Zelus: what's `up(.)`?

Marc Pouzet

École normale supérieure

`Marc.Pouzet@ens.fr`

March 20, 2019

Modeliscale meeting, Rennes

A hybrid system = a system with mixed continuous/discrete signals

We focus on languages to program them,

to write executable models.

Domain specific languages

The basic objects that are manipulated are:

infinite streams, stream function, difference/stream equations.

Hierarchical automata.

+ *Ordinary Differential Equations (ODEs) with event detection.*

+ *Differential Algebraic Equations (DAEs)* (not considered here)

An synchronous interpretation of time: time is global and shared by all.

Examples of such languages

For discrete-time models:

Synchronous langages, e.g., Scade.

Precise semantics, high confidence in the correctness of the compiler.

For the more general case of hybrid models:

Simulink/Stateflow, Modelica, Ptolemy, Scicos.

The precise semantics of a program and/or the specification of all the compilation steps are more difficult to define.

The compiler has a central role

Produces executable code

for efficient simulation and/or and embedded platform.

It has many complicated internal steps:

detect/reject statically certain models.

e.g., typing, detection of algebraic loops, clock/rate inference.

e.g., static scheduling, inlining, source-to-source transformations, separation of the continuous/discrete-time part, link with an ODE solver.

Each can introduce errors.

They are different, but not less important, from the errors introduced by the numerical approximations made by the solver itself.

Statically detect/reject certain models

Some model mix logical discrete time and continuous time in an ambiguous or wrong manner;

and/or contain non desired algebraic loops.
E.g., some basic constructs explictly refer to the "major simulation step".

This make models extremely fragile, hard to reuse;
their simulation is hard to reproduce.

Cf. example by Albert B. on tuesday [1]

Can we do better and at what price?

[1] Many others available at: `zelus.di.ens.fr`

The language Zelus

To study those questions, define a minimalistic PL where the static and dynamic semantics are modular and specified precisely.

Which models make sense?

Which should be statically rejected?

How to ensure determinacy?

How to ensure that the compiler preserves the ideal semantics?

Reuse synchronous languages principles

T. Bourke, A. Benveniste, B. Caillaud.
J.-L. Colaco, B. Pagano, C. Pasteur (ANSYS), since 2013.

- An ideal semantics based on *non standard analysis* *[JCSS'12]*
- Lustre with ODEs *[LCTES'11]*
- Typing discrete/continuous *[LCTES'11]*
- Hierarchical automata, discrete and hybride. *[EMSOFT'11]*
- Causality analysis *[HSCC'14]*
- Sequential code generation *[CC'15]*
- Higher-order, standard library (FIR, PID, etc.) *[EMSOFT'17]*

Implemented in Zélus *[HCSS'13]*

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

Simulation with a variable step solver: SUNDIALS Cvode (from LLNL)

Zélus = Lustre + ODEs + zero crossings

Zélus = Lustre + ...

A discrete system: a stream function; streams are synchronous.

x	1	2	1	4	5	6	...
y	2	4	2	1	1	2	...
$x + y$	3	6	3	5	6	8	...
pre x	*nil*	1	2	1	4	5	...
y->x	2	2	1	4	5	6	...

The equation $z = x + y$ means $\forall n. z_n = x_n + y_n$.

Time is logical: inputs x and y arrive "at the same time"; the output z is produced "at the same time"

Example: the heater controller [2]

Model of the heater

- u is the command. $u = \mathit{true}$ (heat); $u = \mathit{false}$ (not heat)
- α, β, c are parameters; ext is the outside temperature.
- The speed temp' is defined below:

$$\mathit{temp}' \;=\; \alpha(c - \mathit{temp}) \text{ if } u \quad \beta(\mathit{ext} - \mathit{temp}) \text{ otherwise}$$

We discretize (with a step h)

temp' is approximated by the difference $(\mathit{temp}_{n+1} - \mathit{temp}_n)/h$

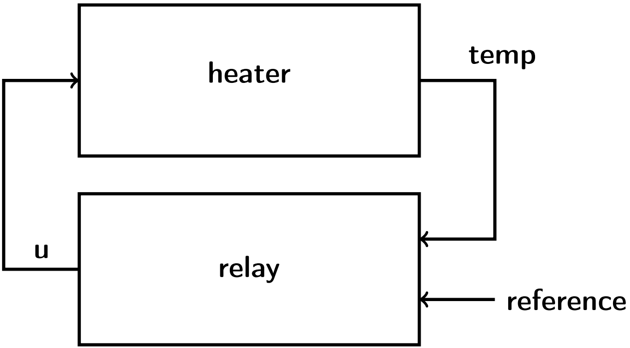
Discrete controller (relay)

$$\begin{array}{rcl} u_n &=& \mathit{true} \text{ if } \mathit{temp}_n < \mathit{low} \quad \mathit{false} \text{ if } \mathit{temp}_n > \mathit{high} \\ u_n &=& \mathit{false} \text{ if } n = 0 \text{ otherwise } u_{n-1} \end{array}$$

---

[2] Example given by Nicolas Halbwachs at CdF (2010).

Feedback loop



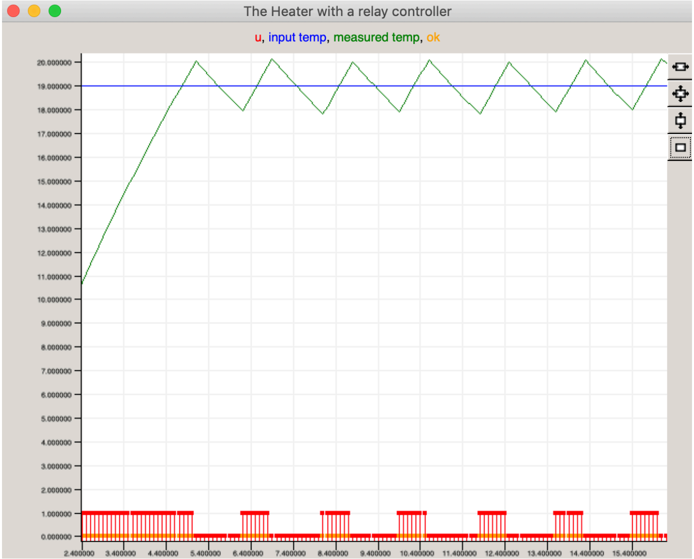
```
(* Integration Euler *)
let node euler(h)(x0, xprime) = x where
  rec x = x0 -> pre(x +. h *. xprime)

(* Heater model *)
let node heat(h)(c, alpha, beta, temp_ext, temp0, u) = temp
 where
  rec temp =
    euler(h)(temp0,
             if u then alpha *. (c -. temp)
             else beta *. (temp_ext -. temp))

(* Relay *)
let node relay(low, high, v) = u where
  rec u = if v < low then true
          else if v > high then false
          else false -> pre u
```

```
let low = 1.0
let high = 1.0

let c = 50.0

let alpha = 0.1
let beta = 0.1

let h = 0.1

(* Main program *)
let node system(reference) = (u, temp) where
  rec
      u = relay(reference -. low, reference +. high, temp)
  and
      temp = heater(h)(c, alpha, beta, 0.0, 0.0, u)
```

Demo [3]

[3] If you have an access to the repo `git@gitlab.inria.fr:parkas/zelus.git`, see file `examples/heater/heat.zls` in branch `v2`.

A single run


The Heater with a relay controller
u, input temp, measured temp, ok
20.000000
19.000000
18.000000
17.000000
16.000000
15.000000
14.000000
13.000000
12.000000
11.000000
10.000000
9.000000
8.000000
7.000000
6.000000
5.000000
4.000000
3.000000
2.000000
1.000000
0.000000
2.400000
3.400000
4.400000
5.400000
6.400000
7.400000
8.400000
9.400000
10.400000
11.400000
12.400000
13.400000
14.400000
15.400000

The model is discrete-time

Essentially a Lustre program.

The choice of h, the integration scheme are hardwired in the model.

If h is too big, the simulation is unprecise; too small, it is slow.

If the ODE is more complex (e.g., non linear), the forward Euler scheme must be replaced by a more complicated integration scheme.

Can we write a model of a higher level, with an explicit ODE; using an external off-the-shelf solver for simulating it?

possibly composed with a discrete-time model (e.g., software) or other continuous-time models.

...+ ODEs + zero-crossings

The model of the heater in continuous-time. Essentially the same program.

```
(* Integrator *)
let hybrid int(x0, xprime) = x where
  rec der x = xprime init x0

(* Model of the heater *)
let hybrid heater(c, alpha, beta, temp_ext, temp0, u) = temp
  where rec temp =
            int(temp0,
                if u then alpha *. (c -. temp)
                else beta *. (temp_ext -. temp))

(* relay *)
let hybrid relay(low, high, v) = u where
  rec u = present
          | up(low -. v) -> true
          | up(v -. high) -> false init (v < high)
```

```
let low = 1.0
let high = 1.0

let c = 50.0

let alpha = 0.1
let beta = 0.1

(* Main program *)
let hybrid system(reference) = (u, temp) where
  rec
      u = relay(reference -. low, reference +. high, temp)
  and
      temp = heater(c, alpha, beta, 0.0, 0.0, u)
```

`u` is piecewise constant; it changes every 0.1 second.

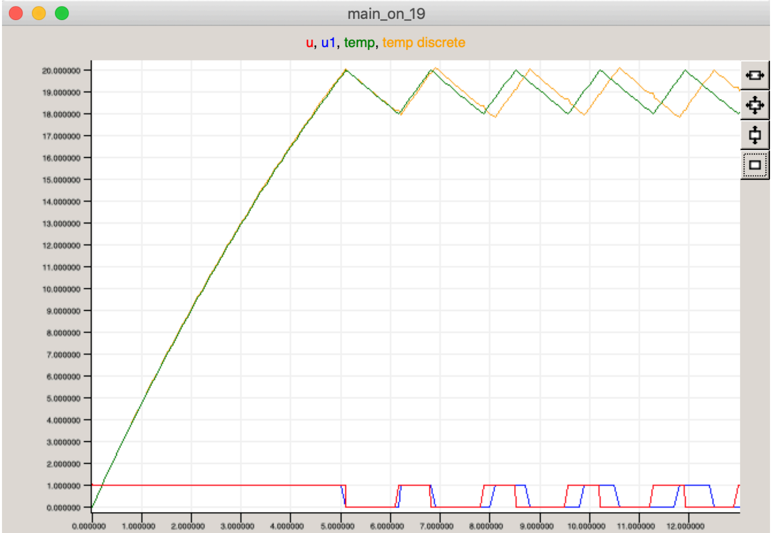
An integrator (construct `der`) breaks an instantaneous dependences exactly like the synchronous register does (construct `pre`).

```
(* The same with a discrete-time controller *)
(* periodically sampled *)
let hybrid system_with_sampled_relay(reference) = (u, temp) where
  rec
      u = present
            (period (0.1)) ->
               Heat.relay(reference -. low, reference +. high,
                          temp)
          init false
  and
      temp = heater(c, alpha, beta, 0.0, 0.0, u)
```

Demo [4]

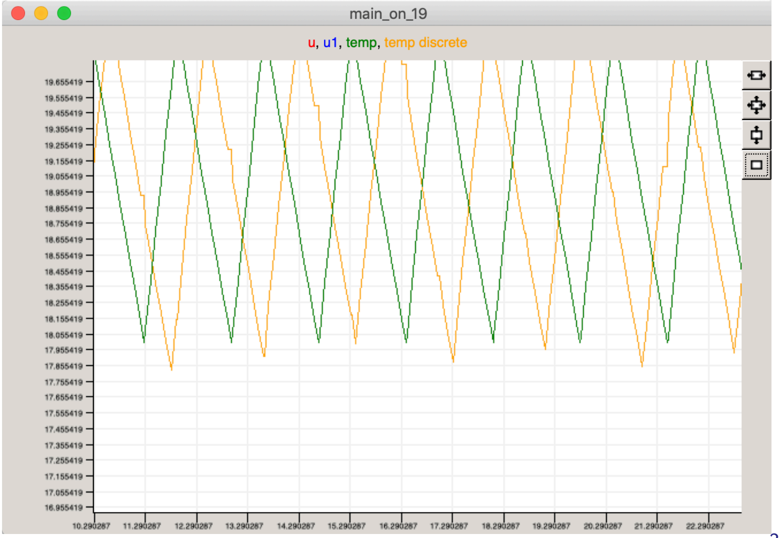
[4] See file `examples/heater/heatc.zls` in branch `v2`.

A single run



main_on_19
u, u1, temp, temp discrete
20.000000
19.000000
18.000000
17.000000
16.000000
15.000000
14.000000
13.000000
12.000000
11.000000
10.000000
9.000000
8.000000
7.000000
6.000000
5.000000
4.000000
3.000000
2.000000
1.000000
0.000000
0.000000
1.000000
2.000000
3.000000
4.000000
5.000000
6.000000
7.000000
8.000000
9.000000
10.000000
11.000000
12.000000

A single run


main_on_19
u, u1, temp, temp discrete
19.655419
19.555419
19.455419
19.355419
19.255419
19.155419
19.055419
18.955419
18.855419
18.755419
18.655419
18.555419
18.455419
18.355419
18.255419
18.155419
18.055419
17.955419
17.855419
17.755419
17.655419
17.555419
17.455419
17.355419
17.255419
17.155419
17.055419
16.955419
10.290287
11.290287
12.290287
13.290287
14.290287
15.290287
16.290287
17.290287
18.290287
19.290287
20.290287
21.290287
22.290287

In brief

A discrete-time signal = a stream.

A continuous-time signal = an "hyper stream" (Suenaga, Sekine, and Hasuo *[POPL'13]*).

A system = a streams/hyper streams function.

New features w.r.t Lustre:

- `der` defines a signal by its derivative;
- `up` defines a zero-crossing event.

Static typing to reject monsters.

The compiler generates sequential code (OCaml);

linked to an ODE solver (Sundials Cvode).

Can we define a standard library of control block, e.g., that of Simulink, such that the definition is a formal specification?

A comprehensive set has been written in Zelus, in both discrete-time and continuous-time *[EMSOFT'17]*.

A version has also been defined, for discrete-time blocks only, in Scade 6 (ANSYS/Esterel-Technologies).

Discrete-time integrator (lib. "Discrete-time blocks")

E.g., forward/backward Euler, Trapezoidal, with/without saturation.

```
let node forward_euler(t)(k, x0, u) = output where
  rec output = x0 fby (output +. (k *. t) *. u)

let node backward_euler(t)(k, x0, u) = output where
  rec output = x0 -> pre output +. (k *. t) *. u
```

Compiling it with `zeluc -i -ic example.zls` we get the type signature:

```
val forward_euler : float -S-> float * float * float -D-> float
val backward_euler : float -S-> float * float * float -D-> float
```

and the causality type signatures that express the input/output dependences.

```
val forward_euler : {'a < 'b}. 'b -> 'b * 'a * 'b -> 'a
val backward_euler : {}. 'a -> 'a * 'a * 'a -> 'a
```

PID (discrete time)

p proportional gain; *i* integral gain; *d* derivative gain; *n* filter coefficient.

Transfert function:

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

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

val pid_par :
  float -S-> float
        -S-> float * float * float * float -D-> float
val pid_par :
  {'a < 'b}. 'b -> 'a -> 'a * 'b * 'a * 'a -> 'a
```

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

```
let node filter(n)(h)(k, u) = (u -. u fby u) /. h
```

Otherwise, approximate with a low-pass filter. It also depend on the integration method.

```
(* Apply a low pass filter on the input *)
(* see Book by Astrom & Murray, 2008). *)
let node filter(n)(h)(k, u) = udot where
  rec udot = n *. (k *. u -. f)
  and f = forward_euler(h)(n, 0.0, udot)
```

For the PID, we should write $n \times m$ versions, if n is the number of possible integration methods and m is the possible number of filtering methods (which can use a different integration scheme).

This has to be multiplied if we want to deal with a single input, an input vector, a input matrix, etc.

A more generic version

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

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

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

val generic_pid :
  {'a < 'b; 'c < 'b, 'd, 'a, 'e, 'f}.
    ('e -> 'd * 'c * 'a -> 'b) ->
      ('e -> 'f * 'a -> 'b) -> 'e -> 'b * 'd * 'f * 'a -> 'b

val pid_forward :  {'a < 'b}. 'a -> 'a * 'b * 'b * 'a -> 'a
val pid_backward : {'a < 'b}. 'a -> 'a * 'a * 'b * 'a -> 'a
```

In continuous time?

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

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

let hybrid gfilter(n)(int)(k, u) = udot where
   rec udot = n *. (u -. f)
   and f = run int (k, 0.0, udot)

let hybrid pid_c(n)(p, i, d, u) =
  gpid_c(int)(gfilter(n)(int))(p, i, d, u)

val pid_c : {'a < 'b}. 'a -> 'a * 'b * 'b * 'a -> 'a
```

Typing

An ML type system. The first order version in *[LCTES'11]*.

k indicates whether a function is static, combinatorial, discrete time or continuous time.

$$
\begin{array}{lcl}
bt & ::= & \texttt{float} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{zero} \\
t & ::= & t \xrightarrow{k} t \mid t \times t \mid \beta \\
\sigma & ::= & \forall \beta_1, ..., \beta_n . t \\
k & ::= & \texttt{D} \mid \texttt{C} \mid \texttt{A} \mid \texttt{S}
\end{array}
$$

$$\texttt{S} \leq \texttt{A} \quad \texttt{A} \leq \texttt{D} \quad \texttt{A} \leq \texttt{C}$$

Initial conditions

$$
\begin{array}{lcl}
(+) & : & \texttt{int} \times \texttt{int} \xrightarrow{\texttt{A}} \texttt{int} \\
\texttt{if} & : & \forall \beta . \texttt{bool} \times \beta \times \beta \xrightarrow{\texttt{A}} \beta \\
(=) & : & \forall \beta . \beta \times \beta \xrightarrow{\texttt{D}} \texttt{bool} \\
\texttt{pre}(\cdot) & : & \forall \beta . \beta \xrightarrow{\texttt{D}} \beta \\
\texttt{up}(\cdot) & : & \texttt{float} \xrightarrow{\texttt{C}} \texttt{zero}
\end{array}
$$

Simple but limited.

It is “block based”, no “signal based”. We distinguish discrete-time blocks (sort `D`) from continuous-time blocks (sort `C`).

There is no polymorphism of sorts. It forces to write the PID, the filters (FIR/IIR), the transfert functions, the block “ABCD”, etc. twice.

one in discrete time, one in continuous time.

this is not satisfactory. Version 3?

Causality analysis

Characterize by a type the input/output dependences of a function.

Only express instantaneous dependences: given an output, what are the inputs that are necessary.

This is enough to reject causality loops and ensure that sequential code can be generated.

The intuition: every loop must cross a delay (discrete time) or an integrator (continuous time).

Causality

An ML type system with sub-typing.

A first-order version was presented at *[HSCC'14, NAHS'17]*.

$$
\begin{array}{lcl}
bt & ::= & \alpha \\
t & ::= & bt \mid t \times t \mid t \longrightarrow t \mid \alpha \\
\sigma & ::= & \forall C.\alpha_1, ..., \alpha_n.t \\
 & & \\
C & ::= & \{\alpha_i < \alpha_j\}_{i,j \in I}
\end{array}
$$

C must define a partial order (graph with no cycle)

Initial conditions

$$
\begin{array}{lcl}
(+) & : & \forall \alpha.\alpha \times \alpha \longrightarrow \alpha \\
\texttt{if} & : & \forall \alpha.\alpha \times \alpha \times \alpha \longrightarrow \alpha \\
\texttt{pre}(\cdot) & : & \forall \alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\}.\alpha_1 \longrightarrow \alpha_2 \\
\cdot\,\texttt{fby}\,\cdot & : & \forall \alpha_1, \alpha_2 : \{\alpha_1 < \alpha_2\}.\alpha_1 \times \alpha_2 \longrightarrow \alpha_1 \\
\texttt{up}(\cdot) & : & \forall \alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\}.\alpha_2 \longrightarrow \alpha_1
\end{array}
$$

Problem: preceding (subtyping) constraints that are generated may be huge and unreadable. They must be simplified.

This is a well studied problem: Aiken & Wimmers, Smith & Trifonov, Pottier, Castagna, etc.

We apply a different algorithm, which uses Input/output relations by Raymond et al. *[EMSOFT'09]*.

On some examples, it gives shorter signatures.

Conclusion

Version 2, with higher-order, array iterrators, and quite a few compilation improvements starts working; available in source code (GitLab INRIA).

The reference manual is outdated (version 1); to be done soon.

The web page (with binary) too.

An experimental library to do *Probabilistic Reactive Programming* (joint work with Guillaume Baudart and Louis Mandel (IBM Watson).

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