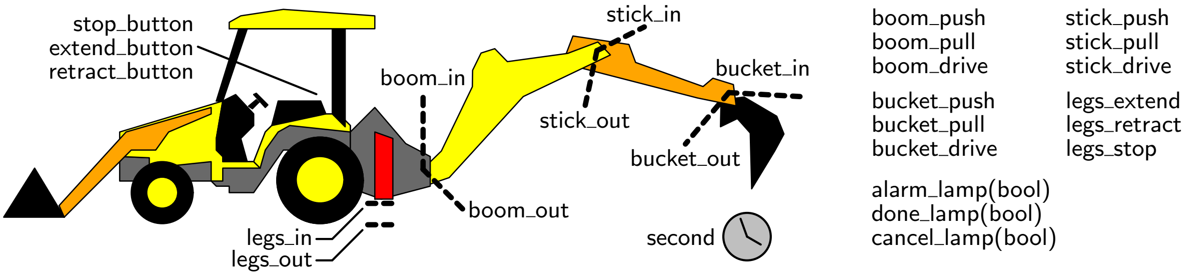

Figure 1: Idealized Backhoe Loader model

Now, if we wanted to achieve a reference velocity v_r using Proportional-Integral (PI) control, we need only add three equations in parallel:[8]

```
... and error = v_r −. v
    and der v = 0.7 ∗. error +. 0.3 ∗. z init 0.0
    and der z = error init 0.0
```

It could be that we also want to reset the controller state z when the angle reaches or exceeds a limit max. Two new constructs are needed: a way of detecting such interesting events and a way of directly setting the value of a continuous state. The standard way to detect events in a numeric solver is via zero-crossings where a solver monitors expressions for changes in sign and then, if they are detected, searches for a more precise instant of crossing. We introduce an up(.) operator to monitor an expression for a (rising) zero-crossing; with this, the definition of z becomes:

```
der z = error init 0.0 reset up(angle −. max) → 0.0
```

which says to reset the value of z to 0.0 the instant when angle − max becomes zero or positive. A der definition defines two values: a state (z) initially and in response to discrete events, and its derivative ($\dot{z}$) during continuous phases.

More complicated behaviors are better described as automata where defining equations and events being monitored change depending on mode. For instance, if the direction of the boom's motion changes in response to the input signals push and pull, and if the boom becomes stuck when it reaches a limit of motion, we may define v_r as follows:

```
automaton
| Pushing → do v_r = maxf
                until up(angle −. max) then Stuck
                else  pull(_) then Pulling
| Pulling → do v_r = −. maxf
               until up(min −. angle) then Stuck
               else  push(_) then Pushing
| Stuck → do v_r = 0.0 done
```

The automaton construct is effectively an equation and each mode contains a set of definitions which, naturally, may themselves include derivatives and automata. Here, the value of v_r is defined as either maxf, −maxf, or 0.

Transitions are ordered by priority and their guards have type α signal; events of type α signal are emitted by discrete components or introduced by up(.) : float ⇝ unit signal. As a consequence of this typing rule, boolean expressions cannot serve directly as guards in continuous automata. While it would be possible to compile an expression up(e : bool) into up(if e then 1.0 else −1.0), as effectively occurs in Stateflow,

[8] −., ∗., +. are floating-point arithmetic operators.

```
let hybrid segment ((min, max, i), maxf, (push, pull, go))
      = ((segin, segout), angle) where

 rec  der angle = v init i
 and error = v_r −. v
 and der v = (0.7 /. maxf) ∗. error +. 0.3 ∗. z init 0.0
                 reset hit(v0) → v0
 and der z = error init 0.0 reset hit(_) → 0.0
 and v_r = if go then rate else 0.0
 and (segin, segout) = (angle <= min, angle >= max)

 and automaton
    | Stuck →
      do rate = 0.0
      until push() on (not segout) then Pushing
      else  pull() on (not segin) then Pulling

    | Pushing → local atlimit in
      do rate = maxf and atlimit = up(angle −. max)
      until atlimit() on (last v > 0.3 ∗. maxf) then
             do emit hit = −0.8 ∗. last v in Pushing
      else atlimit() then Stuck
      else pull() then Pulling

    | Pulling → local atlimit in
      do rate = −. maxf and atlimit = up(min −. angle)
      until atlimit() on (last v < −0.3 ∗. maxf) then
             do emit hit = −0.8 ∗. last v in Pulling
      else atlimit() then Stuck
      else push() then Pushing
```

Figure 2: Model of segment (boom, stick, or bucket)

the search for zero-crossings then degenerates into binary search, and, more unsatisfying still, boolean complement, like signal absence, is not closed on discrete signals. Ultimately, we think we can better analyze and execute hybrid programs by restricting the form of triggering expressions.

One of Zélus's strengths is the way that larger models can be constructed using abstraction and instantiation. For example, the various fragments discussed so far are readily combined into a more interesting model: that of the idealized backhoe loader that we use to teach discrete reactive programming. A labeled screen capture from its graphical simulator is shown in Figure 1. Using input signals from buttons and *_in/*_out sensors, and the outputs listed at right, students must write a discrete controller to operate the three backhoe segments. The simulator must, of course, approximate the movement of the segments.

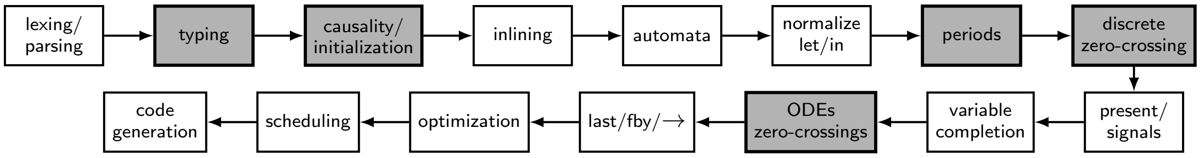

Figure 3: Compiler architecture

The node declaration for a single segment is shown in Figure 2. It declares a hybrid node called segment taking three inputs: a triple of movement parameters (min, max, i), the maximum force maxf, the control signals (push, pull, go); and giving as output the sensor values (segin, segout) and the segment position, angle.

The body of segment combines the elements already discussed with some minor modifications. For one, the reference velocity v_r is 0 when go is false and rate otherwise. The value of rate depends on the direction of motion, which in reality is determined by a hydraulic valve but which we model as a hybrid automaton. The automaton differs in its initial state, but also because of the self-loop transitions that model bouncing at the limits of motion:

```
until (atlimit() on (last v > 0.3 *. maxf)) then do ...
```

The operator on : α signal $\times$ bool $\rightsquigarrow$ α signal filters signals through boolean expressions. When atlimit occurs and the expression is satisfied, the transition resets the value of v and emits the signal hit. We are obliged to write last v in the guard and do/in equations to respect causality: the value of v cannot be tested before it is defined! Semantically last v is the left-limit of v. The hit signal resets the controller integrator, z, so that the sudden spikes on v are ignored.

A complete simulation is constructed as the parallel composition of instantiations of the segment node (three times: boom, stick, and bucket), a similar node that moves the legs, a function for updating the visualization, and a node implementing a discrete controller. Whereas a hybrid node like segment may be executed repeatedly to approximate continuous states, activations of the visualization function and controller node must be triggered more conservatively: the former because it has side-effects (it draws in a window); the latter because it may update internal discrete states. For instance, we call the update function with

```
present period (0.1) →
  showupdate (leg_pos, boom_ang, stick_ang, bucket_ang,
              alarm_lamp, done_lamp, cancel_lamp)
```

3. COMPILER ARCHITECTURE

Our starting point in developing a compiler for Zélus was to recycle existing synchronous techniques so that a language like Scade could be extended without disturbing its existing semantics and compilation. After a year of trial and error, we arrived at the architecture depicted in Figure 3. Zélus is implemented in OCaml;[9] the size of each stage is given in Table 1. The compiler is a pipeline of stages that only aborts early if one of the front-end passes fails:

1. Parsing turns the program into an abstract syntax tree.
2. Typing annotates expressions with types (refer §3.1).

Compiler	*LOC*
Main driver (incl. main data structures)	1769
Abstract syntax and pretty printers	767
Lexer & parser	1002
Typing	1696
Initialization and causality analysis	839
Transformation of hybrid features	974
Transformation of automata	337
Transformation of synchronous features	940
Inlining and other optimizations	597
Code generation	852
Runtime	
Simulation algorithm	317
Solver interface (generic)	340
Solver interface (Sundials, compiler specific)	200
Zero-crossing detection (Illinois)	151

Table 1: Zélus in numbers

3. The causality analysis checks for causality loops (refer §3.1). Then, the initialization analysis checks for reads from uninitialized delays [9]; ODEs are readily treated.

After the front-end stages, a series of source-to-source transformations are applied, each yielding a valid program.

4. 'Small' functions are inlined as an optimization.
5. Each automaton is replaced with a pair of switch-like statements [8]; ODEs remain unchanged.
6. Local declarations are un-nested to simplify later steps. For example, let $x = ($let $y = e_1$ in $e_2)$ in e_3 is transformed into let $x = e_2$ and $y = e_1$ in e_3.
7. A primitive exists for periodic clocks, like period 0.2(3.4) which has a phase of 0.2 and a period of 3.4 and is mathematically equivalent to z and the sawtooth s:

   z = up(s)   der s = 1.0 init −0.2 reset z → −3.4

   Nonetheless, this direct translation with its costly continuous state and zero-crossing is avoided in favor of an output that returns the next horizon to the solver.
8. Each hybrid function is augmented with a boolean flag go to signal when a weak transition has occurred and thus that a subsequent discrete reaction is required.
9. The present and emit constructs are replaced, respectively, by a switch statement and the pairing of a value with an enable bit [7].
10. ODEs and up(.) operators are removed (refer §3.2).

(At this point, the code is purely synchronous.)

11. All last, fby, and → operators are replaced by pre delays.

[9]http://caml.inria.fr

12. Simple optimizations occur: dead-code removal, elimination of copy variables and common sub-expressions.
13. Equations are statically scheduled according to data-flow dependencies.
14. The code is modularly translated into sequential code.

The architecture is mainly that of the Lucid Synchrone compiler[10] and only the highlighted boxes are really novel. We detail them in the following sections.

3.1 Typing and Causality

As the backhoe example demonstrates, Zélus allows liberal combinations of combinational, discrete, and continuous elements. Nevertheless, discontinuities and side-effects must only occur on discrete clocks; programs that violate this rule are rejected. The principles and formal rules underlying the type system of Zélus are presented elsewhere [2, 3]; here we focus on the pragmatic motivations and implementation.

Every function definition, equation or expression is associated to a kind $k \in \{\mathsf{A}, \mathsf{D}, \mathsf{C}\}$: A if combinatorial, D if discrete-time, and C if continuous-time. Kinds are related by the minimal subtyping relation such that $\mathsf{A} \leq \mathsf{D}$ and $\mathsf{A} \leq \mathsf{C}$. They are ascribed to entire 'blocks' rather than to individual 'wires'—each node or set of equations has a single kind which is inherited by individual inputs and outputs. The system extends naturally to automata: all the states of a 'continuous' automaton are also of kind C and may thus contain ODEs. The signature of a function f with input type t_1 and output type t_2 is thus of the form:

$$f : \forall \beta_1, ..., \beta_n . t_1 \xrightarrow{k} t_2 \quad \text{(where the } \beta_i \text{ are type variables)}$$

(We write $\xrightarrow{\mathsf{A}}$ as $\rightarrow$, $\xrightarrow{\mathsf{D}}$ as $\Rightarrow$, and $\xrightarrow{\mathsf{C}}$ as $\rightsquigarrow$). This block-based scheme greatly simplifies the formal rules, implementation, and type-related messages (interface files and errors) and so far we have not found it hinders writing programs.

A combinational function is defined by writing:

```
let square(x) = x *. x
```

Its inferred type is float → float. A declaration with the keyword node gives a function that executes in discrete time and which may thus contain unit delays, and the keyword hybrid gives a function that executes in continuous time.

As an example of the typing analysis, consider a program that tries to place an ODE in parallel with a discrete counter:

```
let hybrid wrong(x) = o where
  rec der o = 1.0 init −1.0
  and cpt = 0.0 fby cpt +. o
```

The compiler rejects it with the message:

```
File "ex.ls", line 3, characters 12−25:
>  and cpt = 0 fby cpt +. o
>            ^^^^^^^^^^^^^
Type error: this is a discrete expression and is
expected to be continuous.
```

As wrong is defined with keyword hybrid, it must not contain a discrete-time computation which is not triggered on a discrete clock. It could be made valid by writing, for example,

```
let hybrid good(x) = o where
  rec der o = 1.0 init −1.0 reset z() → −1.0
  and cpt = present z() → (0 fby cpt + o) init 0
  and z = up(last o)
```

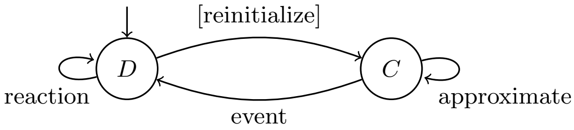

Figure 4: The basic structure of a hybrid simulation

After typing, a causality analysis is performed to ensure the absence of instantaneous loops. It follows two simple principles: every loop on a discrete signal must be broken by a unit delay and every loop on a continuous-time signal must be broken by an integrator. This is essentially what existing tools like Simulink do.

3.2 Compilation of ODEs and Runtime

The typing and causality analyzes help fulfill three fundamental requirements: *(a)* to simulate continuous processes with state-of-the-art numeric solvers, *(b)* to import existing synchronous code without modification, and, *(c)* to use existing tools to compile everything.

A hybrid simulation alternates between two phases as depicted in Figure 4. In state D, some code *next* is executed,

$$y', go' = next(y, \vec{x}),$$

to compute the next values of the discrete state variables y and go from the current value of y and a continuous state $\vec{x}$. This computation occurs when any of the zero-crossings $z_1, ..., z_n$ or go are true. Side effects and changes of state variables (continuous or discrete) may occur at such instants. A simulation iterates in this state, in which physical time does not progress, until go becomes false. Then, integration in a numeric solver begins. A solver takes functions f_y and g_y parameterized by y and a horizon h with:

$$\dot{\vec{x}} = f_{y,h}(t, \vec{x}) \qquad \vec{z} = g_{y,h}(t, \vec{x})$$

and approximates $\vec{x}(t_i + h)$ from the current time t_i while monitoring the elements of $\vec{z}$ for changes of sign. The solver stops when it reaches time $t_i + h$ or when one or more zero-crossings has been detected. State D is then entered and the code is executed and may respond to detected events.

It is up to the compiler to construct *next*, *f*, *g*, the discrete state y, and the continuous state $\vec{x}$. This is done by the source-to-source transformations which turn hybrid functions into discrete nodes with additional inputs and outputs as shown in this extract from the compilation of der z = ... and up(angle −. max) in segment:

```
let node segment((iz1, iz2), (lz, lv, langle), d,
          (min, max, i), maxf, (push, pull, go)) =
  ((upz1, upz2), (nz, nv, nangle),
   go1 or go2 or go3, ((segin, segout), angle))
  ...
  and lastz = 0.0 → lz
  and dz = error
  and match hit with
      | (_, true) → do go1 = true and z = 0.0 done
      | _ →          do go1 = false and z = lastz done
  and nz = if d then z else dz
  ...
  and atlimit = iz1 and upz1 = angle −. max
  ...
```

[10] Through it has been rewritten entirely.

Each $\mathsf{up}(e)$ is replaced with a boolean input $\mathsf{iz}i$ to signal detection of the corresponding zero-crossing and a floating-point output $\mathsf{upz}i$ to transmit the value of the expression e. The equation $\mathsf{der\ z = error\ init\ 0.0\ reset\ hit(\_)} \rightarrow 0.0$ is translated into $\mathsf{dz = error}$, a match statement on the concrete representation of the signal hit (when hit is present z is set directly to 0.0), and equation $\mathsf{nz = if\ d\ then\ z\ else\ dz}$, where d is a boolean flag that is true in D and false in C, and nz is an output. The lastz variable replaces all occurrences of last z, whether implicit or explicit, and lz is an input that contains the value of the state variable z estimated by the solver (an element of $\vec{x}$). This synchronous function is then compiled to sequential code.

Once the source program has been compiled into an executable, it is possible to choose the numeric solver and zero-crossing detection algorithm, and to set their parameters from the command-line. We have implemented a modular framework based on OCaml functors and first-class modules to integrate solvers. In addition to an interface to the SUNDIALS CVODE solver [15], we have implemented several numeric solvers and the 'Illinois' false position method for zero-crossing detection using standard techniques [10] (Butcher tables, Hermite interpolation, and error estimation).

We have experimented with various examples from the literature, including the sticky masses [16, Example 6.8] and air traffic control [21], and from SIMULINK, including the bang-bang temperature controller and clutch model.

4. COMPARISON WITH OTHER TOOLS

The ZÉLUS language is most distinguished from SIMULINK by the type system that regulates compositions of discrete and continuous elements and the compilation by source-to-source transformation. While the basic semantics of a SIMULINK model follow simple principles, the behavior of important corner-cases can really only be understood by careful operational reasoning, and such intricacies must be faithfully reproduced by compilers and analysis tools. Users intending to compile their models into controllers are advised to avoid certain features, or to favor others; for instance, to use function call triggers to explicitly determine the execution order of blocks. In contrast, ZÉLUS has a consistent and simple semantics, the same source-to-source translations generate code for simulation and for embedded targets. Only the last step, that turns basic dataflow assignments into imperative code, requires customization for specific targets.

ZÉLUS shares the same basic model of executions, that alternate between continuous phases and sequences of 'run-to-completion' discrete actions, as specification frameworks like Charon [1], SpaceEx [12], and Hybrid I/O Automata [18], and like them, concerns itself with modular composition. The biggest difference is the use of synchronous, rather than interleaved, parallelism, which enforces a strong discipline on communication through shared variables, which we consider as clocked streams, and on causality, since the language ensures a single value per variable per instant. Furthermore, we insist on externalizing all non-determinism from models.

Acknowledgments

We warmly thank Cyprien Lecourt for his work on solver modules and the development of various examples.

5. REFERENCES

[1] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in CHARON. In *HSCC'00*, pages 6–19, 2000.

[2] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. A Hybrid Synchronous Language with Hierarchical Automata: Static Typing and Translation to Synchronous Code. In *EMSOFT'11*, Taiwan, Oct. 2011.

[3] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *LCTES'11*, USA, Apr. 2011.

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

[5] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1), Jan. 2003.

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

[7] J.-L. Colaço, G. Hamon, and M. Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. In *EMSOFT'06*, South Korea, Oct. 2006.

[8] J.-L. Colaço, B. Pagano, and M. Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *EMSOFT'05*, USA, Sept. 2005.

[9] J.-L. Colaço and M. Pouzet. Type-based initialization analysis of a synchronous data-flow language. *J. Software Tools for Technology Transfer*, 6(3):245–255, Aug. 2004.

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

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

[12] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *23rd Conf. CAV*, pages 379–395, USA, July 2011.

[13] G. Hamon. A denotational semantics for Stateflow. In *EMSOFT'05*, pages 164–172, 2005.

[14] T. Henzinger. The theory of hybrid automata. *NATO ASI Series F: Comp. & Systems Sciences*, 170:265–292, 2000.

[15] A. Hindmarsh, P. Brown, K. Grant, S. Lee, R. Serban, D. Shumaker, and C. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Soft.*, 31(3):363–396, Sept. 2005.

[16] E. A. Lee and P. Varaiya. *Structure and Interpretation of Signals and Systems*. http://LeeVaraiya.org, second edition, 2011.

[17] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT'07*, Austria, Sept. 2007.

[18] N. A. Lynch, R. Segala, and F. W. Vaandrager. Hybrid I/O automata. *Info. & Comp.*, 185(1):105–157, Aug. 2003.

[19] P. Mosterman, J. Zander, G. Hamon, and B. Denckla. Towards computational hybrid system semantics for time-based block diagrams. In *3rd IFAC Conf. Analysis & Design of Hybrid Sys.*, pages 376–385, Spain, Sept. 2009.

[20] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In *EMSOFT'04*, pages 259–268, 2004.

[21] C. Tomlin, G. J. Pappas, and S. Sastry. Conflict resolution for air traffic management: A study in multiagent hybrid systems. *IEEE Trans. Automatic Control*, 43(4):509–521, Apr. 1998.