


Programming the Simulink Standard Library with Zélus: experience report ¹

Marc Pouzet
Marc.Pouzet@ens.fr

DI, ENS

ANR Cafein
Les Angles
February 2, 2016

¹Joint work with Timothy Bourke (INRIA Paris) and Cédric Pasteur (Esterel-Tech.) 

Hybrid Systems Modelers

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

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

Our approach

- ▶ Build a hybrid modeler on top of a **synchronous language**
- ▶ Recycle existing techniques and tools
- ▶ Clarify underlying principles and guide language design/semantics

Recycle existing tools and techniques

Synchronous languages (SCADE/Lustre)

- ▶ Used for critical systems design and implementation
 - ▶ mathematically sound semantics
 - ▶ certified compiler (DO178C)
- ▶ Expressive language for both discrete **controllers** and **mode changes**
- ▶ **But do not support modelling of continuous-time dynamics**

Off-the-shelf ODEs numeric solvers

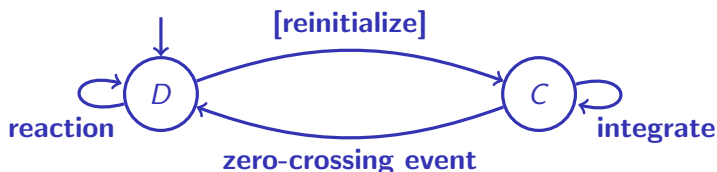
- ▶ Sundials CVODE (LLNL) among others, treated as black boxes
- ▶ Exploit existing techniques and (variable step) solvers

A conservative extension

- ▶ Any synchronous program must be compiled, optimized, and executed as per usual
- ▶ Increase confidence in the simulation code.

The Simulation Engine of Hybrid Systems

Alternate discrete steps and integration steps



$$\sigma', y' = d_{\sigma}(t, y) \quad upz = g_{\sigma}(t, y) \quad \dot{y} = f_{\sigma}(t, y)$$

Properties of the three functions

- ▶ d_{σ} gathers all discrete changes.
- ▶ g_{σ} defines signals for zero-crossing detection.
- ▶ f_{σ} and g_{σ} should be **free of side effects** and, better, **continuous**.

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

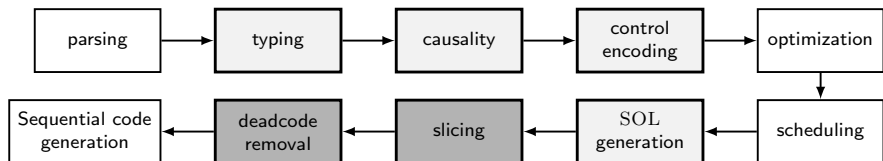
Milestones

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

A validation into the industrial KCG compiler of SCADE

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

- ▶ Prototype based on KCG 6.4 (now 6.6)
- ▶ SCADE Hybrid = full SCADE + ODEs
- ▶ Generates FMI 1.0 model-exchange FMUs with Simplorer



- ▶ Built from scratch.
 - ▶ Follows the organisation of the Lucid Synchrone compiler
 - ▶ Recycle and extend several type-based static analysis: typing, causality analysis, initialization analysis.
 - ▶ Comp. through reduction into a basic language [LCTES'08, CC'15].
- ▶ First prototype around 2011; current version is 1.2.
- ▶ Reference manual and tutorial (2015)
- ▶ 15kLOC for the compiler (Ocaml 4.02.1).
- ▶ Numeric solver: SundialsML (Ocaml binding of SUNDIALS); ODE23, ODE45.

Distribution

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

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

Zélus source code is available on a private svn server.

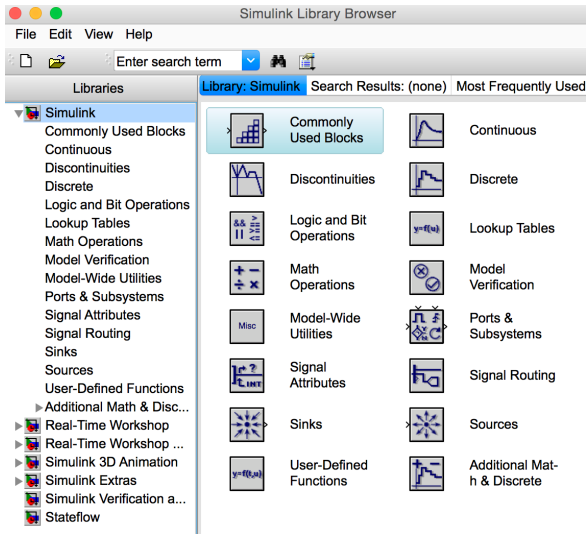
`svn: https://svn.di.ens.fr/svn/zelus/trunk`

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

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

How Zélus behave for programming classical blocks of the Simulink standard library?

The Simulink Standard Library



Discrete Blocks

File Edit View Help

Enter search term




















Libraries

- Simulink
 - Commonly Used Blo...
 - Continuous
 - Discontinuities
 - Discrete**
 - Logic and Bit Operat...
 - Lookup Tables
 - Math Operations
 - Model Verification
 - Model-Wide Utilities
 - Ports & Subsystems
 - Signal Attributes
 - Signal Routing
 - Sinks
 - Sources
 - User-Defined Functi...
 - Additional Math & Di...
- Real-Time Workshop
- Real-Time Worksho...

Library: Simulink/Discrete

Search Results: (none)

Most Frequently Used Blocks

	Difference		Discrete Derivative		Discrete FIR Filter
	Discrete Filter		Discrete PID Controller		Discrete PID Controller (2...
	Discrete State-Space		Discrete Transfer Fcn		Discrete Zero-Pole
	Discrete-Time Integrator		First-Order Hold		Integer Delay
	Memory		Tapped Delay		Transfer Fcn First Order
	Transfer Fcn Lead or Lag		Transfer Fcn Real Zero		Unit Delay
	Zero-Order Hold				

Discrete-time blocks: the Integrator

Examples below are checked by the tool of Timothy Bourke:
`checklisting.sh` ²

```
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)
```

```
type saturation = | Between | Lower | Upper  
val forward_euler : float * float * float * float -D-> float
```

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

Discrete-time blocks: the Integrator (II)

```
(* forall n in Nat.
 * [output(n) = y(n)]
 * [x(0) = x0(0)]
 * [x(n) = y(n-1) + h/2 * u(n-1)]
 * [y(n) = x(n) + h/2 * u(n)] *)
let node trapezoidal_fixed(x0, k, h, u) = output where
  rec x = x0 fby (y +. gain *. u)
  and y = x +. gain *. u
  and gain = k *. h /. 2.0
  and output = y

(* Extended versions with upper/lower limits,
 * reset and output ports *)
let node forward_euler_complete(upper, lower, res, x0, k, h, u) =
  (output, state_port, saturation) where
  rec state_port = x0 fby (output +. k *. h *. u)
  and v = if res then x0 else state_port
  and (output, saturation) =
    if v < lower then lower, Lower
    else if v > upper then upper, Upper else v, Between
```

Discrete-time PID

```
let node int(x0, h, u) = forward_euler(x0, 1.0, h, u)
```

```
let node derivative(h, u) = 0.0 -> (u -. pre(u)) /. h
```

```
(* PID controller in discrete time
```

```
 * p is the proportional gain;
```

```
 * i the integral gain;
```

```
 * d the derivative gain;
```

```
 * h is the time step *)
```

```
let node pid(p, i, d, h, u) = c where
```

```
  rec c_p = p *. u
```

```
  and i_p = int(0.0, h, i *. u)
```

```
  and c_d = derivative(h, d *. u)
```

```
  and c = c_p +. i_p +. c_d
```

Discrete-time PID (II)

```
(* The same but where initial conditions
 * are given from the arguments *)
let node pid_with externals(p, i0, i, d, h, u) = c where
  rec c_p = p *. u
  and i_p = int(i0, h, i *. u)
  and c_d = derivative(h, d *. u)
  and c = c_p +. i_p +. c_d
```

```
(* PID with external reset *)
let node pid_with_reset(r, p, i0, i, d, h, u) = c where
  reset
  c = pid_with externals(p, i0, i, d, h, u)
  every r
```

All other forms are programmed the same way. It is possible to add upper and lower limit.

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 feedbacks are possible.

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

- ▶ Higher order or functors would make the code more generic!
- ▶ A comparison of the generated code must be done.

Well, nothing surprising here. Several tools automatically translate a subset of Simulink discrete-time blocks into Lustre.

But is there some minimal language to define all the blocks precisely?

Continuous Blocks

Enter search term











Libraries

- Simulink
 - Commonly Used Blo...
 - Continuous**
 - Discontinuities
 - Discrete
 - Logic and Bit Operat...
 - Lookup Tables
 - Math Operations
 - Model Verification
 - Model-Wide Utilities
 - Ports & Subsystems
 - Signal Attributes
 - Signal Routing
 - Sinks
 - Sources
 - User-Defined Functi...
 - ▶ Additional Math & Di...
- Real-Time Workshop

Library: Simulink/Continuous

Search Results: (none)

Most Frequently Used Blocks

 $\frac{du}{dt}$	Derivative	 $\frac{1}{s}$	Integrator	 PID(s)	PID Controller
 $\text{Ref PID}(s)$	PID Controller (2DOF)	 $\begin{matrix} \dot{x}' = Ax + Bu \\ y = Cx + Du \end{matrix}$	State-Space	 $\frac{1}{s+1}$	Transfer Fcn
 Δ	Transport Delay	 Δ To	Variable Time Delay	 Δ Ti	Variable Transport Delay
 $\frac{(s-1)}{s(s+1)}$	Zero-Pole				

Navigation icons: back, forward, search, etc.

Continuous-time Integrator

```
(* regular integration of a signal [u]
 * with initial condition [x0]*)
let hybrid int(x0, u) = x where
  rec der x = u init x0

val int : float * float -C-> float

(* integrator with zero-crossing and initial condition *)
let hybrid int_with_reset_state_port(x0, res, u) = (x, state_port)
  rec der x = u init x0 reset res -> x0
  and state_port = last x

val int_with_reset_state_port : float * zero * float -C-> float * f
```

Last

- ▶ `last x` is the previous value of `x`
- ▶ When `x` is a continuous state variable, it is its left limit.
- ▶ It corresponds to the so-called *state port* of Simulink.

Continuous-time Integrator (II)

```
(* integrator with initial condition [x0] with threshold [lower]
 * and [upper] *)
let hybrid int_limit(x0, upper, lower, u) =
  (x, state_port, saturation) where
    rec init x = x0
    and automaton
      | Between ->
        (* regular mode. Integrate the signal *)
        do der x = u and saturation = Between
        until (xup(x, upper)) then do x = upper in Upper
        else (xdown(x, lower)) then do x = lower in Lower
      | Upper ->
        (* when the speed becomes negative, go to *)
        (* the regular mode *)
        do saturation = Upper until (down(u)) then Between
      | Lower ->
        (* when positive, go to the regular mode *)
        do saturation = Lower until (up(u)) then Between
    end
  and state_port = last x
```

Continuous-time Integrator (III)

The same with parameterized states.

```
let hybrid int_limit2(x0, upper, lower, u) =
  (x, state_port, saturation) where
  rec init x = x0
  and automaton
    | Between ->
      (* regular mode. Integrate the signal *)
      do der x = u and saturation = Between
      until (xup(x, upper)) then do x = upper in Bound(false)
      else (xdown(x, lower)) then do x = lower in Bound(true)
    | Bound(right) ->
      (* when the speed becomes negative, go to the *)
      (* regular mode *)
      do saturation = if right then Lower else Upper
      until (up(if right then u else -. u)) then Between
  end
  and state_port = last x
```

Continuous-time Integrator (IV)

```
(* integrator with zero-crossing, initial condition,  
 * thresholds, and reset *)  
let hybrid int_complete(x0, upper, lower, res, u) =  
  (x, state_port, saturation) where  
  rec reset  
    (x, state_port, saturation) =  
      int_limit(x0, upper, lower, u)  
  every res
```

Continuous-time PID

```
(* 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 derivative(n, f0, u) = udot where
  rec udot = n *. (u -. f)
  and der f = udot init f0

let hybrid int(x0, u) = x where
  rec der x = u init x0

(* 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(p, i, d, n, u) = c where
  rec c_p = p *. u
  and i_p = int(0.0, i *. u)
  and c_d = derivative(n, 0.0, d *. u)
  and c = c_p +. i_p +. c_d
```

Continuous-time PID (II)

```
(* The same but where initial conditions are given from
 * the arguments *)
let hybrid pid_with externals(p, i0, f0, i, d, n, u) = c where
  rec c_p = p *. u
  and i_p = int(0.0, i *. u)
  and c_d = derivative(n, f0, d *. u)
  and c = c_p +. i_p +. c_d

(* PID with external reset *)
let hybrid pid_with_reset(r, p, i0, f0, i, d, n, u) = c where
  reset
  c = pid_with externals(p, i0, f0, i, d, n, u)
  every r
```

Continuous-time blocks

The programming style is similar than that for discrete-time blocks.













- ▶ The dependence-based causality analysis extends directly to ODEs:
 - ▶ An equation `der x = ...x... init x0` is like `x = x0 fby (...x...)` in data-flow: `x` does not depend on itself.
 - ▶ E.g., `der x = 1.0 -. x init 0.0` is causally correct.
 - ▶ E.g., `der x = 1.0 init 0.0 reset z → 0.2 *. x` is not.
 - ▶ `last x` must be used to break causality loops.
- ▶ Same weaknesses as before: no overloading, arrays, etc.
- ▶ Again, high-order would make the code more generic.
- ▶ Yet, current typing constraints that forbid the use of comparison operators during integration is a burden.

Discontinuous Blocks

Libraries

- Simulink
- Commonly Used Blo...
- Continuous
- Discontinuities**
- Discrete
- Logic and Bit Operat...
- Lookup Tables
- Math Operations
- Model Verification
- Model-Wide Utilities
- Ports & Subsystems
- Signal Attributes
- Signal Routing
- Sinks

Library: Simulink/Discontinuities Search Results: (none) Most Frequently Used Blocks

	Backlash		Coulomb & Viscous Friction		Dead Zone
	Dead Zone Dynamic		Hit Crossing		Quantizer
	Rate Limiter		Rate Limiter Dynamic		Relay
	Saturation		Saturation Dynamic		Wrap To Zero

Rising, falling, either

```
(* Hit crossing *)  
let hybrid rising(input, offset) = up(input -. offset)  
  
let hybrid falling(input, offset) = up(-. input +. offset)  
  
let hybrid positive(x) = 0 where  
rec 0 = present (up(x)) -> true  
      | (up(-. x)) -> false  
      init (x >= 0.0)  
  
val rising : float * float -C-> zero  
val falling : float * float -C-> zero  
val positive : float -C-> bool
```

Rising, falling, either

```
let hybrid either(eps, input, offset) = ok where
  rec automaton
    | AtCrossing ->
      do ok = true
      unless (up(input -. (offset +. eps))) then Higher
      else (up(-. (input -. (offset -. eps)))) then Lower
    | Higher ->
      do ok = false
      unless (up(-. (input +. offset))) then AtCrossing
    | Lower ->
      do ok = false
      unless (up(input -. offset)) then AtCrossing
  end

val either : float * float * float -C-> bool
```

Either (II)

$ok = true$ when $abs(x - v) \leq \epsilon$

```
let hybrid either_abs(eps, x, v) = ok where  
  rec o = abs_float(x -. v) -. eps and ok = positive(o)
```

```
val either_abs : float * float * float -> bool
```

Difficulty

- ▶ up(e) only detect that e crosses zero, i.e., from strictly negative to strictly positive.
- ▶ It does not detect that a signal stucked to zero then leaves zero.
- ▶ Adding ϵ (as we do), with or without an hysteresis is not satisfactory.
- ▶ The problem is not simple at all!

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.

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

Other blocks

Discontinuous 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.

Modeling a second order system with saturation

See: <http://blogs.mathworks.com/seth/2014/01/22/how-to-model-a-hard-stop-in-simulink/>

Example: clutch control

DEMO

The Zélus typing discipline

- ▶ All discontinuities are aligned with a zero-crossing.
- ▶ Checked during static typing.
- ▶ In particular, values with a discrete type (e.g., bool, int) do not change during integration.

Discrete events

A zero-crossing `up(e)` returns a value of type zero.

It can be used to trigger a regular Lustre node.

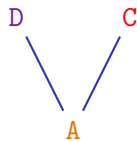
Separation between Discrete and Continuous Time

The type language [LCTES'11]

$bt ::= \text{float} \mid \text{int} \mid \text{bool} \mid \text{zero} \mid \dots$

$\sigma ::= bt \times \dots \times bt \xrightarrow{k} bt \times \dots \times bt$

$k ::= D \mid C \mid A$



Function Definition: $\text{fun } f(x_1, \dots) = (y_1, \dots)$

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

Node Definition: $\text{node } f(x_1, \dots) = (y_1, \dots)$

- ▶ **Discrete-time constructs** (D) of SCADE/Lustre: pre, \rightarrow , fby.

Hybrid Definition: $\text{hybrid } f(x_1, \dots) = (y_1, \dots)$

- ▶ **Continuous-time constructs** (C): der $x = \dots$, up, down, etc.

An excerpt of the Pervasives module from Zélus

AD for discrete stateless functions ($A < AD < D$).

```
val ( = ) : 'a * 'a -AD-> bool
val ( <> ) : 'a * 'a -AD-> bool
val ( < ) : 'a * 'a -AD-> bool
val ( > ) : 'a * 'a -AD-> bool
val ( <= ) : 'a * 'a -AD-> bool
val ( >= ) : 'a * 'a -AD-> bool

val min : 'a * 'a -A-> 'a

val not : bool -A-> bool
val ( & ) : bool * bool -A-> bool
val ( or ) : bool * bool -A-> bool

val up : float -C-> zero
val disc : 'a -C-> Zero
val ( on ) : zero * bool -> zero
val orz : zero * zero -> zero
...
```

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 above(x) = present
                        | up(x) -> true
                        | up(-. x) -> false
                        init
                          (x >= 0)

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

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

Zélus vs SCADE Hybrid

SCADE Hybrid is less constraining: Only state changes must be aligned on a zero-crossing. But, the argument of an `up(.)` must be “smooth” (C^∞).

The separation between continuous-time and discrete-time is done during the **clock calculus**.

It is possible to write comparisons and signals that are discontinuous during integration, whereas Zélus forbids, e.g.:

```
if x >= high then high else if x <= low then low else x
```

```
...
```

```
der x = if x >= 0.0 then -1.0 else 1.0 init 0.0
```

An extra type system indicates whether a signal is piece-wise constant, smooth or (possibly) discontinuous between two zero-crossings.

Current work

Write other blocks from the standard Simulink library; write complete examples to exercise the compiler.

- ▶ Examples: Backhoe, Bang bang controller, clutch model, etc.
- ▶ These experiments exercise all features of the languages.
- ▶ Zélus imposes a change in programming style w.r.t Simulink.
- ▶ Zélus reject programs that are accepted by Simulink.
- ▶ E.g., current typing constraints of Zélus forbids the use of comparisons during integration (`if x >= 0.0 then ...` is forbidden). Use boolean operations instead (`if x then ...` is valid).

This is preliminary work: compare efficiency, accuracy, style, **in detail**.

Some problems we faced (I)

Causality analysis

An output should not depend on a condition when it weak. Zélus is too constraining, making the output depending on the condition.

In comparison, SCADE Hybrid does better.

Typing discrete/continuous blocks

- ▶ We may have been overly constraining as solvers manage discontinuous signals. Yet, results may be weird.
- ▶ SCADE Hybrid is more permissive. It is possible to write `der x = if x >= 0 then -1.0 else 1.0` but impose that `e` be smooth in `up(e)`.
- ▶ More experiments to be done. An idea is to associate the kind to type variables. E.g.,:

```
val (=) : Discrete('a). 'a * 'a -A-> bool
```

or use an extra type system as in SCADE Hybrid.

II

What is the minimal mathematical and executable language we need to define all the blocks, with a description not longer than the Simulink documentation?

Extensions

- ▶ Translate ODEs into difference equations (synchronous code) for real-time simulation.
- ▶ The derivative of a signal is not provided. Should we provide automatic differentiation or an internal implementation that give an approximated value (e.g., as Simulink does)?
- ▶ Arrays. SCADE Hybrid have them; Implementation in Zélus is under way with a form of iterator inspired by SISAL.

Verification

- ▶ Translate programs into a basic Lustre-like language with ODEs, used as an input for formal verification.

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 Synchronic](#) (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.