Mixing Discrete and Continuous-time with the Synchronous Language Zélus

Marc Pouzet

DI, École normale supérieure
Marc.Pouzet@ens.fr

Seminar OVSTR
Paris, April 26, 2017
Model Based Design

Write executable mathematical specifications in a high-level language so that a model is:

- a reference semantics independent of any implementation
- a base for simulation, testing, formal verification
- then compiled into executable code, sequential and parallel
- with semantics preservation all along the chain

A way to achieve correct-by-construction software
The early 80’s

The very first block diagrams (flight-by-wire command of Airbus A320)
Lustre: a dataflow programming language


Programming with streams

constants

\[
\begin{align*}
1 &= 1 & 1 & 1 & 1 & 1 & \ldots
\end{align*}
\]

operators

\[
\begin{align*}
x + y &= x_0 + y_0 & x_1 + y_1 & x_2 + y_2 & x_3 + y_3 & \ldots
\end{align*}
\]

\((z = x + y \text{ means that at every instant } i : z_i = x_i + y_i)\)

unit delay

\[
\begin{align*}
0 \text{ fby } (x + y) &= 0 & x_0 + y_0 & x_1 + x_1 & x_2 + x_2 & \ldots
\end{align*}
\]

\[
\begin{align*}
\text{pre } (x + y) &= \text{nil} & x_0 + y_0 & x_1 + x_1 & x_2 + x_2 & \ldots
\end{align*}
\]

\[
\begin{align*}
0 \rightarrow \text{pre } (x + y) &= 0 & x_0 + y_0 & x_1 + x_1 & x_2 + x_2 & \ldots
\end{align*}
\]
The beautiful idea of Lustre

“Program in the semantics” (Paul Caspi)

- Time is logical and discrete = first neglect actual execution time then check the implementation is fast enough.
- Parallel composition preserves determinacy: very important in practice for reproducibility
- Parallelism is compiled: programs are translated into seq. code.
- This code executes in bounded memory and bounded time.
- Invariant properties are expressed as Lustre programs.
- Formal verification and testing considered from the very beginning.
- Numerous extensions have been proposed. E.g., programming constructs, mix of deterministic and non-deterministic programs.
SCADE:¹ (Safety Critical Application Development Env.)

- Pioneering work of Caspi and Halbwachs.
- Integrate original language extensions and compilation techniques.

¹http://www.esterel-technologies.com/products/scade-suite/
What about hybrid systems which mix discrete-time and continuous-time signals?
The Current Practice of Hybrid Systems Modeling

Embedded software interacts with physical devices

The whole system has to be modeled: the controller and the plant

Mix of models: discrete time and continuous time (ODEs, DAEs)

²Image from ANSYS/Esterel-Technologies
The Current Practice of Hybrid Systems Modeling

A wide range of languages and tools:

- **PL:** Simulink/Stateflow/SimScape, Modelica, Scicos, Ptolemy,...
- **Interconnect tools:** Simulink + Modelica + SCADE + Simplocre
- **Interchange format** for co-simulation: S-functions, FMU/FMI

But:

- Simulink, Modelica used to **model**, less to **implement** critical soft.
- Software often **reimplemented** in imperative code.

**Can we increase the confidence in what is simulated and executed?**

**Existing tools are not perfect: can we improve them?**
The Simulation Engine of Hybrid Systems

Alternate discrete steps and integration steps

\[ \sigma', y' = \text{next}_\sigma(t, y) \quad \text{upz} = g_\sigma(t, y) \quad \dot{y} = f_\sigma(t, y) \]

Properties of the three functions

- \( \text{next}_\sigma \) gathers all discrete changes.
- \( g_\sigma \) defines signals for zero-crossing detection.
- \( f_\sigma \) is the function to integrate.
Compilation

The Compiler has to produce:

1. Initialization function \textit{init} to define $y(0)$ and $\sigma(0)$.
2. Functions $f$ and $g$.
3. Function \textit{next}.

The Runtime System

1. This compilation scheme is independent of the solver.
2. Either program the simulation loop, using a black-box solver (e.g., SUNDIALS CVODE);
3. Or rely on an existing infrastructure (FMU/FMI, etc.).

\textit{f} and \textit{g} should be side effect free and, better, continuous.
Some models are fragile...
Typing issue 1: Mixing continuous & discrete components

- **Constant**
- **Integrator**
- **Add**
- **Unit Delay**
- **Scope**

The shape of `cpt` depends on the steps chosen by the solver. Putting another component in parallel can change the result.

**Basic model**

![Graph showing the output of the model over time](image)
Typing issue 1: Mixing continuous & discrete components

- The shape of \( \text{cpt} \) depends on the steps chosen by the solver.
- Putting another component in parallel can change the result.
How long is a discrete step?

- Adding a parallel component changes the result.
- No warning by the compiler.
- The manual says: “A single transition is taken per major step”.

Discrete time is not logical: it is that of the simulation engine.
Causality issue: the Simulink state port

The output of the state port is the same as the output of the block’s standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block’s standard output if the block had not been reset.

–Simulink Reference (2-685)
The output of the state port is the same as the output of the block’s standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block’s standard output if the block had not been reset.

–Simulink Reference (2-685)
Causality issue: the Simulink state port

\[ t < 2: \quad x(t) = t, \quad y(t) = \frac{t^2}{2} \]

\[ t = 2: \quad x = -3 \cdot \text{last } y = -6, \quad y = -4 \cdot \text{last } x = -8 \]

The output of the state port is the same as the output of the block’s standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block’s standard output if the block had not been reset.

–Simulink Reference (2-685)
Causality issue: the Simulink state port

\[ t < 2: \quad x(t) = t, \quad y(t) = \frac{t^2}{2} \]

\[ t = 2: \quad x = -3 \cdot \text{last } y = -6, \quad y = -4 \cdot \text{last } x = -8 \]

The output of the state port is the same as the output of the block's standard output except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.

–Simulink Reference (2-685)
static void mdlOutputs(SimStruct * S, int_T tid)
{
    _rtX = (ssGetContStates(S));
    ...
    _rtB = (_ssGetBlockIO(S));
    _rtB->B_0_0_0 = _rtX->Integrator1_CSTATE + _rtP->P_0;
    _rtB->B_0_1_0 = _rtP->P_1 * _rtX->Integrator1_CSTATE;
    if (ssIsMajorTimeStep (S))
    {
        if (zcEvent || ...)
        {
            (ssGetContStates (S))->Integrator0_CSTATE =
                _ssGetBlockIO (S))->B_0_1_0;
        }
    }
    ...
    (_ssGetBlockIO (S))->B_0_2_0 =
        (ssGetContStates (S))->Integrator0_CSTATE;
    _rtB->B_0_3_0 = _rtP->P_2 * _rtX->Integrator0_CSTATE;
    if (ssIsMajorTimeStep (S))
    {
        if (zcEvent || ...)
        {
            (ssGetContStates (S))->Integrator1_CSTATE =
                (ssGetBlockIO (S))->B_0_3_0;
        }
    }
    ... } ... 

Before assignment: integrator state contains ‘last’ value
After assignment: integrator state contains the new value

So, $y$ is updated with the new value of $x$,

Before assignment: $x = -3 \cdot \text{last } y$
After assignment: $x = -3 \cdot \text{new } y$
Causality: Modelica example

model scheduling
  Real x(start = 0);
  Real y(start = 0);
equation
  der(x) = 1;
  der(y) = x;

  when x >= 2 then
    reinit(x, −3 * y)
  end when;
  when x >= 2 then
    reinit(y, −4 * x);
  end when;
end scheduling;
Causality: Modelica example

model scheduling
  Real x(start = 0);
  Real y(start = 0);
equation
  der(x) = 1;
  der(y) = x;
  when x >= 2 then
    reinit(x, -3 * y)
  end when;
  when x >= 2 then
    reinit(y, -4 * x);
  end when;
end scheduling;
model scheduling
Real x(start = 0);
Real y(start = 0);
equation
  der(x) = 1;
  der(y) = x;
when x >= 2 then
  reinit(x, −3 * y)
end when;
when x >= 2 then
  reinit(y, −4 * x);
end when;
end scheduling;

OpenModelica 1.9.2beta1 (r24372)
Also in Dymola
Integrating a discontinuous signal

Solver Reset
Resetting the solver at every zero-crossing event degrades performance and precision.

Yet, integrating a discontinuous signal gives non faithfull results, e.g.:

\[
\text{der } x = \begin{cases} 
1.0 & \text{if } \text{floor}(t \mod 2.0) = 0.0 \\
0.0 & \text{else }
\end{cases} \text{ init } 0.0
\]

E.g., SUNDIALS CVODE, the ones provided by Simulink.

Can we impose a strong type discipline to either reject this program or indicate that the result of \( x \) is fragile?
Hybrid Systems from a Programming Language Perspective

Edward Lee and Haiyang Zheng (HSCC, 2005):

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

- Which compositions of **discrete and continuous time** make sense?
- Which programs should be **statically rejected**?
- How to ensure **determinacy**?
- How to **compile** programs faithfully and efficiently?

With A. Benveniste, B. Caillaud (Inria Rennes) since 2010.
Build a Hybrid Modeler on Synch. Language Principles

Milestones

▶ An ideal semantics based on non standard analysis [JCSS’12]
▶ Lustre with ODEs [LCTES’11]
▶ Hierarchical automata, both discrete and hybrid [EMSOFT’11]
▶ Causality analysis [HSCC’14]
▶ Sequential code generation [CC’15]

Implemented in Zélus [HCSS’13]


Simulate with an off-the-shelf, variable-step numerical solver: SUNDIALS CVODE from LLNL (with OCaml binding)
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
Synchronous languages in a slide

▶ Compose stream functions; basic values are streams.
▶ Operation apply pointwise + unit delay \((\text{fby})\) + automata.

\[(* \text{computes } [x(n) + y(n) + 1] \text{ at every instant } [n] *)\]

\[
\text{fun add } (x,y) = x + y + 1
\]

\[(* \text{returns } [\text{true}] \text{ when the number of } [t] \text{ has reached } [\text{bound}] *)\]

\[
\text{node after } n \ (\text{bound}, t) = (c = \text{bound}) \text{ where}
\]
\[
\text{rec } c = 0 \ \text{fby } (\text{min}(\text{tick}, \text{bound}))
\]
\[
\text{and } \text{tick} = \text{if } t \text{ then } c + 1 \text{ else } c
\]

The counter can be instantiated twice in a two state automaton,

\[
\text{node blink } (n, m, t) = x \text{ where}
\]
\[
\text{automaton}
\]
\[
| \text{On } \rightarrow \text{do } x = \text{true} \text{ until } (\text{after}(n, t)) \text{ then } \text{Off}
\]
\[
| \text{Off } \rightarrow \text{do } x = \text{false} \text{ until } (\text{after}(m, t)) \text{ then } \text{On}
\]

From it, a synchronous compiler produces \textbf{sequential loop-free code} that compute a single \textbf{step} of the system.
A Simple Hybrid System

Yet, time was discrete. Now, a simple heat controller. ³

(* a model of the heater defined by an ODE with two modes *)

hybrid heater(active) = temp where
  rec der temp = if active then c –. k *. temp else –. k *. temp
  init temp0

(* an hysteresis controller for a heater *)

hybrid hysteresis_controller(temp) = active where
  rec automaton
    | Idle → do active = false until (up(t_min –. temp)) then Active
    | Active → do active = true until (up(temp –. t_max)) then Idle

(* The controller and the plant are put parallel *)

hybrid main() = temp where
  rec active = hysteresis_controller(temp)
  and temp = heater(active)

Three syntactic novelties: keyword hybrid, der and up.

³Hybrid version of N. Halbwachs’s example in Lustre at Collège de France, Jan.10.
From Discrete to Hybrid

The type language [LCTES'11]

\[ bt ::= \text{float} \mid \text{int} \mid \text{bool} \mid \text{zero} \mid \cdots \]

\[ \sigma ::= bt \times \ldots \times bt \xrightarrow{k} bt \times \ldots \times bt \]

\[ k ::= D \mid C \mid A \]

Function Definition: \texttt{fun f(x1,\ldots) = (y1,\ldots)}

- Combinatorial functions (A); usable anywhere.

Node Definition: \texttt{node f(x1,\ldots) = (y1,\ldots)}

- Discrete-time constructs (D) of SCADE/Lustre: \texttt{pre, ->, fby}.

Hybrid Definition: \texttt{hybrid f(x1,\ldots) = (y1,\ldots)}

- Continuous-time constructs (C): \texttt{der x = \ldots, up, down, etc.}
Mixing continuous/discrete parts

Zero-crossing events

- They correspond to event indicators/state events in FMI
- Detected by the solver when a given signal crosses zero

Design choices

- A discrete computation can only be triggered by a zero-crossing
- Discrete state only changes at a zero-crossing event
- A continuous state can be reset at a zero-crossing event
Example

node counter() = cpt where
  rec cpt = 1 → pre cpt + 1

hybrid hybrid_counter() = cpt where
  rec cpt = present up(z) → counter() init 0
  and z = sinus()

Output with SCADE Hybrid + Simplover
Mix of synchronous code and a continuous model

E.g., implement the hysteresis controller by a synchronous function.

(* a discrete−time model of the controller; periodically sampled *)
node hysteresis_controller(temp) = active where
  rec automaton
    | Idle → do active = false until (temp <= t_min)) then Active
    | Active → do active = true until (temp >= t_max) then Idle

(* The controller and the plant are put parallel *)
hybrid main() = temp where
  rec active = present (period (0.1)) → hysteresis_controller(temp) init false
  and temp = heater(active)
Continuous and discrete PI controller [demo]

```
hybrid integr(x, y0) = y where
  rec der y = x init y0

hybrid pi(kp, ki, i) = cmd where
  rec cmd = kp * i + ki * integr(i, 0.0)
tel

let ts = 0.05

node disc_integr(x, y0) = y where
  rec init y = y0 and y = last y + Ts * x

node disc_pi(kp, ki, i) = cmd where
  rec init cmd = Kp*i
  and cmd = kp * i + ki * disc_integr(i, 0.0)
```
How to communicate between continuous and discrete time?

E.g., the bouncing ball

```plaintext
hybrid ball(y0) = y where
  rec der y = y \_v init y0
  and der y \_v = - \cdot g init 0.0 reset z \to 0.8 \cdot last y \_v
  and z = up(- \cdot y)
```

- Replacing `last y \_v` by `y \_v` would lead to a deadlock.
- In SCADE and Zélus, `last y \_v` is the previous value of `y \_v`.
- It coincides with the **left limit** of `y \_v` when `y \_v` is left continuous.
Internals
Compiler Architecture

Two implementations: Zéłus and KCG 6.4 (Release 2014) of SCADE.

KCG 6.4 of SCADE

- Generates FMI 1.0 model-exchange FMUs for Simploter.
- Only 5% of the compiler modified. Small changes in:
  - static analysis (typing, causality).
  - automata translation; code generation.
  - FMU generation (XML description, wrapper).
- FMU integration loop: about 1000 LoC.

```
<table>
<thead>
<tr>
<th>parsing</th>
<th>typing</th>
<th>causality</th>
<th>control encoding</th>
<th>optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>C code generation</td>
<td>deadcode removal</td>
<td>slicing</td>
<td>SOL generation</td>
<td>scheduling</td>
</tr>
</tbody>
</table>
```
A SCADE-like Input Language

Essentially SCADE with three syntax extensions (in red).

\[
\begin{align*}
\text{d} & ::= \text{const } x = e \mid k \ f(pi) = pi \text{ where } E \mid d; d \\
\text{k} & ::= \text{fun} \mid \text{node} \mid \text{hybrid} \\
\text{e} & ::= x \mid v \mid \text{op}(e, \ldots, e) \mid v \ \text{fby} \ e \mid \text{last } x \mid f(e, \ldots, e) \mid \text{up}(e) \\
\text{p} & ::= x \mid (x, \ldots, x) \\
\text{pi} & ::= xi \mid xi, \ldots, xi \\
\text{xi} & ::= x \mid x \text{ last } e \mid x \text{ default } e \\
\text{E} & ::= p = e \mid \text{der } x = e \\
& \quad \mid \text{if } e \text{ then } E \text{ else } E \\
& \quad \mid \text{reset } E \text{ every } e \\
& \quad \mid \text{local } pi \text{ in } E \mid \text{do } E \text{ and } \ldots \text{ E done}
\end{align*}
\]
A Clocked Data-flow Internal Language

The internal language is extended with three extra operations. Translation based on Colaco et al. [EMSOFT’05].

\[ d ::= \ \text{const} \ x = c \mid k \ f(p) = a \ \text{where} \ C \mid d; d \]

\[ k ::= \ \text{fun} \mid \text{node} \mid \text{hybrid} \]

\[ C ::= (x_i = a_i)_{x_i \in I} \ \text{with} \ \forall i \neq j. x_i \neq x_j \]

\[ a ::= e^{ck} \]

\[ e ::= x \mid v \mid \text{op}(a, ..., a) \mid v \ fby \ a \mid \text{pre}(a) \]
\[ \quad \mid f(a, ..., a) \]
\[ \quad \mid \text{merge}(a, a, a) \mid a \ \text{when} \ a \]
\[ \quad \mid \text{integr}(a, a) \mid \text{up}(a) \]

\[ p ::= x \mid (x, ..., x) \]

\[ ck ::= \ \text{base} \mid ck \ \text{on} \ a \]
Clocked Equations Put in Normal Form

Name the result of every stateful operation. Separate into syntactic categories.

- **se**: strict expressions
- **de**: delayed expressions
- **ce**: controlled expressions.

Equation $lx = \text{integr}(x', x)$ defines $lx$ to be the continuous state variable; possibly reset with $x$.

\[
\begin{align*}
\text{eq} & ::= x = \text{ce}^{ck} \mid x = f(sa, \ldots, sa)^{ck} \mid x = \text{de}^{ck} \\
\text{sa} & ::= \text{se}^{ck} \\
\text{ca} & ::= \text{ce}^{ck} \\
\text{se} & ::= x \mid v \mid \text{op}(sa, \ldots, sa) \mid sa \text{ when } sa \\
\text{ce} & ::= \text{se} \mid \text{merge}(sa, ca, ca) \mid ca \text{ when } sa \\
\text{de} & ::= \text{pre}(ca) \mid v \text{ fby } ca \mid \text{integr}(ca, ca) \mid \text{up}(ca)
\end{align*}
\]
Well Scheduled Form

Equations are statically scheduled.

\(\text{Read}(a):\) set of variables read by \(a\).

Given \(C = (x_i = a_i)_{x_i \in I}\), a valid schedule is a one-to-one function

\[\text{Schedule}(.) : I \rightarrow \{1 \ldots |I|\}\]

such that, for all \(x_i \in I, x_j \in \text{Read}(a_i) \cap I\):

1. if \(a_i\) is strict, \(\text{Schedule}(x_j) < \text{Schedule}(x_i)\) and
2. if \(a_i\) is delayed, \(\text{Schedule}(x_i) \leq \text{Schedule}(x_j)\).

From the data-dependence point-of-view, \(\text{integ}(ca_1, ca_2)\) and \(\text{up}(ca)\) break instantaneous loops.
A Sequential Object Language (SOL)

- Translation into an intermediate imperative language [Colaco et al., LCTES’08]
- Instead of producing two methods step and reset, produce more.
- Mark memory variables with a kind \( m \)

\[
\begin{align*}
md & ::= \text{const } x = c \\
& \quad | \text{const } f = \text{class} \langle M, I, (method_i(p_i) = e_i \text{ where } S_i)_{i \in [1..n]} \rangle \\
M & ::= [x : m[= v]; \ldots; x : m[= v]] \\
I & ::= [o : f; \ldots; o : f] \\
m & ::= \text{Discrete} | \text{Zero} | \text{Cont} \\
e & ::= v | lv | \text{op}(e, \ldots, e) | o\text{.method}(e, \ldots, e) \\
S & ::= () | lv \leftarrow e | S ; S | \text{var } x, \ldots, x \text{ in } S | \text{if } c \text{ then } S \text{ else } S \\
R, L & ::= S; \ldots; S \\
lv & ::= x | lv\text{.field} | \text{state}(x)
\end{align*}
\]
State Variables

Discrete State Variables (sort *Discrete*)

- Read with `state(x)`;
- modified with `state(x) ← c`

Zero-crossing State Variables (sort *Zero*)

- A pair with two fields.
- The field `state(x).zin` is a boolean, true when a zero-crossing on `x` has been detected, false otherwise.
- The field `state(x).zout` is the value for which a zero-crossing must be detected.

Continuous State Variables (sort *Cont*)

- `state(x).der` is its instantaneous derivative;
- `state(x).pos` its value
Example: translation of the bouncing ball

let bouncing = machine(continuous) {
  memories disc init_25 : bool = true;
  zero result_17 : bool = false;
  cont y_v_15 : float = 0.; cont y_14 : float = 0.

  method reset =
    init_25 <- true; y_v_15.pos <- 0.

  method step time_23 y0_9 =
    (if init_25 then (y_14.pos <- y0_9; ()) else ());
    init_25 <- false;
    result_17.zout <- (~-.) y_14.pos;
    if result_17.zin
      then (y_v_15.pos <- (* .) 0.8 y_v_15.pos);
    y_14.der <- y_v_15.pos;
    y_v_15.der <- (~-.) g; y_14.pos }

Finally

1. Translate as usual to produce a function \texttt{step}.
2. For hybrid nodes, \textbf{copy-and-paste} the step method.
3. Either into a \texttt{cont} method activated during the continuous mode, or two extra methods \texttt{derivatives} and \texttt{crossings}.
4. Apply the following:
   - During the continuous mode (method \texttt{cont}), all zero-crossings (variables of type \texttt{zero}, e.g., \texttt{state(x).zin}) are surely false. All zero-crossing outputs (\texttt{state(x).zout \leftarrow ...}) are useless.
   - During the discrete step (method \texttt{step}), all derivative changes (\texttt{state(x).der \leftarrow ...}) are useless.
   - Remove dead-code by calling an existing pass.
5. That’s all!

Examples (both Zélus and SCADE) at: \texttt{zelus.di.ens.fr/cc2015}
Example: translation of the bouncing ball

let bouncing = machine(continuous) {
  memories disc init_25 : bool = true;
  zero result_17 : bool = false;
  cont y_v_15 : float = 0.; cont y_14 : float = 0.
  method reset =
    init_25 <- true; y_v_15.pos <- 0.
  method step time_23 y0_9 =
    (if init_25 then (y_14.pos <- y0_9; ()) else ());
    init_25 <- false;
    if result_17.zin
      then (y_v_15.pos <- ( \cdot ) 0.8 y_v_15.pos);
    y_14.pos
  method cont time_23 y0_9 =
    result_17.zout <- (~-) y_14.pos;
    y_14.der <- y_v_15.pos;
    y_v_15.der <- (~-) g }

Some conclusion

Two experiments

- The **Zélus** academic language and compiler.
- The industrial **KCG 6.7** (Release 2016) code generator of SCADE.
- For KCG, **less than 5%** of extra LOC, in all.
- The extension is **fully conservative** w.r.t existing SCADE.
- The very same code is used both for simulation and embedded code.

Yet, is this type discipline too constraining for writing real applications?
Current Work

Can we define a standard library of control blocks, in both discrete and continuous-time so that the *the code is the mathematical specification*?

- E.g., delays and tapped-delays, FIR/IIR, transfer functions, integration (with limit, reset), PID, etc.
- Avoid explicit reference to the major step.
- Some continuous-time blocks are wrongly typed: memory block, derivative, rate limiter, backlash, transport delay.
- Replace SUNDIALS by a guaranteed solver working with intervals?
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
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.

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