Mixing discrete-time and continuous-time signals

Marc Pouzet

ENS

Marc.Pouzet@ens.fr

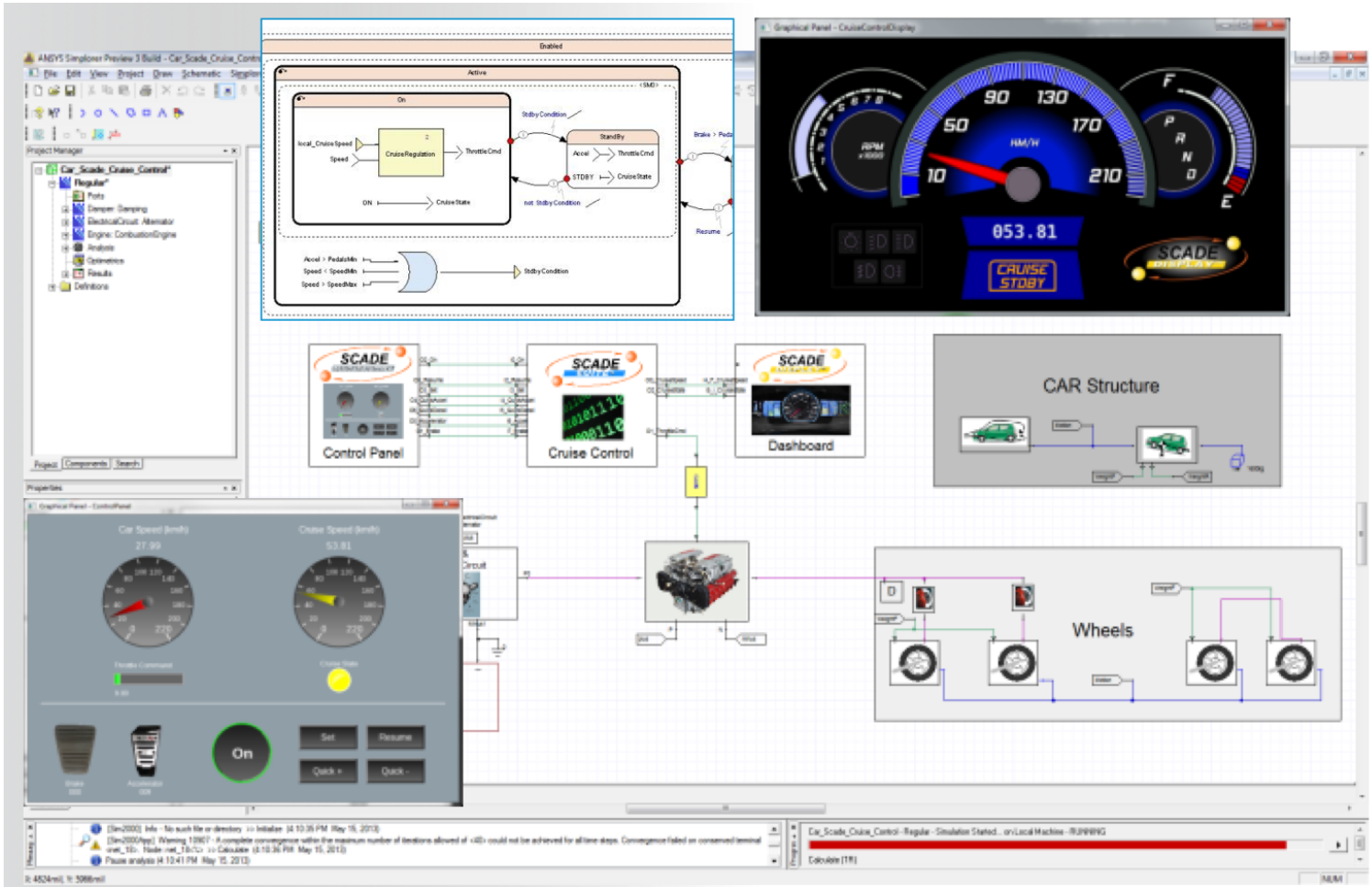
Course notes, MPRI, November 2024

Lustre describes discrete time models.

What about hybrid discrete/continuous models?

Example: the cruise control in its environment[1]

Model the whole system: the controller and the plant.



[1] Image from ANSYS/Esterel-Technologies

A hybrid system = a system with mix of discrete-time and continous-time signals, discrete and continuous changes.

E.g., a software model + a model of the physics,

a continuous-time model with modes and/or discontinuous jumps.

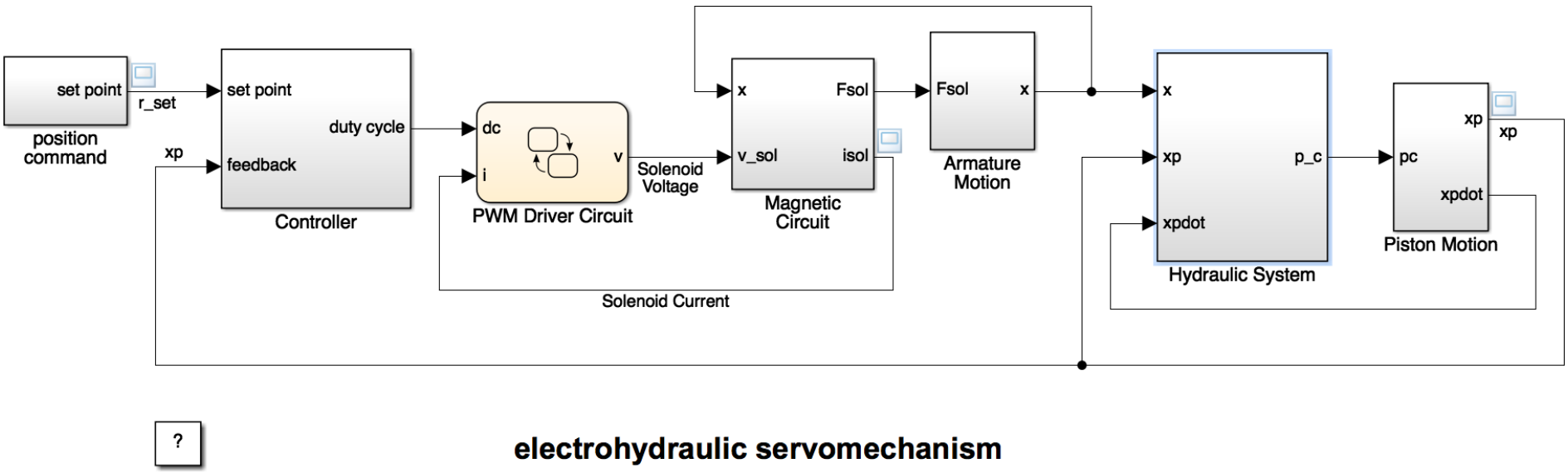
We focus on languages to write executable models.

This is complementary to the formal verification problem.

E.g., Lustre and SCADE are restricted to discrete-time models.

Otherwise, e.g., Simulink/Stateflow, Modelica, Ptolemy, Scicos.

For example [2]



[2] Image taken from the standard distribution of Simulink

In those languages, the compiler does a lot

Produces executable code for efficient simulation and/or an embedded implementation.

Detect/reject statically certain models.

Does non trivial transformations.

E.g., static scheduling, inlining, rewriting, separation of the continuous/discrete-time part, data representations, link with an ODE solver.

A precise static/dynamic semantics is necessary to argue that the compiler is correct.

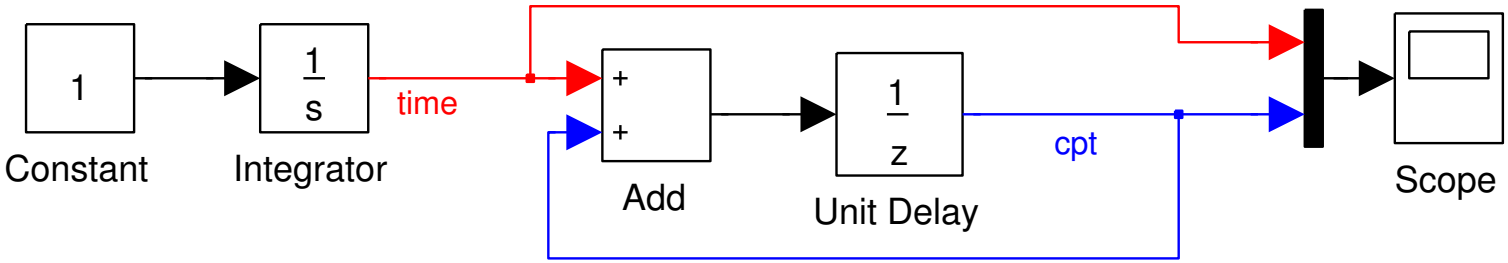
Where are the monsters?

Besides difficulties related to numerical approximation of ODEs,

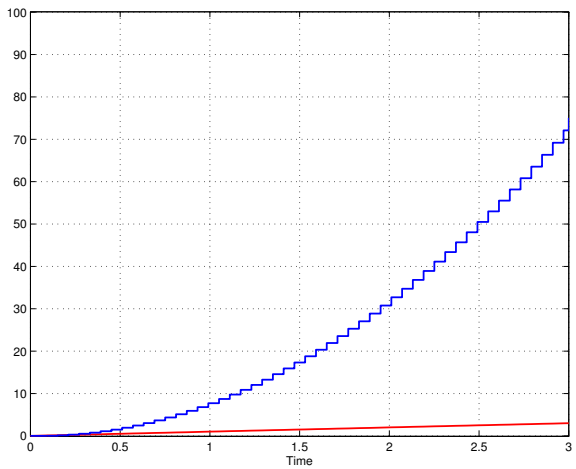
some model mix discrete time and continuous time in an ambiguous or wrong manner;

They are fragile, hard to reuse, their simulation is difficult to reproduce.

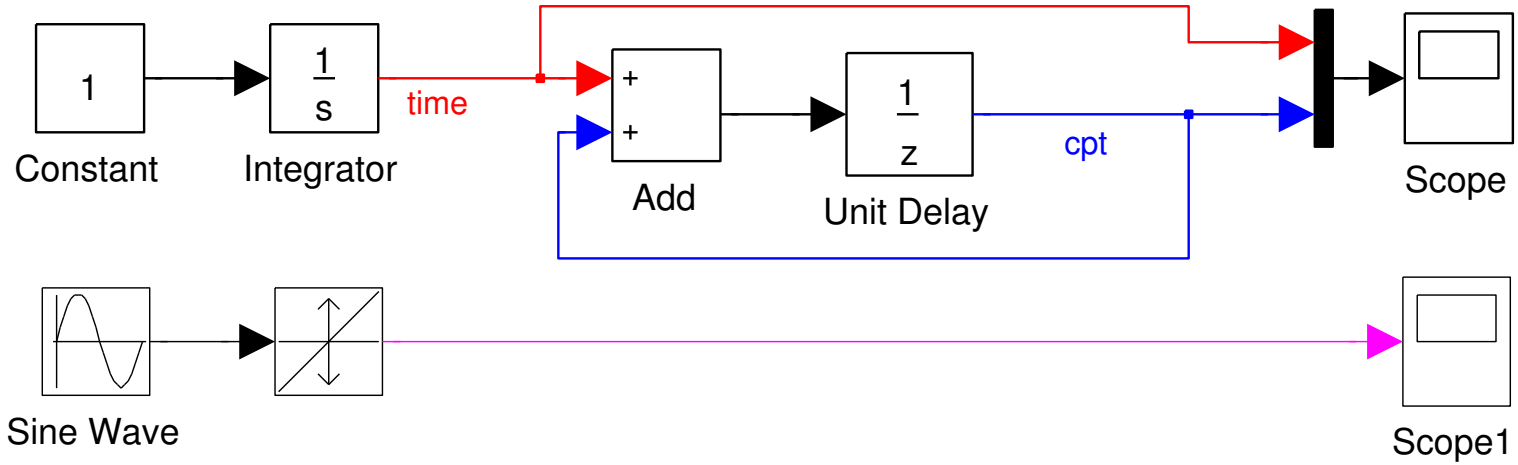
Typing discrete/continuous issues (in Simulink)



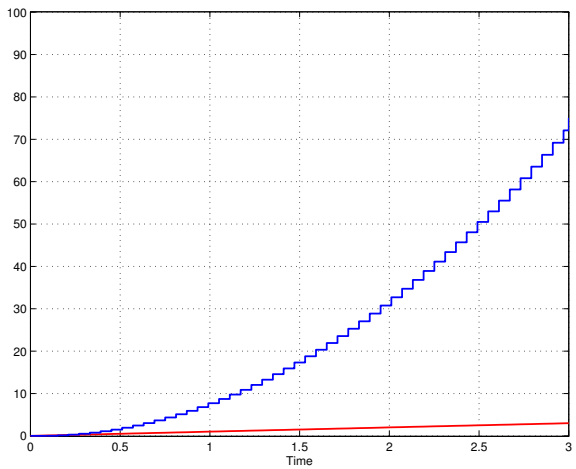
Basic model



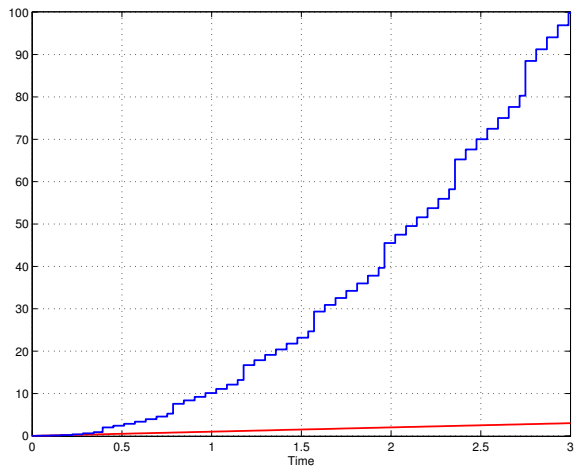
Typing discrete/continuous issues (in Simulink)



Basic model



with Sine Wave



Discrete time is not the logical time of Lustre.

It is that of the simulation engine.

Some blocks explicitly refer to the *major step* of the simulation engine.

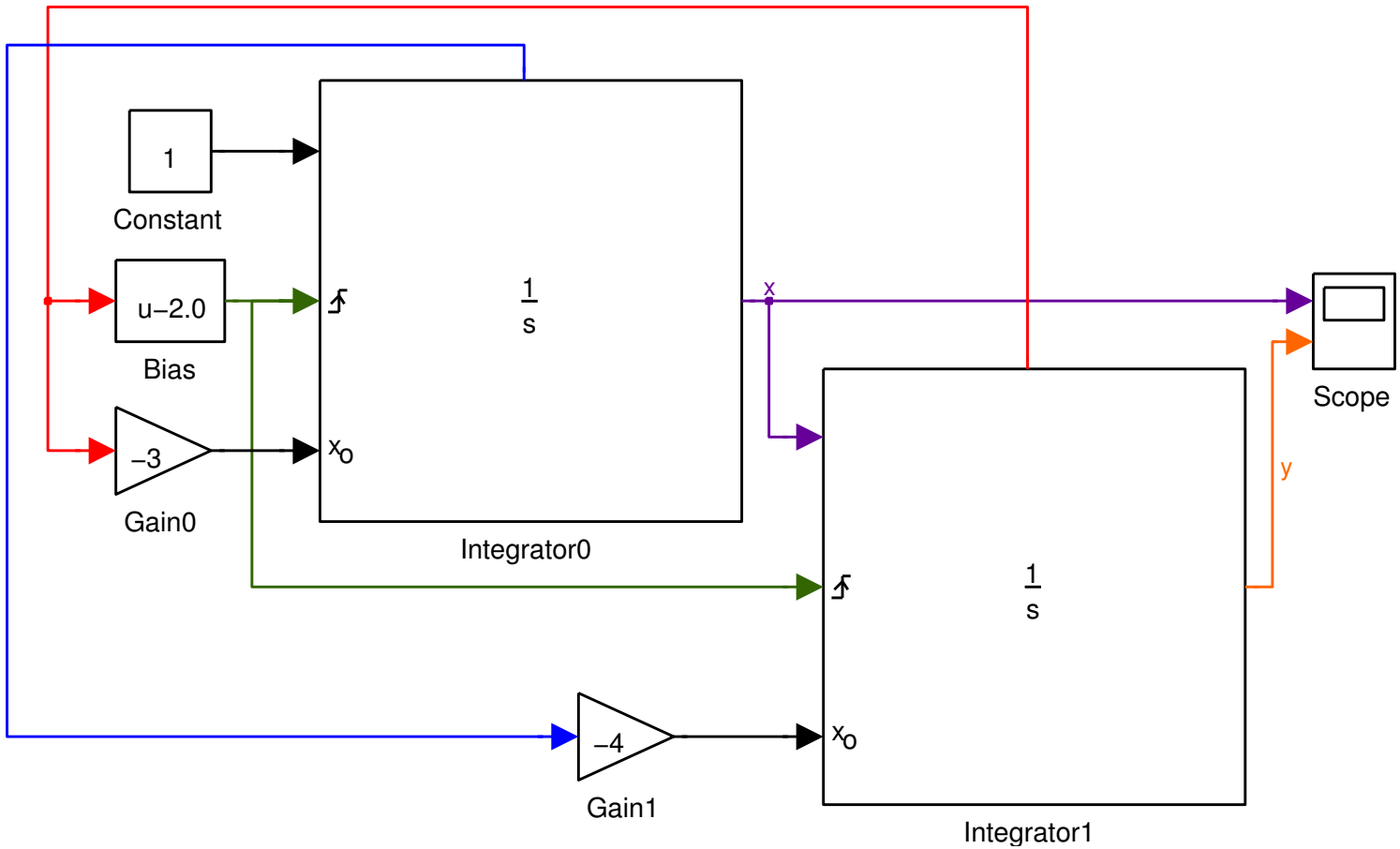
E.g., the derivative, transport delay, backlash, memory block.

They should be used very carefully when applied to continuous-time imputs.

Yet, if we forbid them, some systems are difficult to write.

E.g., the memory block is necessary to break certain algebraic loops and get sequential code.

Causality issue: the Simulink state port



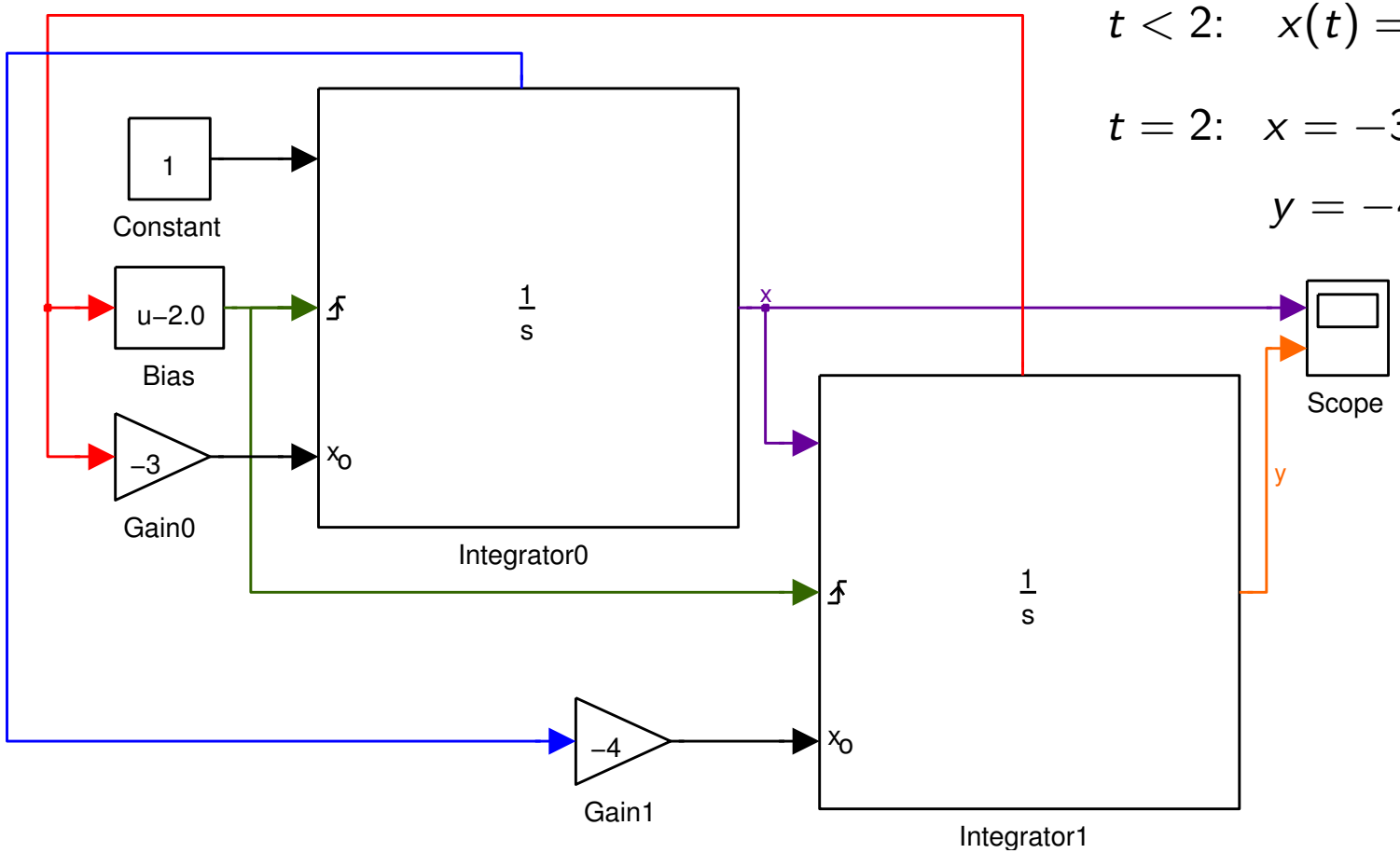
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.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

Causality issue: the Simulink state port

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

$$t = 2: \quad x = -3 \cdot \text{last}\, y = -6,$$

$$y = -4 \cdot \text{last}\, x = -8$$



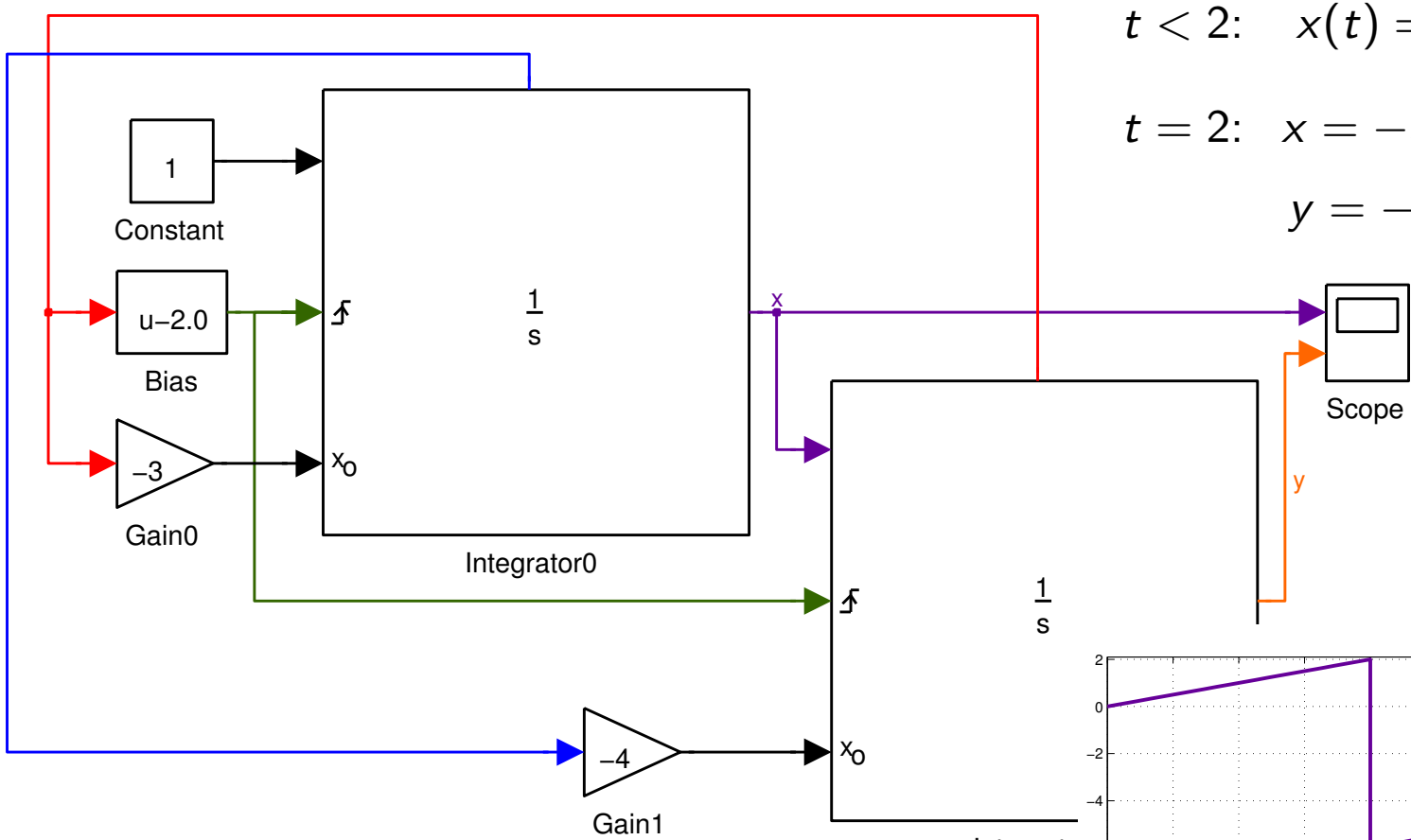
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.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

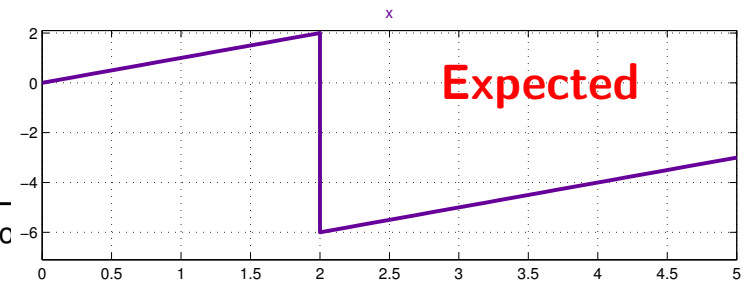
Causality issue: the Simulink state port

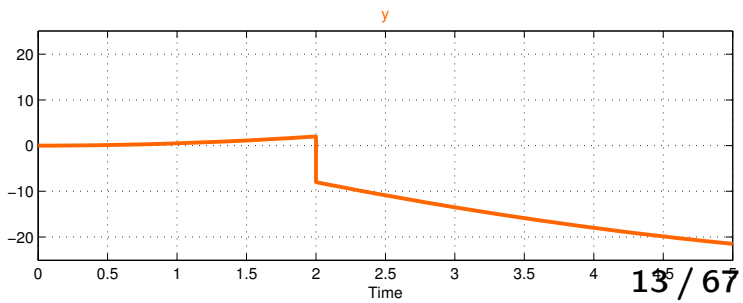
$$t < 2: \quad x(t) = t,\ y(t) = \frac{t^2}{2}$$

$$t = 2: \quad x = -3 \cdot \text{last}\, y = -6,$$

$$y = -4 \cdot \text{last}\, x = -8$$







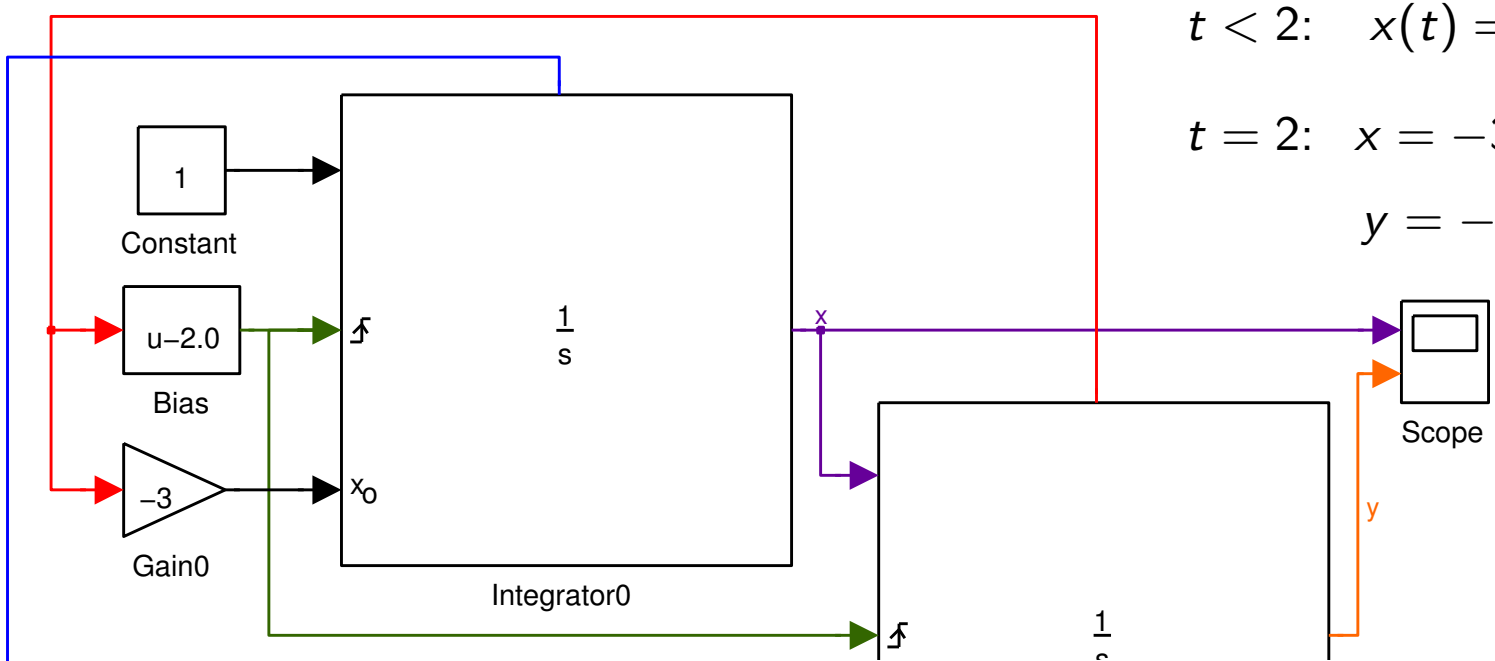
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.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

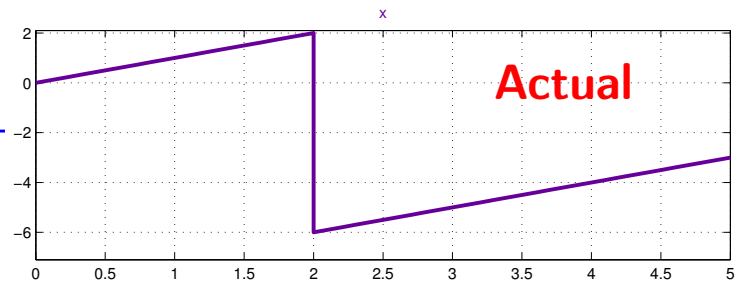
Causality issue: the Simulink state port

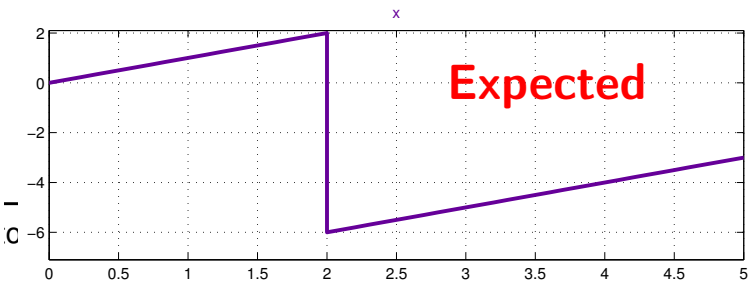
$$t < 2: \quad x(t) = t,\ y(t) = \frac{t^2}{2}$$

$$t = 2: \quad x = -3 \cdot \text{last}\, y = -6,$$

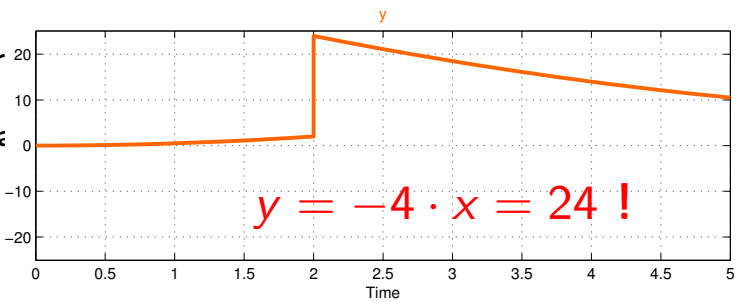
$$y = -4 \cdot \text{last}\, x = -8$$

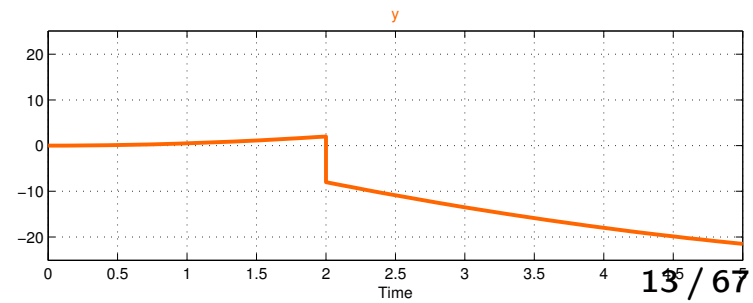






The output of the st
the block's standard
case. If the block is r
put of the state port
at the block's standa
reset.
–Simulink Reference





Excerpt of C code produced by RTW (release R2009)

```
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

$x = -3 \cdot \text{last } y$

After assignment: integrator state contains the new value

$y = -4 \cdot x$

So, y is updated with the new value of x

There is a problem in the treatment of causality.

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;
```

OpenModelica 1.9.2beta1
(r24372) Also in Dymola

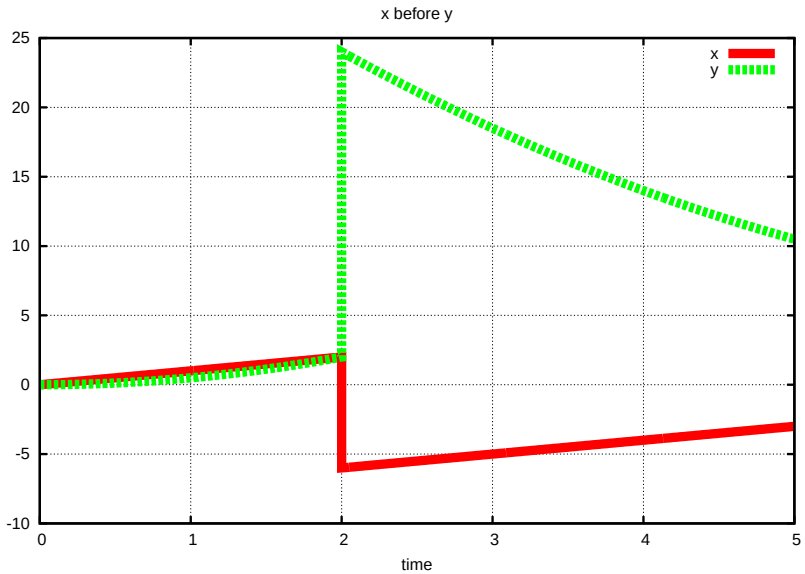
Causality: Modelica example

```
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;
```



OpenModelica 1.9.2beta1 (r24372) Also in Dymola

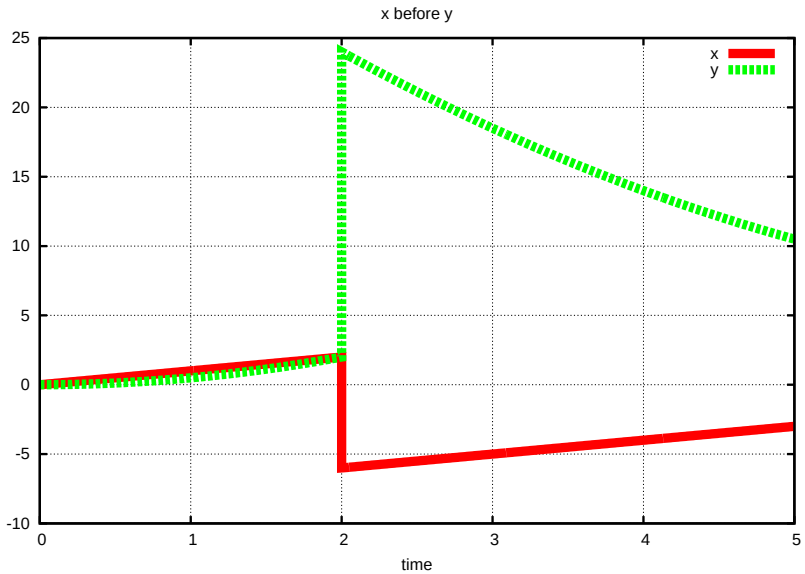
Causality: Modelica example

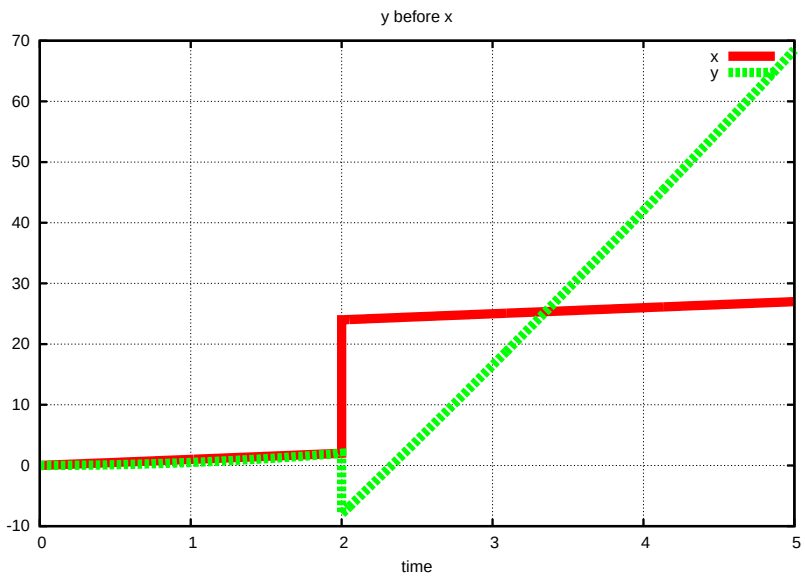
```
model scheduling
  Real x(start = 0);
  Real y(start = 0);
equation

  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;
```





OpenModelica 1.9.2beta1 (r24372) Also in Dymola

Wrongly typed models

Design type systems to statically reject bizarre models.

Can we formally ensure a property like:

"Well typed programs cannot go wrong" (Robin Milner) ?

What is a wrong model/program?

Recycle/extend principles and techniques
developed for synchronous languages.

Zélus [3]

Zélus = Lustre + ODEs + zero crossings

[3] http://zelus.di.ens.fr

In brief

Write data-flow equations (Lustre)

combined ordinary differential equations;

type checking to reject certain bizarre models;

compile to sequential code;

paired with an off-the-shelf solver when the model has ODEs.

The two examples in Zélus

```
let hybrid wrong() = (time, cpt) where
  rec
    der time = 1.0 init 0.0
  and
    cpt = 0.0 fby (time +. cpt)

>     cpt = 0.0 fby (time +. cpt)
>           ^^^^^^^^^^^^^^^^^^^^
Type error: this is a discrete expression
and is expected to be continuous.


let hybrid causal() = (x, y) where
  rec
    der x = 1.0 init 0.0 reset z -> -3.0 *. last y
  and
    der y = x init 0.0 reset z -> -4. *. last x
  and
    z = up(last y -. 2.0)
```

Zélus = Lustre + ...

A discrete system = a Lustre node.

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 [4]

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:

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

We discretize (with a step h)

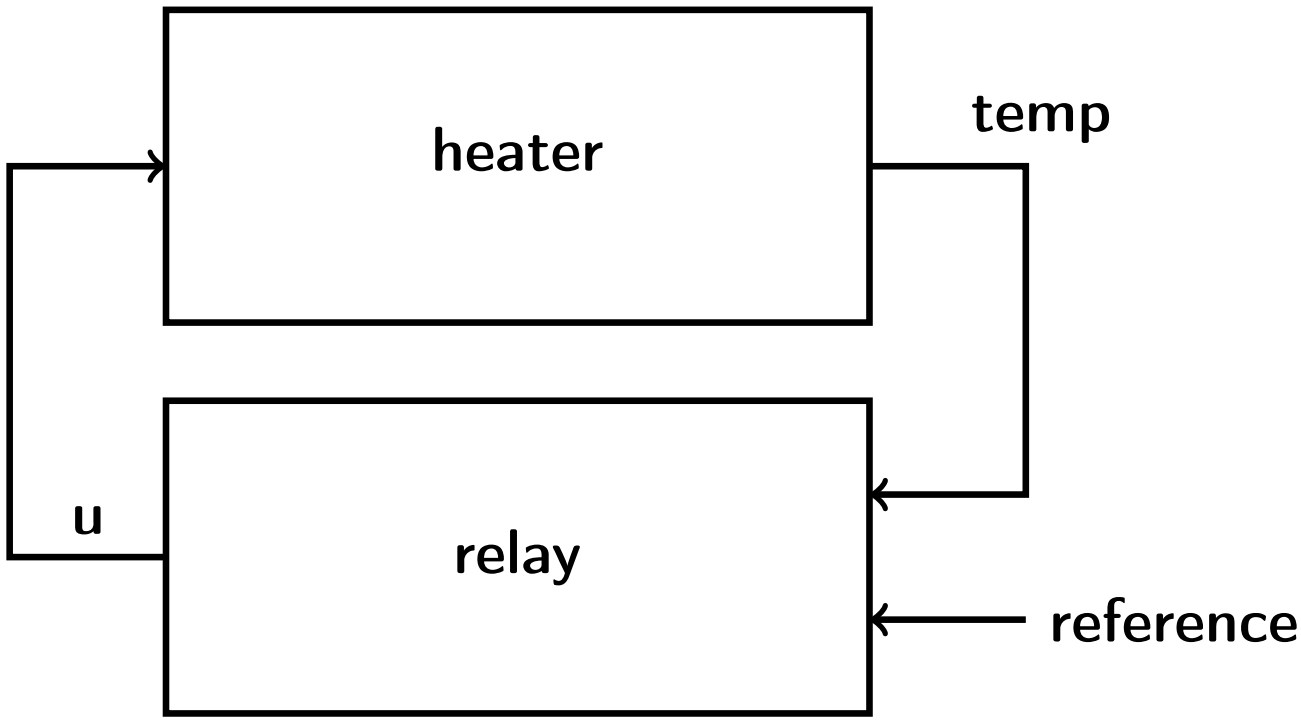
temp' is approximated by the difference $(\mathit{temp}_{n+1} - \mathit{temp}_n)/h$

Discrete controller (relay)

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

[4] 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 > haut 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 main(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

This is a discrete time model

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

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

In particular with a more complicated (non linear) ODE.

Instead, write an ODE; approximate it with an off-the-shelf numerical solver.

...+ ODEs + zero-crossings

The model of the heater in continuous-time.

```
(* 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 < haut)
```

```
let low = 1.0
let high = 1.0

let c = 50.0

let alpha = 0.1
let beta = 0.1

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

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

Demo

Internals

A Non-standard Semantics for Hybrid Modelers [JCSS'12]

We proposed to build the semantics on non-standard analysis.

```
let hybrid f () = y where
  rec
    der y = z init 4.0
  and
    z = 10.0 -. 0.1 *. y
  and k = y +. 1.0
```

defines signals y, z and k, where for all $t \in \mathbb{R}^+$:

$$\frac{dy}{dt}(t) = z(t) \quad y(0) = 4.0 \quad z(t) = 10.0 - 0.1 \cdot y(t) \quad k(t) = y(t) + 1$$

What would be the value of y if computed by an ideal solver taking an infinitesimal step of duration ∂?

${}^\star\mathbb{R}$ and ${}^\star\mathbb{N}$ are the non-standard extensions of $\mathbb{R}$ and $\mathbb{N}$. [5]

An infinitesimal is smaller in absolute value than any real number: $\partial \in {}^\star\mathbb{R}$ is such that $|\partial| < a$, for any positive $a \in \mathbb{R}$. If $x, y \in \mathbb{R}$, $x \approx y$ if $x - y$ is an infinitesimal.

Every hyperreal $x \in {}^\star\mathbb{R}$ possesses a unique standard part $st(x) \in \mathbb{R}$ such that $st(x) \approx x$.

Let $x : \mathbb{R} \mapsto \mathbb{R}$ a $\mathbb{R}$-valued (standard) signal. Then:

- x is continuous at instant $t \in \mathbb{R}$ iff for any $\partial \in {}^\star\mathbb{R}$, $x(t + \partial) \approx x(t)$.
- x is differentiable at instant $t \in \mathbb{R}$ iff there exists an $a \in \mathbb{R}$ such that, for any infinitesimal $\partial \in {}^\star\mathbb{R}$, $(x(t + \partial) - x(t))/\partial \approx a$.
  In that case, $a = x'(t)$.

[5] The paper by Lindstrom [Lin88] is a an introduction to non standard analysis.

The base clock

Let $\partial \in {}^{\star}\mathbb{R}$ be an infinitesimal, i.e., $\partial > 0, \partial \approx 0$.

Imagine that the system is doing a sequence of steps of ∂ duration each.

Define the base clock:

$$0, \partial, 2\partial, 3\partial, 4\partial, \ldots$$

that is:

$$\mathbb{T}_{\partial} = \{t_n = n\partial \mid n \in {}^{\star}\mathbb{N}\}$$

$\mathbb{T}_{\partial}$ inherits its total order from ${}^{\star}\mathbb{N}$.

${}^\star y(n)$ stands for the values of y at instant $n\partial$, with $n \in {}^\star\mathbb{N}$ a non-standard integer.

A differential equation can be now turned into a difference equation:

$$({}^\star x(n+1) - {}^\star x(n))/\partial$$

is the derivative of signal x.

Example:

$$\begin{aligned} {}^\star y(0) &= 4 & {}^\star z(n) &= 10 - 0.1 \cdot {}^\star y(n) \\ {}^\star y(n+1) &= {}^\star y(n) + {}^\star z(n) \cdot \partial & {}^\star k(n) &= {}^\star y(n) + 1 \end{aligned}$$

Non standard semantics [JCSS'12, HSCC'14]

A sub-clock $T \subset \mathbb{T}_\partial$.

What is a discrete clock?

> *A clock T is termed discrete if it is the result of a zero-crossing or a sub-sampling of a discrete clock. Otherwise, it is termed continuous.*

If $T \subseteq \mathbb{T}$, we write ${}^\bullet T(t)$ for the immediate predecessor of t in T and $T^\bullet(t)$ for the immediate successor of t in T.

Signals and clocks

Signals

Let V a set. $V_\perp = V + \{\perp\}$ with $\forall v \in V, \perp \leq v$. $S(V) = \mathbb{T} \mapsto V_\perp$ is the set of signals.

A signal $x : T \mapsto V_\perp$ is a total function from $T \subseteq \mathbb{T}$ to $V_\perp$. Moreover, for all $t \notin T, x(t) = \perp$.

Sampling

Let $Bool = \{false, true\}$ and $x : T \mapsto Bool_\perp$. The sampling of T according to x, written $T \,\mathtt{on}\, x$ is the subset of instants:

$$T \,\mathtt{on}\, x = \{t \mid (t \in T) \wedge (x(t) = true)\}$$

Note that $T \,\mathtt{on}\, x \subseteq T$, it is also totally ordered.

Semantics of basic operations

Replay the classical semantics of a synchronous language.

An ODE with reset

An ODE $\mathrm{der}\ x = e\ \mathtt{init}\ e_0\ \mathtt{reset}\ z \longrightarrow e_1$ is interpreted as a stream equation.

$$
\begin{aligned}
{}^\star x(0) &= {}^\star e_0(0) \\
{}^\star x(n) &= \mathtt{if}\ {}^\star z(n)\ \mathtt{then}\ {}^\star e_1(n)\ \mathtt{else}\ {}^\star x(n-1) + \partial \cdot {}^\star e(n-1)
\end{aligned}
$$

Zero-crossing $\mathrm{up}(x)$

It is interpreted as a edge-front detection.

$$
\begin{aligned}
{}^\star \mathrm{up}(x)(0) &= \mathit{false} \\
{}^\star \mathrm{up}(x)(n+1) &= ({}^\star x(n-1) < 0) \wedge ({}^\star x(n) \geq 0) \wedge ({}^\star x(n+1) \geq 0)
\end{aligned}
$$

These definitions extend to the case where they are not defined on the base clock but on a subclock T.

Fixpoint Semantics: Principle [HSCC'14]

Define semantics as mutual least fixpoint of set of monotonous operators (one for each expression or definition). Semantics of expression e:

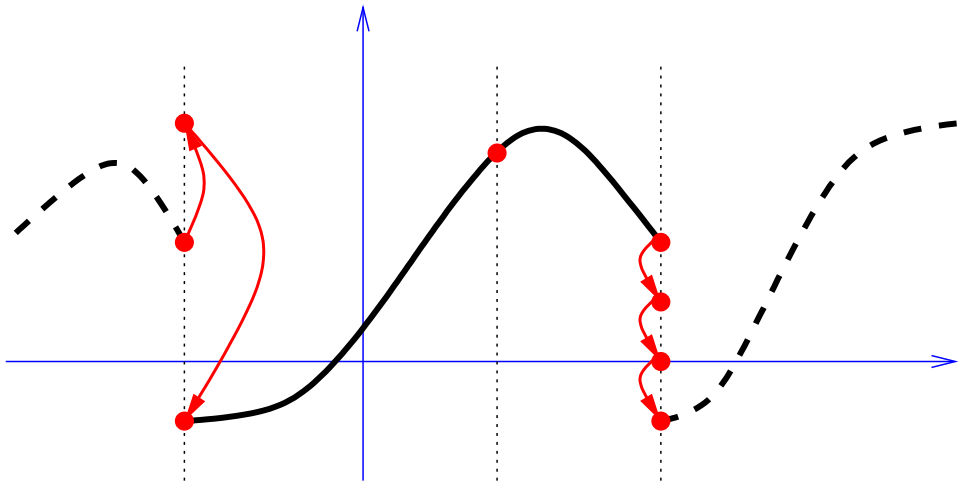
$$^\star[\![e]\!]^\rho_G(T)(t) \quad = \quad (v, z)$$

With:

- $t \in \mathbb{T} = \{n\partial | n \in {}^\star\mathbb{N}\}$ non standard date
- $T \subseteq \mathbb{T}$: set of dates of evaluation of expression T is a discrete clock for a Lustre expression.
- $v \in {}^\star V \uplus \{\bot\}$: $\bot$ if undefined, $\bot < v \in V$ (flat order)
- $z \in \mathbb{B}$: true iff zero-crossings occurs in e at instant t
- Signals: $S({}^\star V) = \mathbb{T} \to {}^\star V_\bot$
- $G : L_g \to S({}^\star V) \to S({}^\star V)$ maps global function names to semantics
- $\rho \quad : \quad L \to S({}^\star V)$ maps local variable names to semantics

Non-standard time vs. Super-dense time

- Maler et al., Lee et al. super-dense time modeling $\mathbb{R} \times \mathbb{N}$



Super-dense time

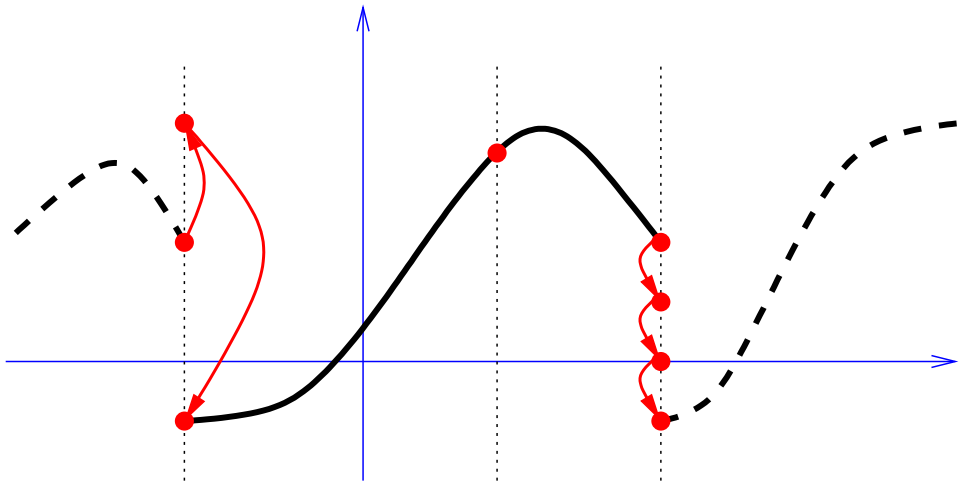
Define the time index $\mathbb{S} = \mathbb{R} \times \mathbb{N}$. A signal as a total function $\mathbb{R} \times \mathbb{N} \mapsto V_\perp$.

Instants are lexically ordered: $(t, n) <_{\mathbb{S}} (t', n')$ iff $t <_{\mathbb{R}} t'$, or $t = t'$ and $n <_{\mathbb{N}} n'$.

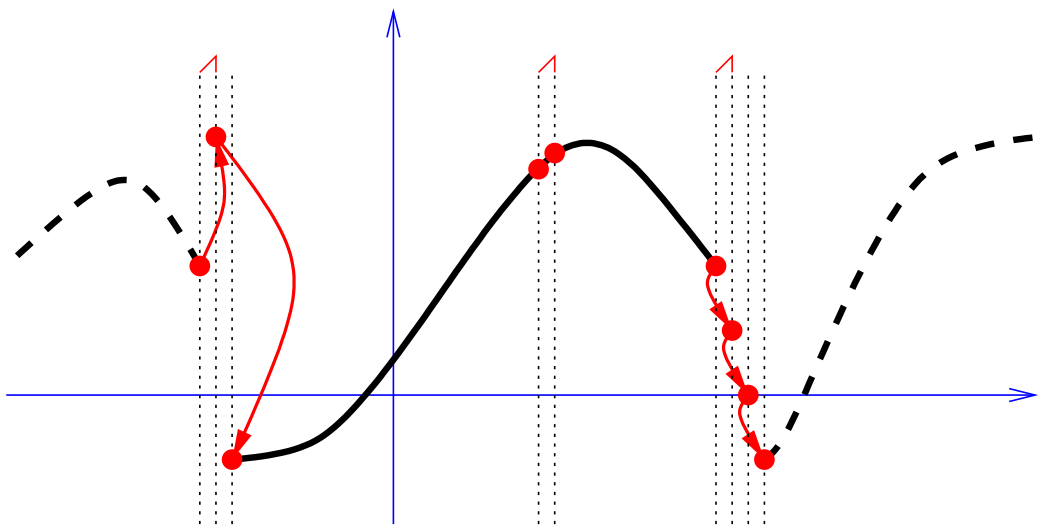
For any (t, n) and (t, n') where $n \leq_{\mathbb{N}} n'$, if $x(t, n') \neq \perp$ then $x(t, n) \neq \perp$.

Non-standard time vs. Super-dense time

- Edward Lee & al. super-dense time modeling $\mathbb{R} \times \mathbb{N}$



- non-standard time modeling $\mathbb{T}_\partial = \{n\partial \mid n \in {}^\star\mathbb{N}\}$



Standardisation [HSCC'14]

A signal in non standard time has a standard part, in super-dense time.

Read HSCC'14 to see how it is done.

A signal in non standard time is denominated an *Hyperstream* by Hasuo et al. [POPL'13].

NSA is modular and helps to design static analyses to reject some meaningless programs.

Is the use of NSA more than a "style exercice"? Is-it useful to prove an interesting theorem?

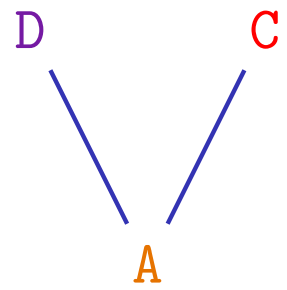
Yes! when a program is well-typed, signals are continuous during integration, provided imported operators are also continuous.

Basic typing [LCTES'11]

A simple ML type system with effects.

The type language

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



Initial conditions

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

The Example in Zélus

```
let hybrid wrong() = (time, cpt) where
  rec
    der time = 1.0 init 0.0
  and
    cpt = 0.0 fby (time +. cpt)

  File ‘‘example.zls’, line 5, character 10-31:
  > cpt = 0.0 fby (time +. cpt)
  > ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  >
  Type error: this is a discrete expression and is expected
  to be continuous.
```

Causality [HSCC'14]

A simple ML type system with sub-typing constraints.

The type language

$$
\begin{array}{lcl}
bt & ::= & \alpha \\
t & ::= & bt \mid t \times t \mid \alpha \\
\sigma & ::= & \forall C.\alpha_1, ..., \alpha_n.t \xrightarrow{k} t \\
k & ::= & \mathtt{D} \mid \mathtt{C} \mid \mathtt{A}
\end{array}
$$

$$
C ::= \{\alpha_i < \alpha_j\}_{i,j \in I}
$$

C must define a partial order (cycle free).

Initial conditions

$$
\begin{array}{lcl}
(+) & : & \forall \alpha.\alpha \times \alpha \xrightarrow{\mathtt{A}} \alpha \\
\mathtt{if} & : & \forall \alpha.\alpha \times \alpha \times \alpha \xrightarrow{\mathtt{A}} \alpha \\
\mathtt{pre}(\cdot) & : & \forall \alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\}.\alpha_1 \xrightarrow{\mathtt{D}} \alpha_2 \\
\cdot\ \mathtt{fby}\ \cdot & : & \forall \alpha_1, \alpha_2 : \{\alpha_1 < \alpha_2\}.\alpha_1 \times \alpha_2 \xrightarrow{\mathtt{D}} \alpha_1 \\
\mathtt{up}(\cdot) & : & \forall \alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\}.\alpha_2 \xrightarrow{\mathtt{C}} \alpha_1
\end{array}
$$

The Example in Zélus

```
let hybrid causal() = (x, y) where
  rec
    der x = 1.0 init 0.0 reset z -> -3.0 *. last y
  and
    der y = x init 0.0 reset z -> -4. *. last x
  and
    z = up(last y -. 2.0)
```

The main theorem [HSCC'14, NAHS'17]

When programs are well typed and causally correct,

signals are continuous during integration

provided imported functions are.

Compilation

Objective: simulate a hybrid model

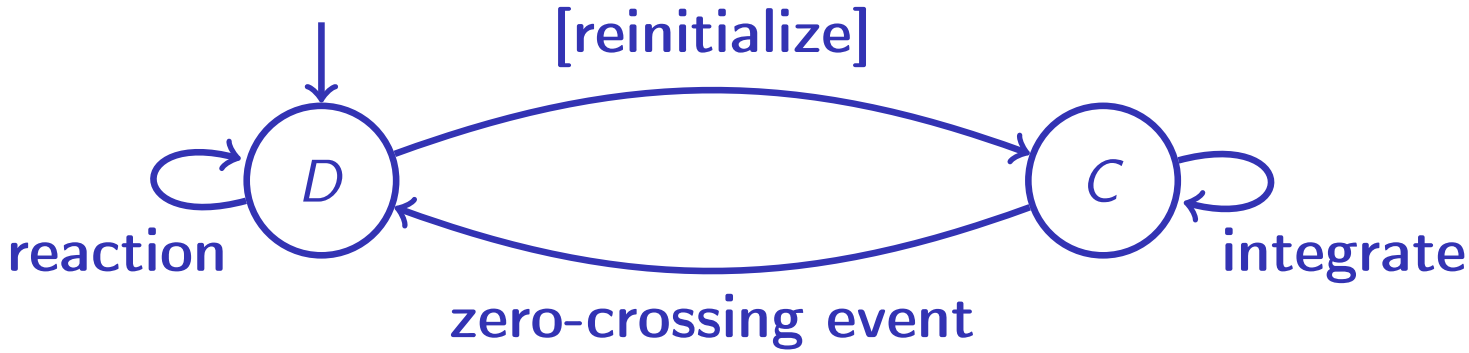
- Generate (statically scheduled) sequential code.
- For hybrid models, this code has to be paired with an ODE solver and a zero-crossing detection mechanism.
- The Zelus run-time uses the off-the-shelf solver SUNDIALS CVODE.[6]
- Uses the SundialML binding [7].
- And the classical Illinois algorithm, implemented in OCaml, for zero-crossings detection.

[6] `https://computing.llnl.gov/projects/sundials`
[7] `https://github.com/inria-parkas/sundialsml`

How does the simulation of a hybrid system works?

A simulation loop alternates between two behaviors [BCP+15]:

- A "discrete" phase (D) where possible input/outputs are read/produced and an internal state is changed.
- An "continuous" (or integration) phase (C) where a solver computes an approximation of solutions and observes the zero-crossing of some of the signals of the system.
- From D to C, possibly resets the solver.
- From C to D, when a zero-crossing is detected.



Given a hybrid model, the compiler has to produce three functions, *step*, g and f, an initial state σ_0. y is called the continuous state.

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

- $next_\sigma$ gathers all discrete changes. Given a state σ, the current time t and continuous state y, it returns a new state σ' and a new continuous state y'.
- g_σ defines zero-crossing signals to be observed during integration.
- f_σ is the function to integrate and passed to an ODE solver.

f and g must be free of side effect!

Why? because the solver call them a variable (and unknown) number of times to compute an approximation of the solution of $\dot{y} = f_\sigma(t, y)$.

It is even better that f be continuous (C^0) which is a sufficient condition for the existence of a solution.

Can we ensure it by static typing?

A Hybrid Systems Language Kernel

A synchronous language core extended with three primitives (in red).

$$
\begin{array}{lcl}
d & ::= & \texttt{let}\, x = e \mid \texttt{let}\, k\, f(pi) = pi\, \texttt{where}\, E \mid d; d \\
k & ::= & \texttt{fun} \mid \texttt{node} \mid \texttt{hybrid} \\
e & ::= & x \mid v \mid op(e, ..., e) \mid v\ \texttt{fby}\ e \mid \texttt{last}\, x \mid f(e, ..., e) \mid \texttt{up}(e) \\
p & ::= & x \mid (x, ..., x) \\
pi & ::= & xi \mid xi, ..., xi \\
xi & ::= & x \mid x\ \texttt{last}\ e \mid x\ \texttt{default}\ e \\
E & ::= & p = e \mid \texttt{der}\ x = e \\
 & & \mid \texttt{if}\, e\, \texttt{then}\, E\, \texttt{else}\, E \\
 & & \mid \texttt{reset}\ E\ \texttt{every}\ e \\
 & & \mid \texttt{local}\ pi\ \texttt{in}\ E \mid \texttt{do}\ E\ \texttt{and}\ \ldots E\ \texttt{done}
\end{array}
$$

In order to generate sequential code, follow and adapt the compilation method used in a synchronous compiler [DGP08], that is:.

- Reduction into a small data-flow language kernel;
- optimizations; normalisation and scheduling;
- generation of code in an intermediate sequential language.
- generation of target code (e.g., OCaml for Zelus, C for Scade Hybrid)

An intermediate data-flow language

The intermediate language is extended with three new constructions.

$$
\begin{array}{lcl}
d & ::= & \texttt{let}\, x = c \mid \texttt{let}\, k\, f(p) = a\, \texttt{where}\, C \mid d; d \\
k & ::= & \texttt{fun} \mid \texttt{node} \mid \texttt{hybrid} \\
C & ::= & (x_i = a_i)_{x_i \in I} \text{ with } \forall i \neq j . x_i \neq x_j \\
a & ::= & e^{ck} \\
e & ::= & x \mid v \mid op(a, ..., a) \mid v\ \texttt{fby}\ a \mid \texttt{pre}(a) \\
 & & \mid f(a, ..., a) \\
 & & \mid \texttt{merge}(a, a, a) \mid a\ \texttt{when}\ a \\
 & & \mid \texttt{integr}(a, a) \mid \texttt{up}(a) \\
p & ::= & x \mid (x, ..., x) \\
ck & ::= & \texttt{base} \mid ck\, \texttt{on}\, a
\end{array}
$$

Put data-flow equations in normal form

Name the result of every memory operation or node instanciation. Separate them into three categories.

- *se*: strict expressions
- *de*: delay
- *ce*: expressions guarded by a condition (clock)

The equation $lx = \texttt{integr}(x', x)$ defines lx as a (continuous) state variable; possibly re-initialized by x during a discrete step.

$$
\begin{array}{lcl}
eq & ::= & x = ce^{ck} \mid x = f(sa, ..., sa)^{ck} \mid x = de^{ck} \\
sa & ::= & se^{ck} \\
ca & ::= & ce^{ck} \\
se & ::= & x \mid v \mid op(sa, ..., sa) \mid sa \ \texttt{when} \ sa \\
ce & ::= & se \mid \texttt{merge}(sa, ca, ca) \mid ca \ \texttt{when} \ sa \\
de & ::= & \texttt{pre}(ca) \mid v \ \texttt{fby} \ ca \mid \texttt{integr}(ca, ca) \mid \texttt{up}(ca)
\end{array}
$$

Well Scheduled Form

Equations are statically scheduled.

$Read(a)$: 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

$$Schedule(.) : I \to \{1 \dots |I|\}$$

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

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

From the data-dependence point-of-view, $\texttt{integr}(ca_1, ca_2)$ and $\texttt{up}(ca)$ 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

$$
\begin{array}{lll}
md & ::= & \mid \texttt{let}\, x = c \\
 & & \mid \texttt{let}\, f = \texttt{class}\langle M,\, I,\, (method_i(p_i) = e_i\ \texttt{where}\ S_i)_{i \in [1..n]} \rangle \\
M & ::= & [x : m[= v]; ...; x : m[= v]] \\
I & ::= & [o : f; ...; o : f] \\
m & ::= & \mathit{Discrete} \mid \mathit{Zero} \mid \mathit{Cont} \\
e & ::= & v \mid lv \mid op(e, ..., e) \mid o.method(e, ..., e) \\
S & ::= & () \mid lv \leftarrow e \mid S\ ;\ S \mid \texttt{var}\ x, ..., x\ \texttt{in}\ S \mid \texttt{if}\ c\ \texttt{then}\ S\ \texttt{else}\ S \\
R, L & ::= & S; ...; S \\
lv & ::= & x \mid lv.field \mid \texttt{state}\,(x)
\end{array}
$$

State Variables

Discrete State Variables (sort *Discrete*)

- Read with `state`(x);
- modified with `state`$(x) \leftarrow c$

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

The bouncing ball

```
let hybrid bouncing (y0) = y where
rec der y = y' init y0
and der y' = −. g init 0.0 reset up(−. y) → 0.8 ∗. last y'
```

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

Examples (both Zélus and SCADE) at: `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 <- ( *. ) 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 }
```

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.

Added to 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.

Conclusion

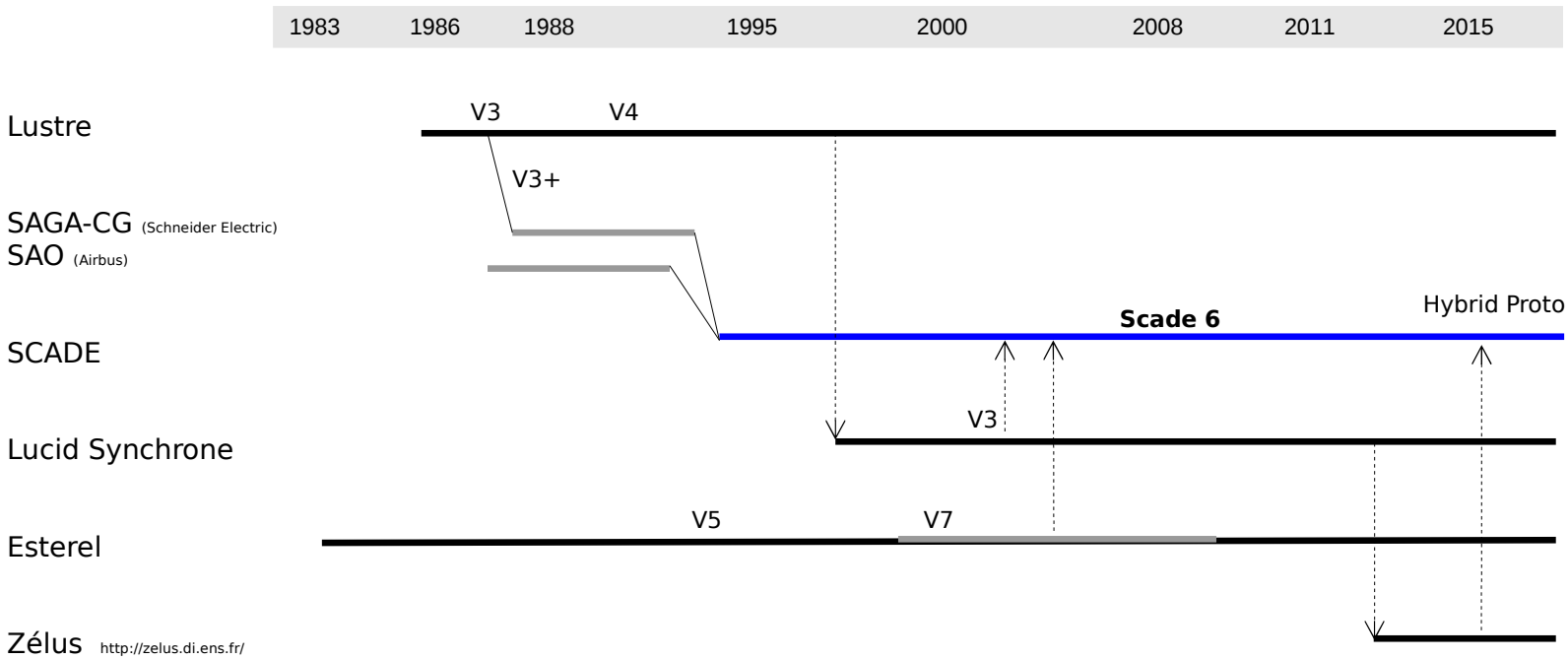
Two experiments

- The language **Zélus** and its compiler.
- An industrial prototype SCADE Hybrid based on the production compiler **KCG 6.7** of Scade.
- For KCG, **less than 5%** of LOC added to account for hybrid features.
- It is a conservative extension w.r.t the Scade language.
- This means that the very same code is used for simulation and the platform.
- A new tool developed by ANSYS, called DigitalTwins, reuses a part of Scade Hybrid.

Is the language expressive enough to define a standard library?

- In many tools (e.g., Simulink), many blocks (e.g., filters, integrators, etc.) are directly programmed in C.
- Instead, can we program them (e.g., directly in Zelus)? How the generated code compares (in efficiency) w.r.t, the hand-written code?

Timeline


1983
1986
1988
1995
2000
2008
2011
2015
Lustre
V3
V4
V3+
SAGA-CG (Schneider Electric)
SAO (Airbus)
SCADE
Scade 6
Hybrid Proto
Lucid Synchrone
V3
Esterel
V5
V7
Zélus http://zelus.di.ens.fr/

References I

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.

Timothy Bourke, Francois Carcenac, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet.

A Synchronous Look at the Simulink Standard Library.
In *ACM International Conference on Embedded Software (EMSOFT)*, Seoul, October 15-20 2017.

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.

References II

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.

Gwenael Delaval, Alain Girault, and Marc Pouzet.

A Type System for the Automatic Distribution of Higher-order Synchronous Dataflow Programs.
In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

T. Lindstrom.

An invitation to non standard analysis.
In N. Cutland, editor, *Nonstandard analysis and its applications*. Cambridge Univ. Press, 1988.