A Synchronous Look at the Simulink Standard Library
or
Can we design a functional Simulink? [1]

Marc Pouzet
Marc.Pouzet@ens.fr

DI, ENS

SYNCHRON
Bamberg
December 7, 2016

[1] Joint work with T. Bourke (INRIA Paris), F. Carcenac, JL. Colaço, B. Pagano, C. Pasteur (Esterel-Tech., SCADE Core)

Trends for building safe and complex software

Write executable mathematical specifications in a high-level programming language so that the source is:

A reference semantics independent of any implementation.

A basis for simulation, testing, formal verification.

Compiled into executable code, sequential or parallel

with semantics preservation all along the chain.

A way to achieve correct-by-construction software

so that

"what you simulate/prove is what you execute" (Berry, 89)

Typed Functional Languages: λ-calculus + types

A computation is a sequence of reductions:

$$fact(3) \rightarrow 3 \times fact(2) \rightarrow 3 \times 2 \times fact(1) \rightarrow 3 \times 2 \times 1 \rightarrow 3 \times 2 \rightarrow 6$$

Abstract implementation details to focus on what computes the function.

Only few orthogonal principles:

- function composition;
- inductive data-types, pattern matching;
- types to specify/ensure simple invariants.

The code is safer, smaller and it is faster to get it right.

Examples: Haskell, OCaml, SML, Agda, Coq, etc.

An important vehicule of ideas for formal methods in industry (e.g., Esterel-Tech, Microsoft, Facebook) and general purpose languages (e.g., F#, Swift, Rust)

Synchronous Languages: the beautiful idea of Lustre

A discrete-time system is a stream function; streams evolve synchronously

X	1	2	1	4	5	6	...
$pre(X)$	*nil*	1	2	1	4	5	...
$X - pre(X)$	*nil*	1	-1	3	1	1	...

The equations $Z = X + Y$ means $\forall n \in \mathbb{N}, Z_n = X_n + Y_n$

Make time logical and abstract from impl. details, focus on the function.

Only few orthogonal principles:

- infinite streams, function composition;
- restrict the expressiveness to generate bounded memory/time code.

A solid ground for PL extensions: higher-order, arrays, automata, etc.

SCADE KCG 6 incorporates most of them in a conservative manner.

Hybrid Systems Modelers

Program complex discrete systems and their physical environments in a single language

Edward Lee and Haiyang Zheng (HSCC, 2005):

> *Hybrid modeling languages are best viewed as programming languages that happen to have a hybrid systems semantics*

Many tools and languages exist

- PL: Simulink/Stateflow, LabVIEW, Scicos, Ptolemy, Modelica, etc.
- Verif: SpaceEx, Flow*, dReal, etc.

Focus on Programming Language (PL) issues to improve safety

- Pionneering work of Edward Lee's group on Ptolemy.
- Yet, can we program hybrid systems in a purely functional manner?

`Zélus`, a synchronous language with ODEs [HSCC'13]

An experiment to write hybrid systems with a purely functional language

Milestones

- A conservative extension of `Lustre` with ODEs [LCTES'11]
- A synchronous non-standard semantics [JCSS'12]
- Hierarchical automata, both discrete and hybrid [EMSOFT'11]
- Causality analysis [HSCC'14]; code generation [CC'15]

SCADE Hybrid at Esterel-Tech/ANSYS (2014 - 2015)

- A validation into the industrial KCG compiler of SCADE
- Prototype based on KCG 6.4 (now 6.6)
- SCADE Hybrid = full SCADE + ODEs
- Import/export FMI/FMUs 2.0; model-exchange FMUs (Simplorer)

Distribution

Information on the language (binaries, reference manual, examples):

`http://zelus.di.ens.fr`

`Zélus` source code is available on a private svn server.

svn: `https://svn.di.ens.fr/svn/zelus/trunk`

The SundialsML binding is available on OPAM (source code):

`http://inria-parkas.github.io/sundialsml/`

First prototype in 2011. Current version is 1.2.

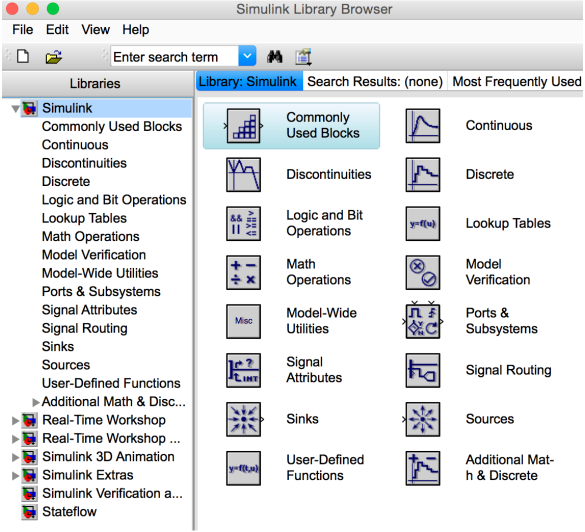
Experimental version: higher-order functions, static values, arrays.

Yet, is that enough to program real applications, e.g., those written in Simulink?

A Simpler Objective

Can we program the Simulink standard library so that the source is the formal specification and turned into sufficiently efficient sequential code?

The Simulink Standard Library



Combinational Blocks

Essentially Lustre: data-flow equations with combinatorial functions.

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

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

val half : bool * bool -A-> bool * bool
val adder : bool * bool * bool -A-> bool * bool
```

The type $t_1 \xrightarrow{A} t_2$ means that $f(x)$ is executed at every instant.

Other "mathematical blocks" are written similarily.

Combinatorial Blocks

Look up Tables are more interesting examples.

Typically programmed in the host language (e.g., C, Matlab).

Yet, the size of the array is statically fixed.

```
val lut1D : (l: int) -S-> float array[l]
                     -S-> float -A-> float

val lut2D : (l1: int) -S-> (l2: int)
                      -S-> float array[l1]  float array[l2]
                      -A-> float  float -A-> float

val lutnD : (k: int) -S-> (l: int) -S-> float array[l][k]
                     -S-> float array[k] -A-> float
```

A function f with type $t_1 \xrightarrow{S} t_2$ means that $f(x)$ is statically evaluated.

Arrays and Loops

The loop construct is borrowed from the SISAL language. The expressiveness is equivalent to that of SCADE iterators.

```
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

val vsum(l:int) -S-> (int[l] * int[l]) -A-> int[l]
```

The equation zi $=$ xi $+$ yi means for all $i \in [0..l-1]$:

$$z(i) \quad = \quad x(i) + y(i)$$

That is for all $i \in [0..l-1]$, for all $n \in \mathbb{N}$:

$$z(i)_n \quad = \quad x(i)_n + y(i)_n$$

Accummulator

```
let node scalar(l)(x, y) = acc where
  rec forall i in 0 .. l - 1, xi in x, yi in y
        do
          acc = (xi * yi) + lastit acc
        initialize
          init acc = 0.0
        done

val scalar : (l: int) -S-> float array[l] * float array[l]
                      -A-> float
```

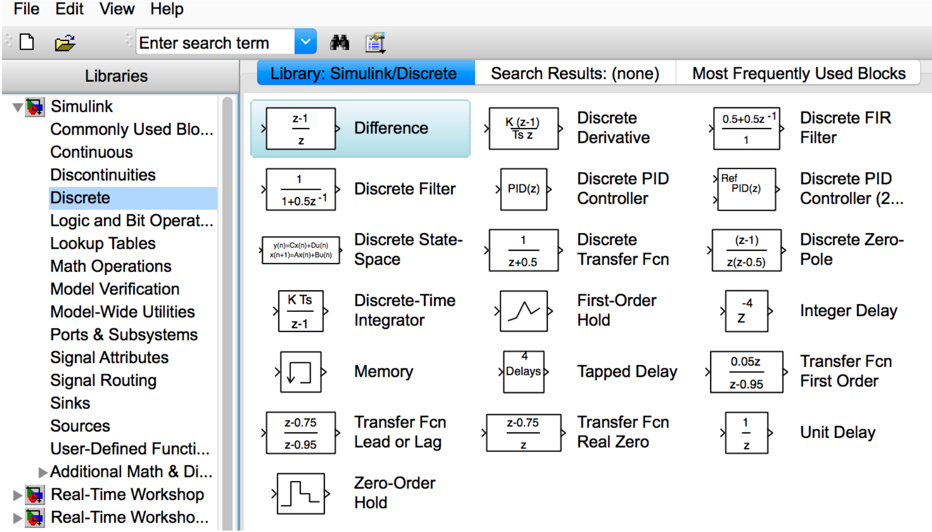
The equation acc = (xi ∗. yi) +. lastit acc stands for:

$$\begin{array}{rcl} acc(i) & = & (x(i) * y(i)) + acc(i-1) \text{ with } i \in [0..l-1] \\ acc(-1) & = & 0 \end{array}$$

and so, for all $n \in \mathbb{N}$ and $i \in [0..l-1]$:

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

Discrete Blocks



Unit Delay

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

Composing delays with a loop

```
(* k-length delay. Complexity in O(k) *)
let node delay_k(k)(v)(u) = o where
  rec forall i in 0 .. k - 1 do
        o = v fby (lastit o)
      initialize
        init o = u
      done
```

that is, forall $n \in \mathbb{N}$, $i \in [0..k-1]$, $n \in \mathbb{N}$:

$$\begin{array}{lcl} o(i)_n & = & (v\ \mathbf{fby}\ o(i-1))_n = \mathit{if}\ \ n = 0\ \mathit{then}\ v_0\ \mathit{else}\ o(i-1)_{n-1} \\ o(-1)_n & = & u_n \end{array}$$

Delays, Tapped delays (sliding window)

```
(* a k-delay in O(1) *)
let node delay_k(k)(x0)(u) = o where
  rec
    init w = Array.create k v
  and
    w = { last w with (i) = u }
  and
    o = w.((i + 1) mod k)
  and
    i = 0 -> (pre i + 1) mod k

(* sliding window in O(k) *)
let node window(k)(v)(x) = t where
  forall i in 0 .. k - 1, ti out t
    do
      acc = v fby (lastit acc) and t_i = acc
    initialize
      init acc = x
    done
```

Discrete-time blocks: the Integrator

```
type saturation = Between | Lower | Upper

(* forall n in Nat.
 * [output(0) = x0(0)]
 * [output(n) = output(n-1) + (h * k) * u(n-1)] *)
let node forward_euler(x0, k, h, u) = output where
  rec output = x0 fby (output +. (k *. h) *. u)

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

Discrete-time PID

Transfer function:

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

```
(* PID controller in discrete time
 * p is the proportional gain;
 * i the integral gain;
 * d the derivative gain;
 * n the filter coefficient *)
 let node pid_par(p)(i)(d)(h)(n)(u) = c where
   rec c_p = p *. u
   and i_p = int(h)(i)(0.0, u)
   and c_d = filter(n)(h)(d *. u)
   and c = c_p +. i_p +. c_d
```

int is the integration function; filter is the filtering function.

When there is no filtering, the definition of filter is simply the derivative:

```
let node filter(n)(h)(u) = derivative(h)(u)
```

Otherwise, approximate using a linear low pass filter:

```
(* n is the filter coefficient;
 * h is the sampling time *)
 * transfer function is [N / (1 + N b(z))]
 * [n = inf] means no filtering *)
let node filter(int)(n)(h)(u) = udot where
  rec udot = n *. (u -. f)
  and f = int(h)(0.0, udot)
```

A Generic Discrete-time PID

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

let node pid_forward_no_filter(p)(i)(h)(n)(u) =
   generic_pid(euler_forward)(derivative)(p)(i)(h)(n)

let node pid_forward(p)(i)(h)(n)(u) =
  generic_pid(euler_forward)(filter(euler_forward))
   (p)(i)(h)(n)
```

Discrete blocks

- Most blocks can be programmed in a `Lustre`-like style with stream equations and a reset.
- The program is very close to the mathematical specification.
- The causality analysis, that computes the input/output relation of a node, is very helpful to understand which feebacks are possible.

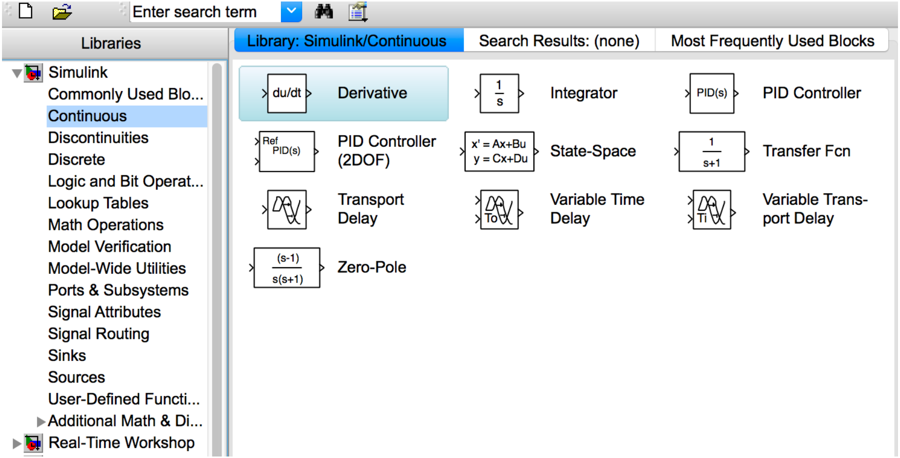
Yet, Simulink provides features `Zélus` does not have: overloading of operators (+ applies to integers, floats, complex, vectors, matrices, etc.).

Well, nothing so surprising here.

Several tools automatically translate a subset of Simulink discrete-time blocks into `Lustre`.

They do not define precisely what can and cannot be encoded. How to ensure that they are "correct"?

Continuous Blocks



Continuous-time Integrator

```
(* Integration with initial value *)
let hybrid int(x0, u) = x where
  rec der x = u init x0

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

Integration with limit

```
(* initial condition [x0] with threshold [lower] and [upper] *)
let hybrid limit_int(y0, upper, lower, r, u) = (y, sat) where
  rec init y = y0
  and reset
        automaton
        | Between ->
            (* regular mode. Integrate the signal *)
          do der y = u reset r -> y0 and sat = Between
          unless up(y -. upper) then Upper
          else down(y -. lower) then Lower
        | Upper ->
          (* when the speed [u] is negative *)
          do y = upper and sat = Upper
          unless down(u) then Between
        | Lower ->
          (* when the speed [u] is positive *)
          do y = lower and sat = Lower
          unless up(u) then Between
        end
      every r
```

Derivative and Filtered derivative

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

This program is statically rejected.

The filtered derivative is:

```
(* Derivative. Applied on a linear filtering of the input
 * n is the filter coef. [n = inf] would mean no filtering.
 * transfer function is [N s / (s + N)] *)
let hybrid filter(n, f0, u) = udot where
  rec udot = n *. (u -. f)
  and f = int(f0, udot)
```

Continuous-time PID

The continuous time PID is now written

```
(* PID controller in continuous time
 * p is the proportional gain;
 * i the integral gain;
 * d the derivative gain;
 * n the filter coefficient *)
let hybrid pid_par(p)(i)(d)(n)(u) = c where
   rec c_p = p *. u
   and i_p = int(i)(0.0, u)
   and c_d = filter(n)(d *. u)
   and c = c_p +. i_p +. c_d
```

The structure of the code is very similar to that of the discrete-time case.

Second Order Integrator Block

The regular behavior for the second order integration block is:

$$\begin{aligned} \dot{x} &= y' & x(t_0) &= x0 \\ \dot{x}' &= u & x'(t_0) &= x0' \end{aligned}$$

Simulink's documentation:

When x is less than [resp. higher] or equal to its lower [resp upper] limit, the value of x is held at its lower [resp. lower] 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.

Simulink provides a special block as it is not possible to write it by composing too first order integrators. [2] Quoting the documentation: [3]

[2] See the blog "modeling a hard stop in Simulink".

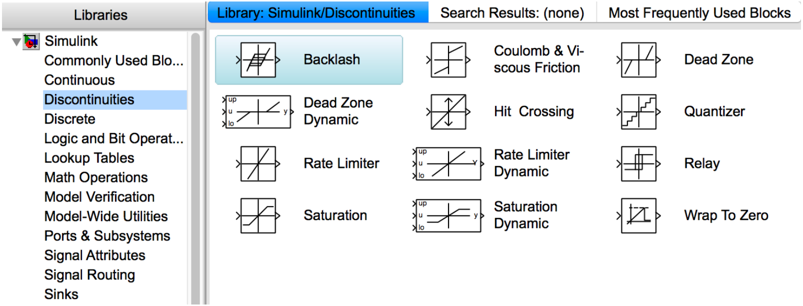
[3] `https://fr.mathworks.com/help/simulink/slref/secondorderintegrator.html`

The Second Order Integrator Block

Compose two first order integration blocks with limits.

```
let hybrid limit_int2
    (xlower, xupper, xlower', xupper', xres, xres', x0, x0', u) =
    (x, x', xstatus, xstatus')
  where
   rec
     (x', xstatus') =
       limit_int(x0', xlower', xupper', xres', fu)
   and
     (x, xstatus) =
       limit_int(x0, xlower, xupper, xres, x')
   and
     fu =
       match xstatus with | Between -> u | Above | Below -> 0.0
```

Discontinuous Blocks



The Backlash

Three modes (Simulink's specification)

- Disengaged: "In this mode, the input does not drive the output and the output remains constant."
- 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."
- 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"

Difficulty

- Detect the change in sign of the derivative.
- But `Zélus` does not provide the derivative of a signal.

The Backlash

Approximate the derivative, either by sampling or a linear filter.

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

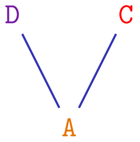
Other blocks

- Saturation blocks, coulomb friction, dead zone, switch, relay, rate limiter, etc.
- Their programming is similar to that for previous examples.
- All programming features of `Zélus` are used: automata, transitions on zero-crossing, left-limit.
- Yet, several blocks cannot be programmed in continuous time: memory block, derivative, time delay.

Separation between Discrete and Continuous Time

The type language [LCTES'11]

$$
\begin{array}{lcl}
bt & ::= & \mathtt{float} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{zero} \mid \cdots \\
\sigma & ::= & bt \times \ldots \times bt \xrightarrow{k} bt \times \ldots \times bt \\
k & ::= & \mathtt{D} \mid \mathtt{C} \mid \mathtt{A}
\end{array}
$$



Function Definition: `fun f(x1,...) = (y1,...)`

- **Combinatorial functions** (`A`); usable anywhere.

Node Definition: `node f(x1,...) = (y1,...)`

- **Discrete-time constructs** (`D`) of SCADE/Lustre: `pre`, `->`, `fby`.

Hybrid Definition: `hybrid f(x1,...) = (y1,...)`

- **Continuous-time constructs** (`C`): `der x = ...`, `up`, `down`, etc.

A program that is rejected

```
let hybrid wrong(x, y) = x >= y

File "wrong.zls", line 1, characters 25-31:
>let hybrid wrong(x, y) = x >= y
>                         ^^^^^^
Type error: this is a stateless discrete expression
and is expected to be continuous.

let hybrid positive(epsilon, x) =
   present
      | up(epsilon -. abs(x)) -> x >= 0.0
   init
     (x >= 0.0)

val above : float -C-> bool
```

Zélus prevents from writting a boolean signal that may change during integration, even if it is not used.

Current status

This is very preliminary work.

- The language was not expressive enough; a very helpful experiment.
- The experiment is done both in `Zélus` and `SCADE Hybrid`
- Is the type system expressive enough when separating discrete an continuous?
- Polymorphism (ad-hoc and parametric) is too limited
- What is the quality of the generated code?

We shall provide an open source version for all blocks.

http://zelus.di.ens.fr

Zélus Download Examples Publications Contact

Zélus

A synchronous language with ODEs

Compiler

Zélus is a synchronous language extended with Ordinary Differential Equations (ODEs) to model systems with complex interaction between discrete-time and continuous-time dynamics. It shares the basic principles of Lustre with features from Lucid Synchrone (type inference, hierarchical automata, and signals). The compiler is written

Research

Zélus is used to experiment with new techniques for building hybrid modelers like Simulink/Stateflow and Modelica on top of a synchronous language. The language exploits novel techniques for defining the semantics of hybrid modelers, it provides dedicated type systems to ensure the absence of discontinuities during integration and the

Bibliography

Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet.
A Type-based Analysis of Causality Loops in Hybrid Systems Modelers.
In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, Berlin, Germany, April 15–17 2014. ACM.

Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet.
A Hybrid Synchronous Language with Hierarchical Automata: Static Typing and Translation to Synchronous Code.
In *ACM SIGPLAN/SIGBED Conference on Embedded Software (EMSOFT'11)*, Taipei, Taiwan, October 2011.

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

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

Albert Benveniste, Benoit Caillaud, and Marc Pouzet.
The Fundamentals of Hybrid Systems Modelers.
In *49th IEEE International Conference on Decision and Control (CDC)*, Atlanta, Georgia, USA, December 15-17 2010.

Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet.
A Synchronous-based Code Generator For Explicit Hybrid Systems Languages.
In *International Conference on Compiler Construction (CC)*, LNCS, London, UK, April 11-18 2015.

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