Hybrid Systems Modelers
(from a programming language perspective) [a]

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

Course notes - MPRI Nov. 2022

[a] See my talk at College de France (seminar of the course given by Gérard Berry).

Plan of the talk

1. What is hybrid systems programming?
2. Some issues with existing hybrid systems modelers.
3. Hybrid systems practicalities (the numeric solver).
4. A proposal for improvement.
5. The `Zélus` prototype.
6. Some solutions and open questions.

Programming languages for hybrid systems

Mix discrete and continuous-time behaviours in a single program source. Use it to simulate, test, and generate embedded code.

Avoid some implementation choices of existing hybrid modelers.

- **Discrete Time** is when the solver decides to **stop**.
- Avoid VHDL-like **run-to-completion** to decide weither a fix-point has been reached or not.

Objective

- A synch. language that mix **data-flow equations, automata and ODEs**.
- Be **conservative** w.r.t the synchronous subset (same semantics, compiler).

Divide & Recycle

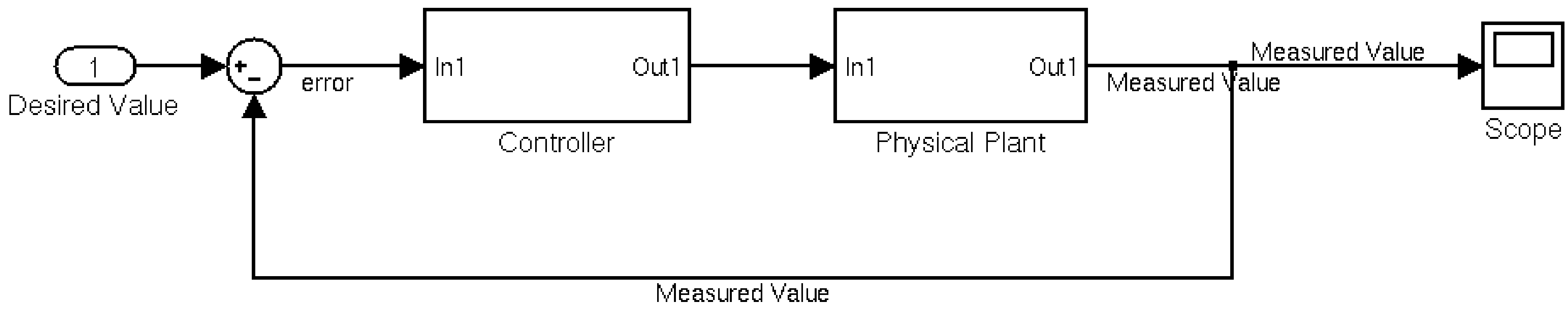
- Typing and analysis to divide discrete-time from continuous-time signals.
- Recycle an existing synchronous compiler; run with an off-the-shelf solver.

The current practice

Embedded software interacts with physical devices for which continuous-time models are effective. A model (program) has to mix:

- a discrete/continuous controller and a discrete/continuous environment;
- a continuous environment with several modes (clutch-control, etc.).

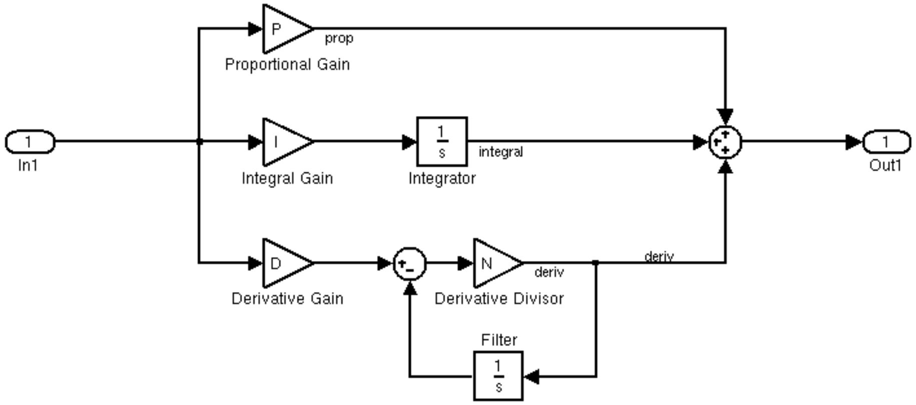
The classical use A controller in closed loop with its environment.

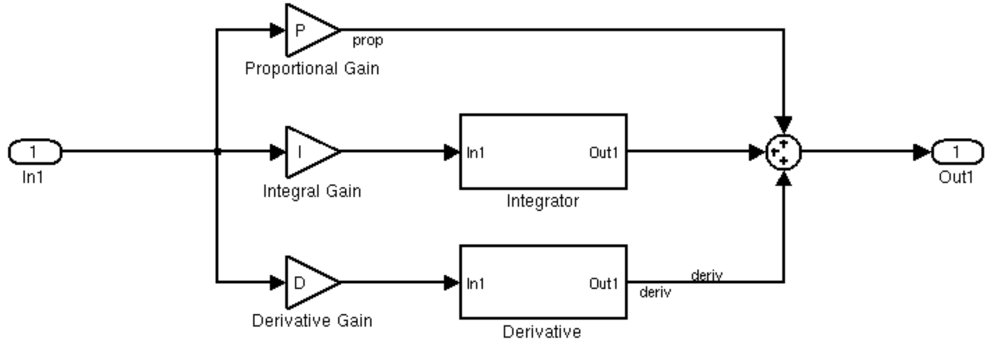


E.g.,: A continuous or discrete PID controller; A continuous or discrete model of the Physical environment.

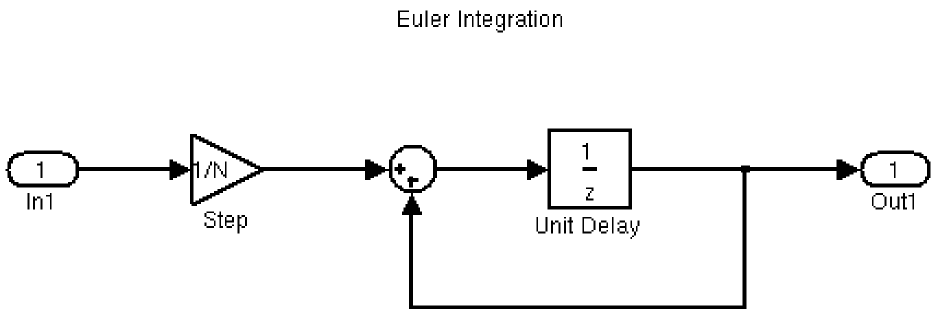
Controllers are not necessarilly discrete (for CS people)

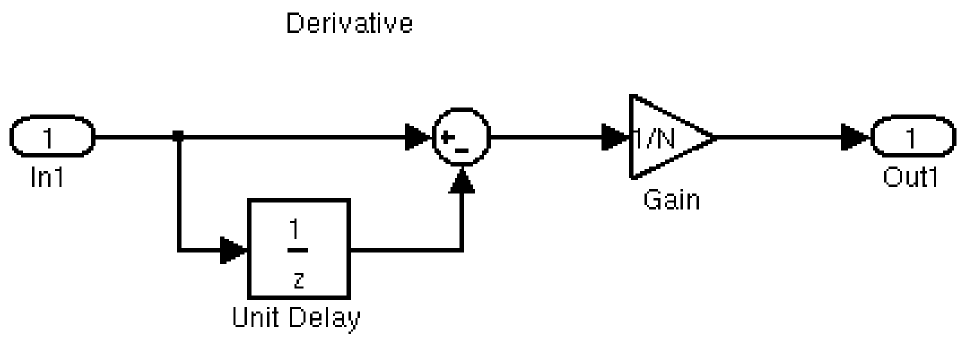
E.g., A PID controller can be either continuous or discrete. A first model made with a continuous controller, than sampled, than turned into discrete.





Simulate with a variable-step solver (for efficiency); then fixed-step (for timing); then turn integral and derivative into difference equations (for embedded code).





Current Tools

A lot of industrial tools exist and are widely used:

- Simulink/Stateflow ($\geq 10^6$ licences), LabVIEW: ODEs+discrete;
- Modelica, VHDL-AMS, VERILOG-AMS: DAEs+discrete;
- Dedicated tools to one continuous physics (e.g., mechanics, electro-magnetics, fluid). Some do multi-physics (ANSYS). Possibly paired with other tools.
- Dedicated tools for discrete-time only: SCADE.

Mathematical objects are:

- Difference equations; hierarchical automata.
- Differential equations (ODEs, DAEs, semi-explicit).

From a programming language perspective:

- **Deterministic parallelism** at every level (block diagrams).
- **Time** is **logical** and **global** (computation times are neglected).

A numeric solver to find a compromise between precision and efficiency.

Yet...

There is a gap between the mathematical models and their implementation. No comprehensive semantics of hybrid modelers exist at the moment.

- A semantics exists for discrete subsets only (e.g., [Caspi et al., Hamon and Rushby, Hamon] for Simulink/Stateflow).
- Mosterman et al., Lee et al., made important clarifications on the behaviour of hybrid modelers continuous/discrete interactions.

Some problems are unavoidable and due to numerical approximation and the use of floatting-point numbers. What we focus on is the **interaction** between continuous and discrete, not the way continuous trajectories are approximated.

One problem is that time is not logical:

Time is that of the simulation engine which exposes its internal steps. This may cause strange behaviours.

Let us see what it mean to have two systems in parallel.

Parallel composition: homogeneous case

Two equations with discrete time:

```
f = 0.0 → pre f + s and s = 0.2 * (x - pre f)
```

and the initial value problem (IVP):

```
der(y') = -9.81 init 0.0 and der(y) = y' init 10.0
```

The first program can be written in any synchronous language, e.g. `Lustre`.

$$\forall n \in \mathbb{N}^*, f_n = f_{n-1} + s_n \text{ and } f_0 = 0 \qquad \forall n \in \mathbb{N}, s_n = 0.2 * (x_n - f_{n-1})$$

The second program can be written in any hybrid modeler, e.g. `Simulink`.

$$\forall t \in \mathbb{R}_+, y'(t) = 0.0 + \int_0^t -9.81\, dt = -9.81\, t$$

$$\forall t \in \mathbb{R}_+, y(t) = 10.0 + \int_0^t y'(t)\, dt = 10.0 - 9.81 \int_0^t t\, dt$$

Parallel composition is clear since equations **share the same time scale**. Given the IVP, the numeric solver computes a sequences of approximated values for y and y' at instants $t \in I$, with $I \subseteq \mathbb{R}$ and order isomorphic to $\mathbb{N}$.

Parallel composition: heterogeneous case

Two equations: a signal defined at discrete instants, the other continuously.

```
der(time) = 1.0 init 0.0 and x = 0.0 fby x + time
```

or:

```
x = 0.0 fby x +. 1.0 and der(y) = x init 0.0
```

One might consider that the first means: $\forall n \in \mathbb{N}, x_n = x_{n-1} + \texttt{time}(n)$

And the second:

$$\forall n \in \mathbb{N}^*, x_n = x_{n-1} + 1.0 \text{ and } x_0 = 1.0$$

$$\forall t \in \mathbb{R}_+, y(t) = 0.0 + \int_0^t x(t)\, dt$$

i.e., $x(t)$ as a piecewise constant function from $\mathbb{R}_+$ to $\mathbb{R}_+$ with $\forall t \in \mathbb{R}_+, x(t) = x_{\lfloor t \rfloor}$.

In both cases, this would be a mistake. `x` is defined on a discrete, logical time. The index of the sequence $(x)_{i \in \mathbb{N}}$ has no relation with absolute time $\mathbb{R}$.

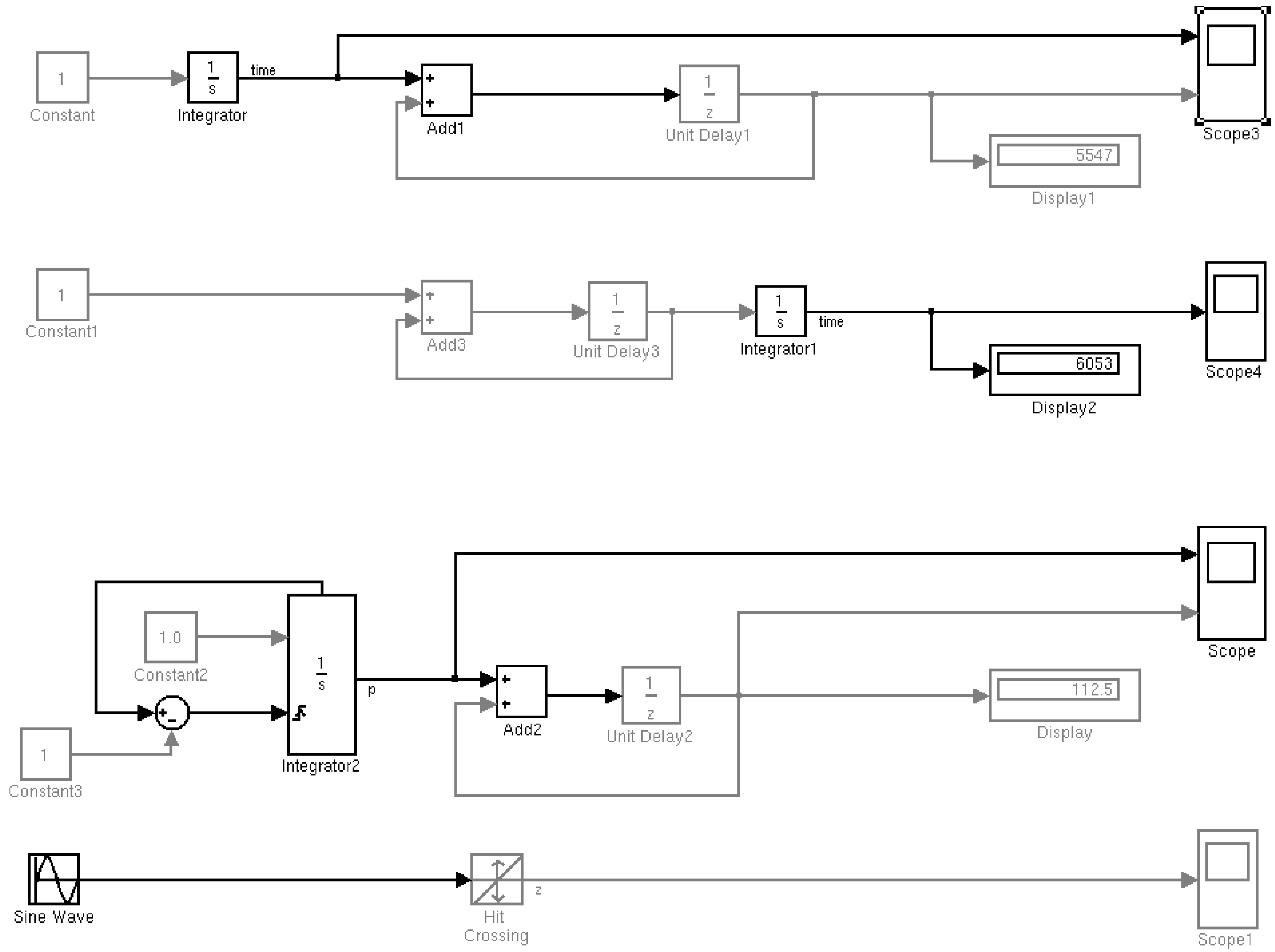
Equations with reset

Two independent groups of equations. A sawtooth signal `p`:

```
  der(p) = 1.0 init 0.0 reset up(p - 1.0) → 0.0
and
  x = 0.0 fby x + p
and
  der(time) = 1.0 init 0.0
and
  z = up(sin (freq * time))
```

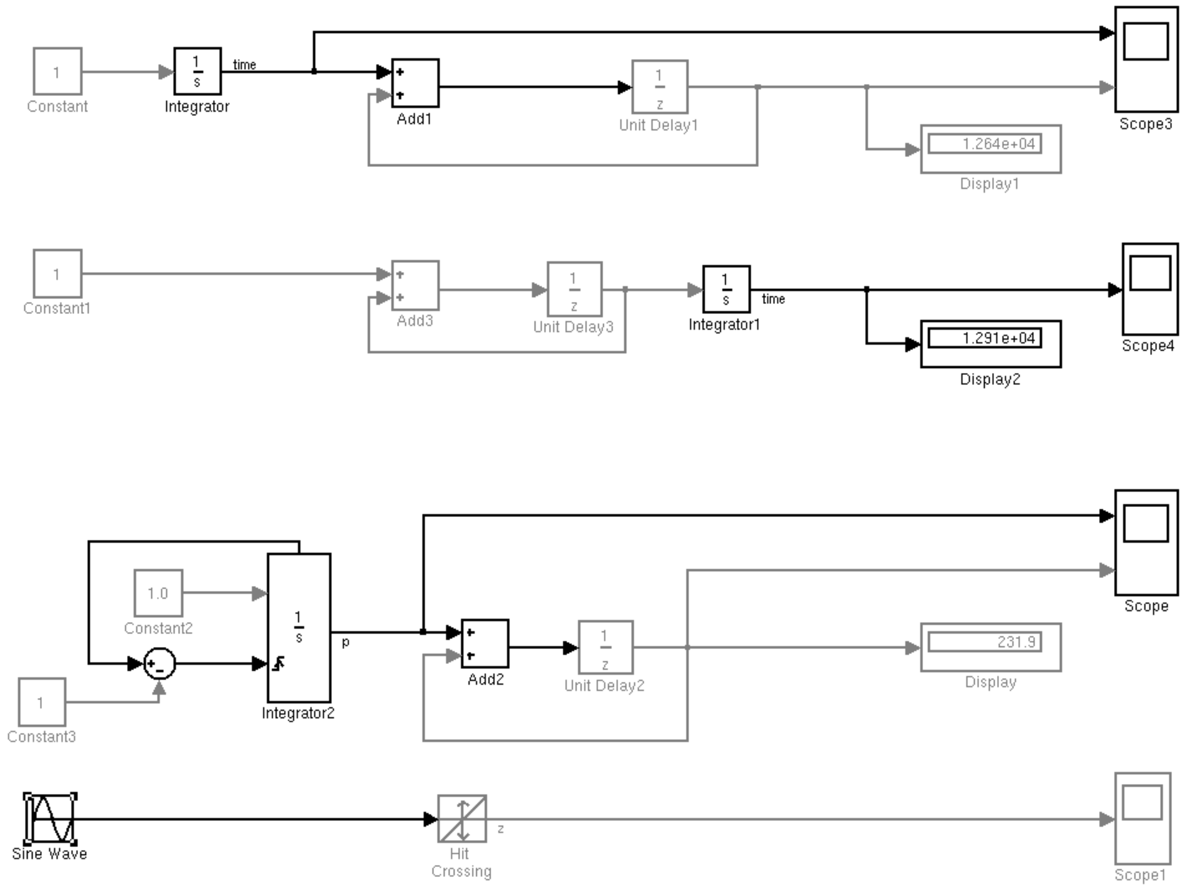
Properly translated in Simulink, changing `freq` changes the output of `x`!

What does Simulink on that example?



Select solver `ode45` (the default one); Amp = 1; Freq = 1 (sinusoid)

What does Simulink on that example?

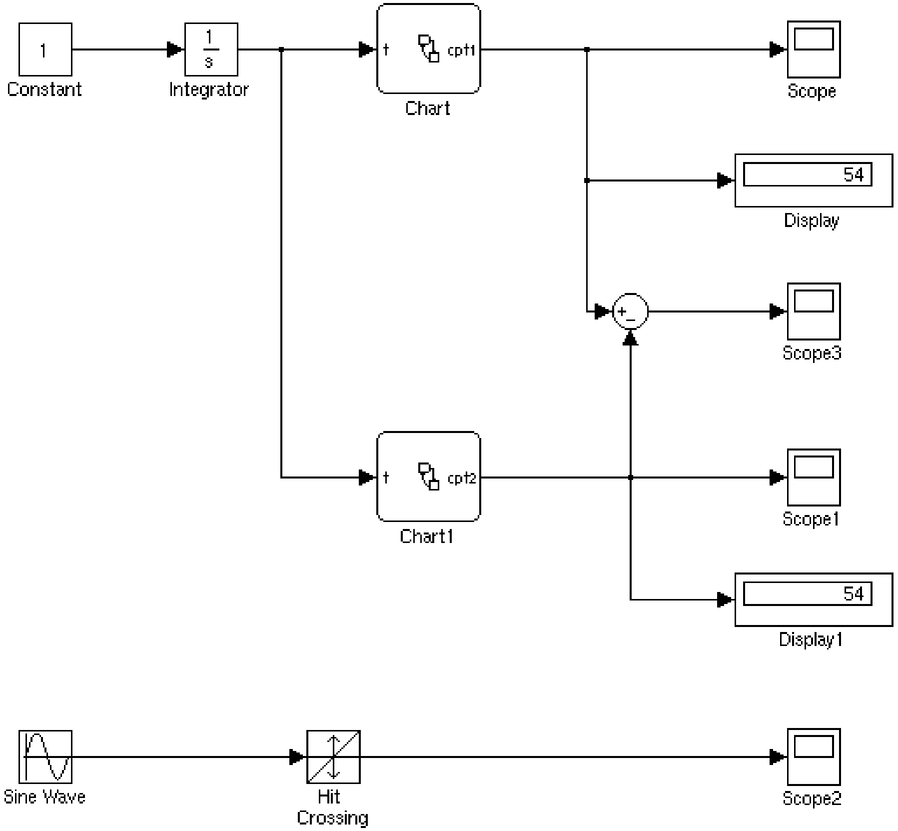


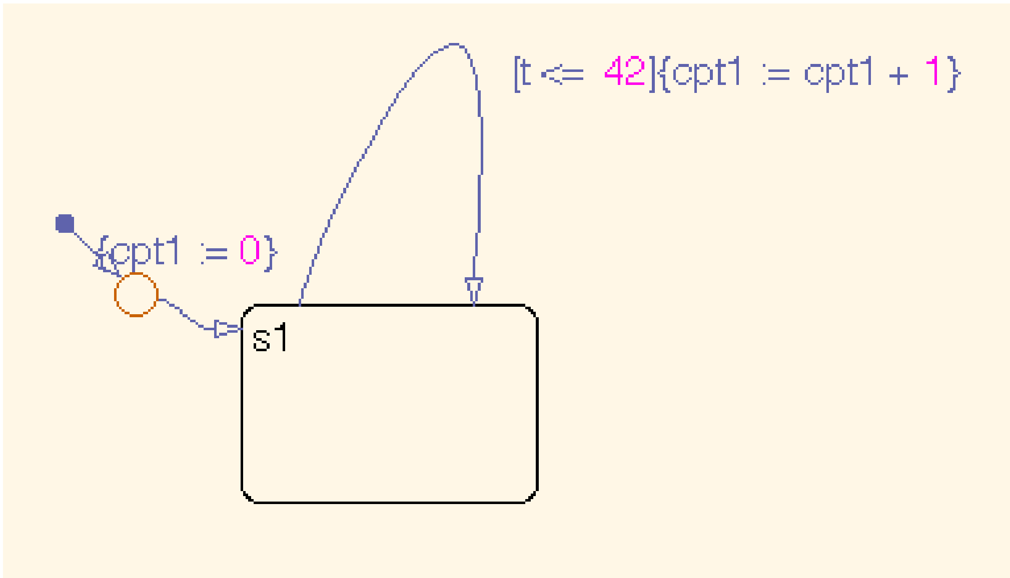
Select solver `ode45` (the default one); Amp = 1; Freq = 5 (sinusoid)

Yet, Simulink complains about this model: it finds an unproper mix of discrete/continuous-time.

Strange behaviours of Hybrid modelers

E.g., Simulink/Stateflow. What is a discrete step? How long is it?

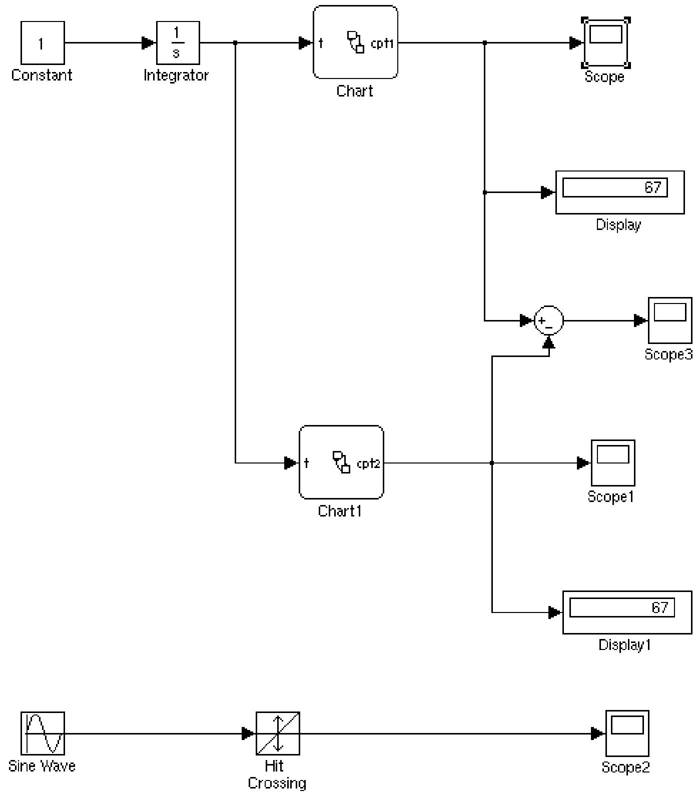


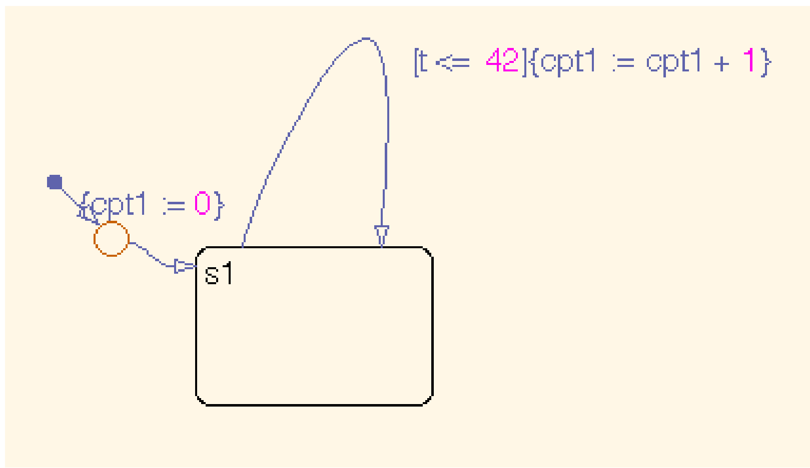


Select solver `ode23s` (stiff/Mod. Rosenbrock); Amp = 1; Freq = 1 (sinusoid).

Strange behaviours of Hybrid modelers

E.g., Simulink/Stateflow.

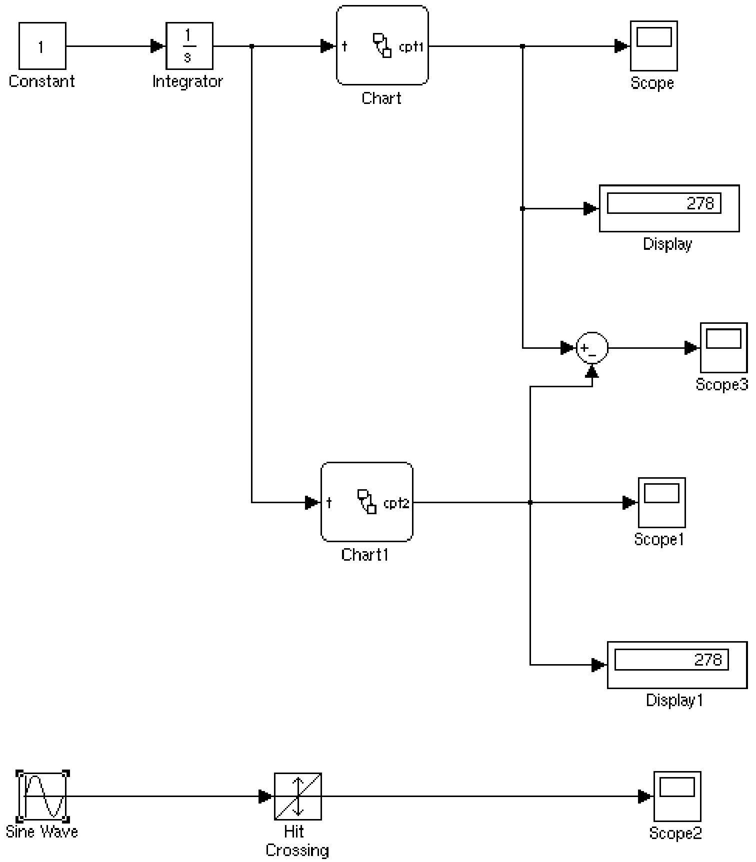


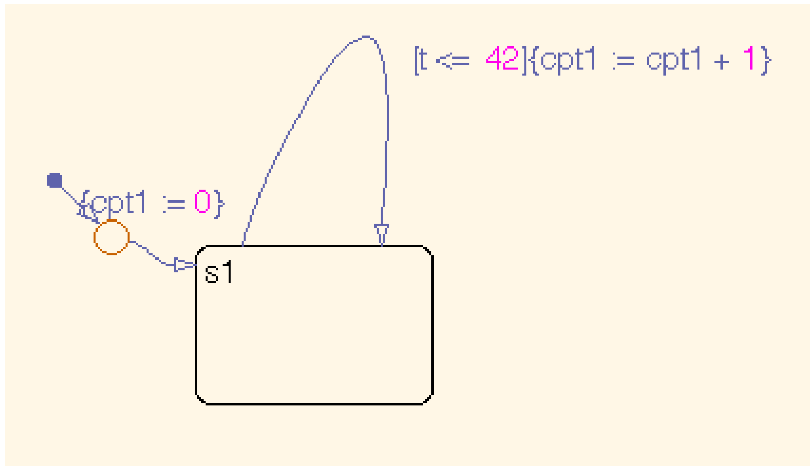


Select solver `ode45` (Dormand-Prince): we get Amp = 1; Freq = 1 (sinusoid).

Strange behaviours of Hybrid modelers

E.g., Simulink/Stateflow.





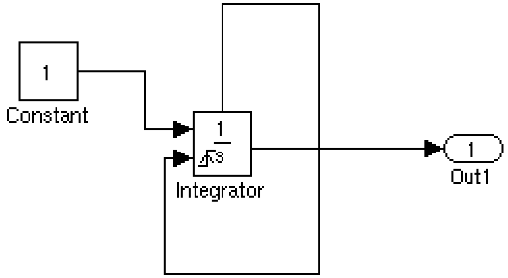
Select solver `ode45` (Dormand-Prince): we get Amp = 1; Freq = 5 (sinusoid).

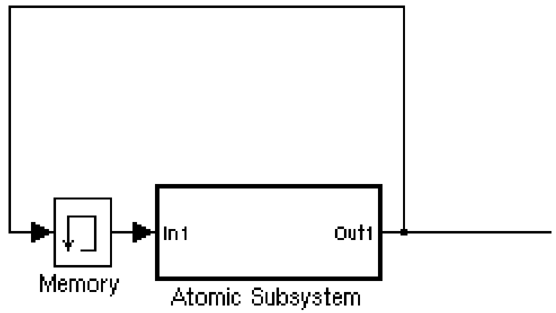
Changing the frequency/solver changes the semantics. Only one transition is taken during a step and it takes some amount of time.

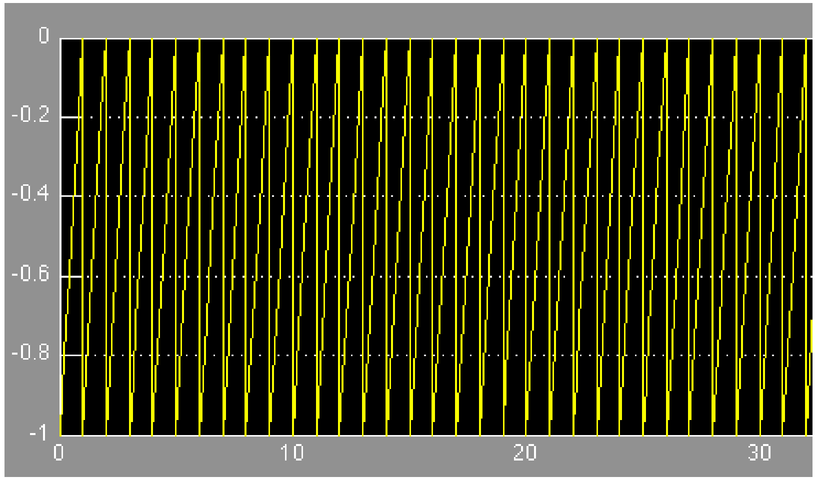
Strange behaviours of Hybrid modelers

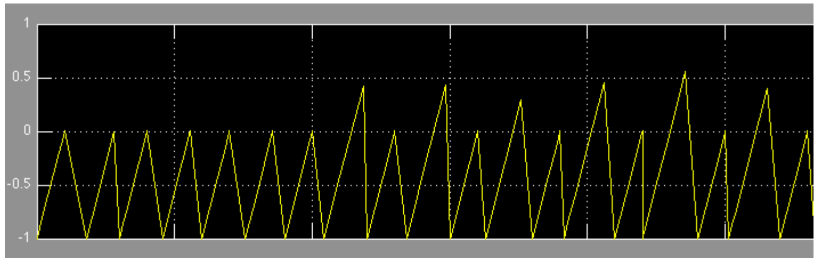
Adding delay/memory blocks or shared variables is sometimes mandatory to break algebraic loops.

E.g., use the state port when resetting an integrator.





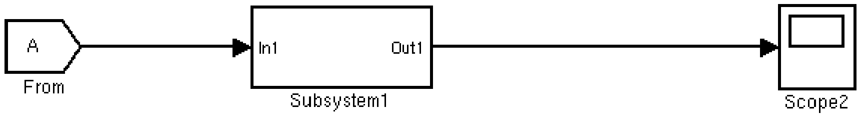


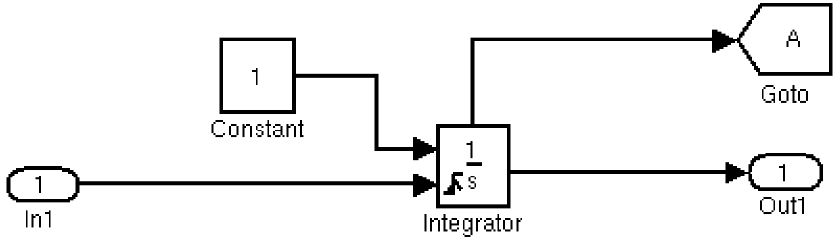


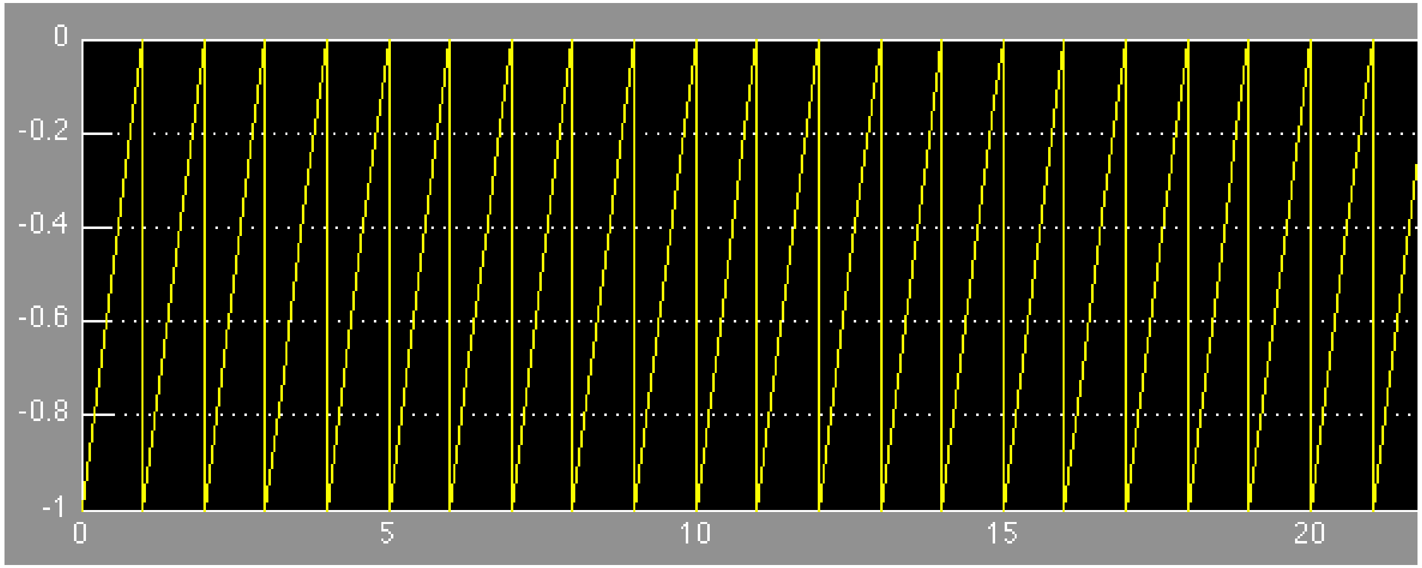
The stateport cannot be outputed. Yet, it is possible to store it into a memory block. The signal is shifted in a variable manner.

Strange behaviours of Hybrid modelers

Use a goto/from port or a write/read block. The main system reads from A. The subsystems outputs the value of the stateport into A.

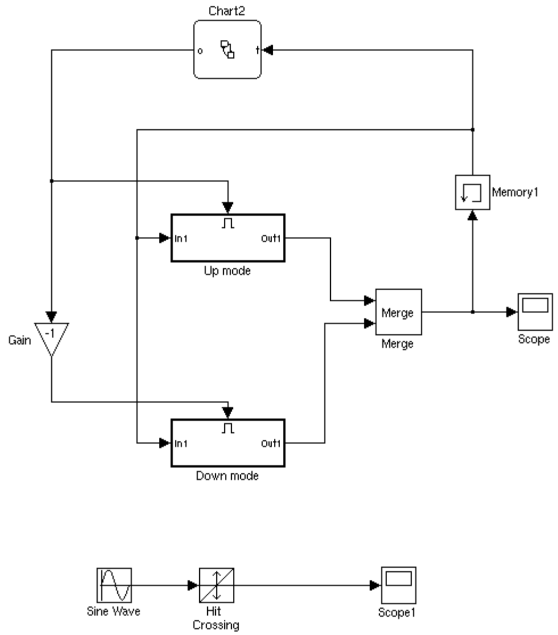


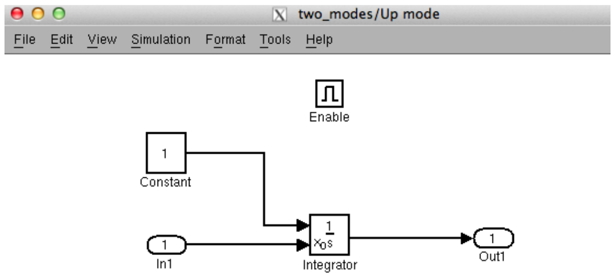


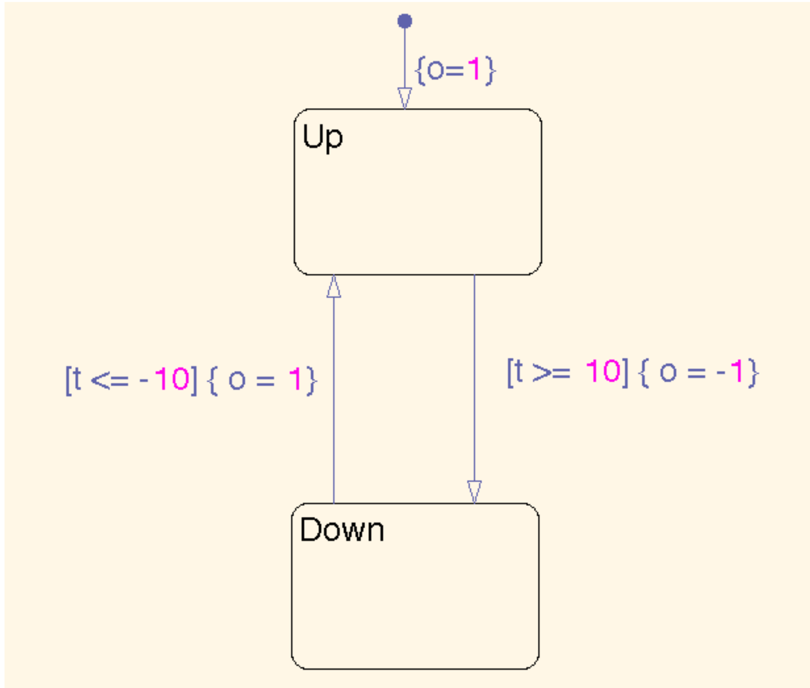


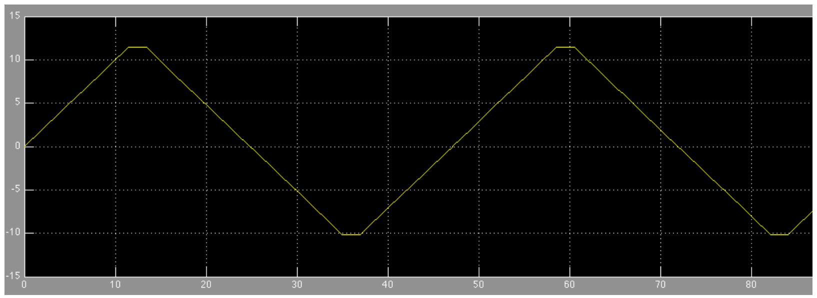
Strange behaviours of Hybrid modelers

Two teams, each of them in charge of a block (`Up` and `Down`). Merge them. A memory block is necessary to break the algebraic loop.



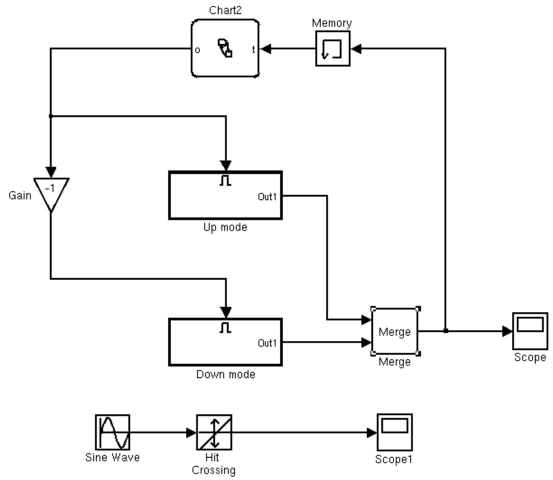


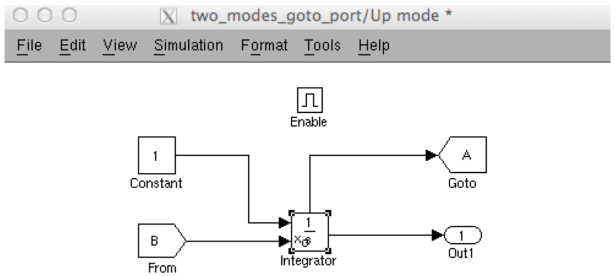


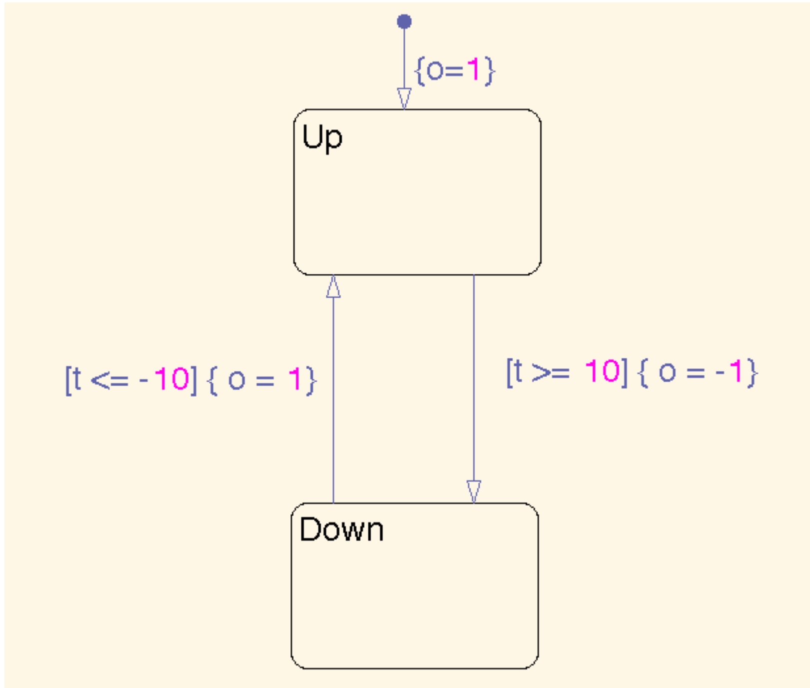


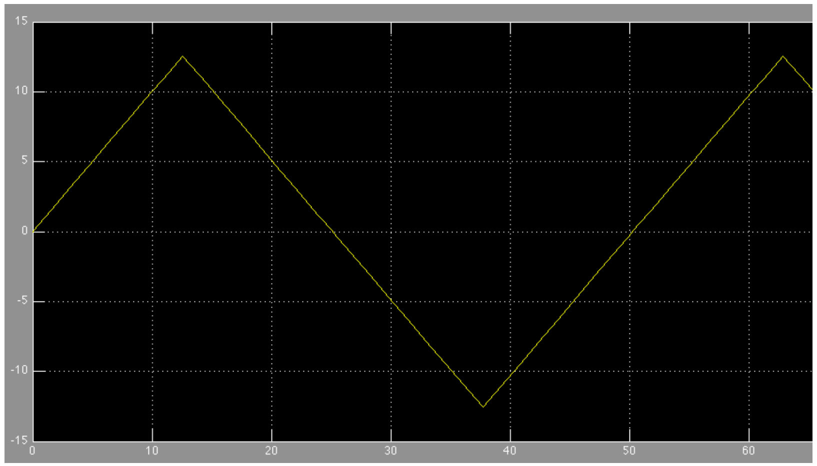
Strange behaviours of Hybrid modelers

Use the from/goto port to pass states between enabled subsystems. This work for two subsystems only. Yet, a memory block is necessary.



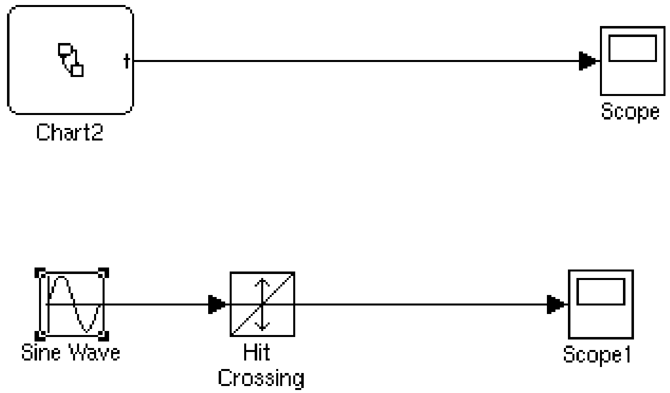


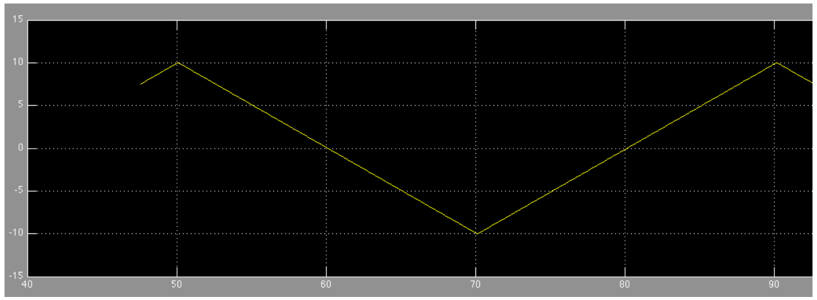


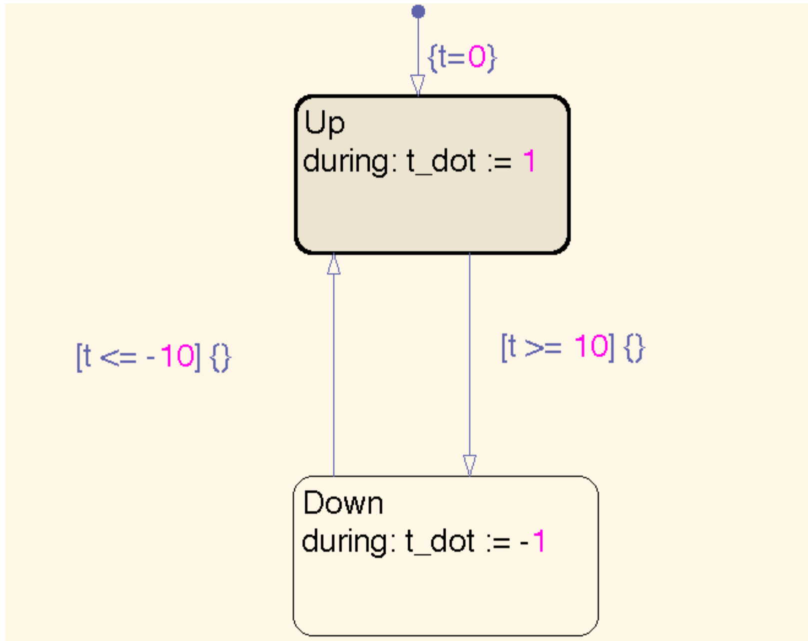


Strange behaviours of Hybrid modelers

Alternatively, directly write the hybrid automaton.







But we have abandonned modular design (two different teams developing different parts of a model).

- The only solution is to use read/write to global shared variables for every mode. They must be used carefully because of possible critical races.
- Concurrent writes are treated according to lexical order between block names.

What went wrong?

The partition between discrete time and continuous time is not strong enough. Discrete time is the one of the global simulation. Those models should be statically detected to be wrong and rejected

We propose the following discipline [LCTES'12, EMSOFT'12]:

A signal is *discrete* if it is activated on a *discrete clock*, that is so defined:

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

Impose a type discipline for that such that signals are proved to be left continuous during integration with all discontinuities announced to the solver. This is not enough. Algebraic loops must be broken.

> Provide a construct `last` x that returns the previous value of x. In Non-standard semantics [CDC'10,JCSS'12], it coincides with the left-limit of x when x is left-continuous; the previous one otherwise.

Ensure the two disciplines above with static typing and a causality analysis. Strange Simulink examples would all be statically rejected.

The Practicalities of Hybrid Simulation

The interface of a numeric solver (e.g. SUNDIALS CVODE) is:

- a function $f_y(t, x)$ that maps simulation time and state vector x to a vector of intantaneous derivatives; it can be parameterized by y.
- a function $g_y(t, x)$ that returns a vector z of zero-crossing expressions values.

Given horizon h, approximates $x(t_i + h)$ from t_i to $t_i + h$, observing the current time t_i, while monitoring the elements of z for change of sign.

$$C \text{ phase:} \qquad \dot{x} = f_y(t, x) \quad z = g_y(t, x)$$

Between continuous phases, discrete changes are allowed:

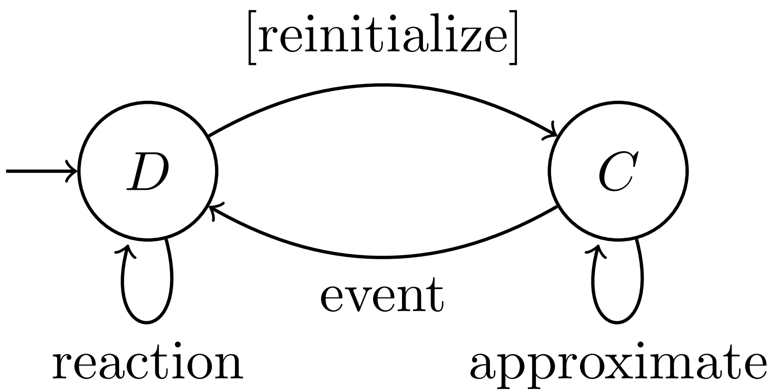
$$D \text{ phase:} \qquad y', \mathit{goagain}', x' = \mathit{next}(y, x) \text{ if } \mathtt{up}(z_1) \vee \cdots \vee \mathtt{up}(z_n) \vee \mathit{goagain}$$

It defines a new value for y, *goagain* and x. Loop while *goagain* is true.

Initialisation:

$$x = x_0$$

The simulation cycles between discrete and continuous phases.



The Practicalities of Hybrid Simulation

In Modelica, these two functions are represented by a single one (called a FMU[a].)

$$\dot{x}, x', y', go', z' = fmu(y, t, x, z)$$

Finally, f_y, g_y and *next* can be programmed by hand (e.g., in C). For the simulation to be faithful, it is important that:

- f_y and g_y are free of side effects and continuous during integration.
- all discrete changes are done outside, typically at a zero-crossing instant or the end of an horizon.

Programmed by hand, **these invariants are difficult to ensure**.

Moreover, the semantics of the simulation engine can be defined formally [BCP+15].

Objective: Define a high-level language that compiles to $f, g, next(\cdot)$ (or $fmu(\cdot)$) with the above invariants ensured by construction.

[a]See Functional Mock-up Interface `https://www.fmi-standard.org`.

The Practicalities of Synchronous Programming

In a synchronous language (e.g., SCADE), one write difference equations and hierarchical automata. Programs are compiled into pairs of:

A state $s : S$; a transition function *step*: $S \times I \rightarrow O \times S$

Then, the execution is cyclic, either periodically sampled or event-triggered.

Example (in Zélus syntax):

```
let node filter(x) = f where
  rec f = 0.0 → pre f +. s and s = 0.2 *. (x -. pre f)
```

is translated into the straight-line imperative code:

```
type filter = { mutable pre16 : float; mutable init17 : bool }
let filter_alloc () = { pre16 = 0.0; init17 = true }
let filter_step self x12 =
  let pf13 = self.pre16 in
  let s14 = ( *. ) (0.2) ((-.) (x12) (pf13)) in
  let f15 = if self.init17 then 0. else (+.) (pf13) (s14) in
  self.init17 <- false; self.pre16 <- f15; f15
```

Zélus

A prototype of a new `Scade`-like language with ODEs

Before showing a bit of theory, let's see a few examples with:

- Difference equations; hierarchical automata; ODEs.

Main features:

- A type system that reject some bad behaviour are the one showed previously.
- An initialization analysis to check that state variables (discrete or continuous) are properly initialised.
- A causality analysis to ensure that fix-points can be computed sequentially.
- A translation into synchronous code which is in turn compiled into sequential code by an existing compiler.
- The continuous part is approximated by an off-the-shelf solver (here SUNDIALS CVODE).

Combinatorial and sequential functions

Time is logical as in `Lustre`. A signal is a sequence of values and nothing is said about the actual time to go from one instant to the other.

```
let add (x,y) = x + y


let min_max (x, y) = if x < y then x, y else y, x


let node after (n, t) = (c = n) where
  rec c = 0 → pre(min(tick, n))
  and tick = if t then c + 1 else c
```

When feed into the compiler, we get:

```
val add : int × int → int
val mix_max : α × α → α × α
val after : int × int ⇒ bool
```

Here `x`, `y`, etc. are sequences.

The counter can be instantiated in a two state automaton,

```
let node blink (n, m, t) = x where
 automaton
 | On → do x = true until (after(n, t)) then Off
 | Off → do x = false until (after(m, t)) then On
```

which returns a value for `x` that alternates between `true` for `n` occurrences of `t` and `false` for `m` occurrences of `t`.

```
let node blink_reset (r, n, m, t) = x where
 reset
   automaton
   | On → do x = true until (after(n, t)) then Off
   | Off → do x = false until (after(m, t)) then On
 every r
```

The type signatures inferred by the compiler are:

```
  val blink : int × int × int ⇒ bool
  val blink_reset : int × int × int × int ⇒ bool
```

Examples

Up to syntactic details, these programs could have been written *as is* in SCADE 6 or `Lucid Synchrone`. Now, a simple heat controller with ODEs. [a]

```
(* an hysteresis controller for a heater *)
let hybrid heater(active) = temp where
  rec der temp = if active then c -. temp else -. temp init temp0

let 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

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

[a] This simple example is the hybrid version of the one of Nicolas Halbwachs, used to present Lustre, during his seminar at Collège de France, in January 2010.

The Bouncing ball

```
let hybrid bouncing(x0,y0,x'0,y'0) = (x,y) where
    der(x) = x' init x0
 and
    der(x') = 0.0 init x'0
 and
    der(y) = y' init y0
 and
    der(y') = -. g init y'0 reset up(-. y) → -. 0.9 *. last y'
```

Its type signature is: $\texttt{float} \times \texttt{float} \times \texttt{float} \xrightarrow{\texttt{C}} \texttt{float} \times \texttt{float}$

When `-. y` crosses zero, re-initilize the speed `y'` with `-. 0.9 * last y'`.

`last y'` stands for the previous value of `y'`. As `y'` is immediately reseted, writting `last y'` is mandatory; otherwise, `y'` would instantaneously depend on itself.

ODEs and Zero-crossings

E.g., the sawtooth signal, the two-state automaton.

```
let hybrid sawtooth() = t where
  rec der t = 1.0 init -1.0 reset up(last t -. 1.0) → -1.0

let hybrid fm() = t where
  rec init t = 0.0
  and automaton
      | Up → do der t = 1.0 until up(t -. 10.0) then Down
      | Down → do der t = -1.0 until up(-10 -. t) then Up

let hybrid fm'() = t where
  rec init t = 0.0
  and automaton
      | Up → do der t = 1.0
                until up(t -. 10.0) then do t = last t -. 1.0 in Down
      | Down → do der t = -1.0 until up(-10.0 -. t) then Up
```

Other examples

Bang-bang controller, bouncing balls, stickysprings, backhoe, etc.

Synchronous zero-crossings

- `up`(e) tests the zero-crossing of expression e (from negative to positive).
- If $x =$ `up`(e), all handlers using x are governed by the same zero-crossing.
- Handlers have priorities.

```
z = present up(x) → 1 | up(y) → 2 init 0
```

- `last` x is the "previous" value of x. It coincides with the left-limit of x.
  - During integration, `last` $x = x$.
  - During a discrete step, `last` x is the previous value of x.

```
z = present up(x) → last z + 1 | up(y) → last z - 1 init 0
```

Priorities

What if two zero-crossings happen at the same time?

```
rec der x = 1.0 init -1.0
and z1 = up(x) and z2 = up(x+0)
and z = present z1 → 1 | z2 → 2 init 0
and z' = present z2 → 1 | z1 → 2 init 0
and ok = (z = z')
```

The Illinois method (e.g., that of SUNDIALS CVODE) for zero-crossing detections finds that both `z1` and `z2` are true.

```
rec der one_every_second = 1.0 init -1.0 reset z1 → -1.0
and der one_every_ten_second = 1.0 init -1.0 reset z2 → -10.0
and z1 = up(one_every_second)
and z2 = up(one_every_ten_second)
```

At time $t = 10.0$, the Illinois method does not detect that both `z1` and `z2` are true. One is slightly before the other.

What should we do with such programs?

- Writting `up`(x) twice (or the first example) is certainly not a good idea. If one wants to synchronize several parts on the same zero-crossing, distribute `z1`.
- Writting a timer this way (in the second example) is neither a good idea.
- The two programs have a potential critical race. Should we warm the user or raise an error a run-time?
- Should we statically (or dynamically) reject a program which combines two signals modified by two independent zero-crossing?
- Would-it be meaningful to consider `z1 & z2` (conjunction), and `z1 | z2` (union) as an event?

  ```
  z = present z1 & z2 → 1 | z1 | z2 → 2 init 0
  ```

  as a short-cut for:

  ```
  z = present z1 & z2 → 1 | z1 → 2 | z2 → 2 init 0
  ```

Synchronous events

```
rec z = present z1 → 1 | z2 → 2 init 0
and z' = present z2 → 2 | z1 → 1 init 0
```

Several options are possible:

- Forbid them: if two events happen at the same time or are too close to each others, stop the simulation.
- Treat them sequentially, one after the other: do one discrete step with `z1 = true`, `t2 = false`, and the next step with `z1 = false`, `z2 = true` (or conversely). This introduces non determinacy.
- Allow it. E.g., if `z1` and `z2` can be true at the same time.
- Set a flag or parameter and specify the composition of events that must not appear during the simulation (e.g., write `assert z1#z2` to indicate that they must be exclusive). The simulation is stopped at run-time, in that case.

Currently, the choice made in `Zélus` is that the disjunction and unions of two events is an event.

Is `z1 & z2` an event?

By considering that `z1 & z2` is a possible event, it is possible to take decisions. E.g.,:

```
let compute_when_not_synchronous(x, default, z1, z2) = o where
  rec o = present z1 & z2 → default else f(x)


let check_non_synchronous(z1, z2) =
  present z1 & z2 → true else false
```

Open questions

- Is-it reasonnable? Shouldn't we stop the simulation when two zero-crossings happen at the same time?
- During a sequence of discrete transitions, if `z1` is false and `z2` is true, is-it then possible to have `z1` true?
- Should we statically analyse pattern matching on events so that if a branch has been taken during a discrete transition because `z1` was false and `z2` is true, it is then not possible to have `z1` true while time does not progress (monotony)?
- Should we raise a run-time error in that case?
- Would it be possible to statically ensure properties like `z1 # z2` (exclusive)?

Somes hints about the semantics, typing and compilation

A language kernel

First-order (two distinct name spaces). Data-flow equations.

$$
\begin{array}{rcl}
d & ::= & \texttt{let}\ k\ f(p) = e\ \texttt{where}\ E \mid d; d \\
e & ::= & x \mid v \mid op(e) \mid e\ \texttt{fby}\ e \mid \texttt{last}\ x \mid f(e) \mid (e, e) \\
p & ::= & (p, p) \mid x \\
E & ::= & x\ =\ e \mid E\ \texttt{and}\ E \mid \texttt{local}\ x\ \texttt{in}\ E \\
 & & \mid \texttt{if}\ e\ \texttt{then}\ E\ \texttt{else}\ E \\
 & & \mid \texttt{present}\ h\ \texttt{init}\ e \\
 & & \mid \texttt{der}\ x\ =\ e\ \texttt{init}\ e\ \texttt{reset}\ h \\
h & ::= & e \rightarrow e \parallel \ldots \parallel e \rightarrow e \\
k & ::= & \texttt{node} \mid \texttt{hybrid} \mid \epsilon
\end{array}
$$

A sketch of the semantics [CDC'10, JCSS'12]

The sets ${}^\star\mathbb{R}$ and ${}^\star\mathbb{N}$ as the non-standard extensions of $\mathbb{R}$ and $\mathbb{N}$.

- ${}^\star\mathbb{N}$ contains elements that are infinitely large (${}^\star n > n$ for any $n \in \mathbb{N}$).
- ${}^\star\mathbb{R}$ contains elements that are *infinitesimal*, $0 < \partial < t$ for any $t \in \mathbb{R}_+$.

The base clock: ∂ infinitesimal, the set

$$BaseClock(\partial) = \{n\partial \mid n \in {}^\star\mathbb{N}\}$$

is isomorphic to ${}^\star\mathbb{N}$ as a total order. For every $t \in \mathbb{R}_+$ and any $\epsilon > 0$, there exists $t' \in BaseClock(\partial)$ such that $|t' - t| < \epsilon$ expressing that $BaseClock(\partial)$ is dense in $\mathbb{R}_+$.

$BaseClock(\partial)$ is a natural candidate for a time index set and ∂ is the corresponding time basis.

For $t = t_n = n\partial \in BaseClock(\partial)$, ${}^\bullet t = t_{n-1}$ and $t^\bullet = t_{n+1}$.

A sketch of the semantics

Reason "as if" the time was discrete and global. We borrowed the idea of using non standard analysis for the semantics of hybrid systems from Bliudze et Krob.

Its use for a hybrid synchronous language is novel.

Clock and signals A *clock* T is a subset of $BaseClock(\partial)$. A *signal* s is a total function $s : T \mapsto V$.

If T is a clock and b a signal $b : T \mapsto \mathbb{B}$, then T `on` b defines a subset of T comprising those instants where $b(t)$ is true:

$$T \texttt{ on } b = \{t \mid (t \in T) \wedge (b(t) = \texttt{true})\}$$

If $s : T \mapsto {}^\star\mathbb{R}$, we write T `on up`(s) for the instants when s crosses zero, that is:

$$T \texttt{ on up}(s) = \{t \mid (t \in T) \wedge (s({}^\bullet t) \leq 0) \wedge (s(t) > 0)\}$$

The effect of `up`(e) could also be delayed by one cycle.

$$T \texttt{ on up}(s) = \{t^\bullet \mid (t \in T) \wedge (s({}^\bullet t) \leq 0) \wedge (s(t) > 0)\}$$

Ideal semantics of ODEs with resets

Write $x(n)$, with $n \in \mathbb{N}$, for the value of x at time $n\partial$.

The semantics of $\texttt{der}\, x = e \;\texttt{init}\; e_0 \;\texttt{reset}\; z_1 \rightarrow e_1 \parallel ...z_k \rightarrow e_k$ is:

$$
\begin{array}{rcl}
x(0) & = & e_0(0) \\
x(n) & = & \texttt{if}\, z_1(n)\, \texttt{then}\, e_1(n)\, \texttt{else}\, ...\texttt{if}\, z_k(n)\, \texttt{then}\, e_k(n)\, \texttt{else}\, x(n-1) + \partial.e(n-1)
\end{array}
$$

The semantics of a zero-crossing $\texttt{up}(e)$ and delay are:

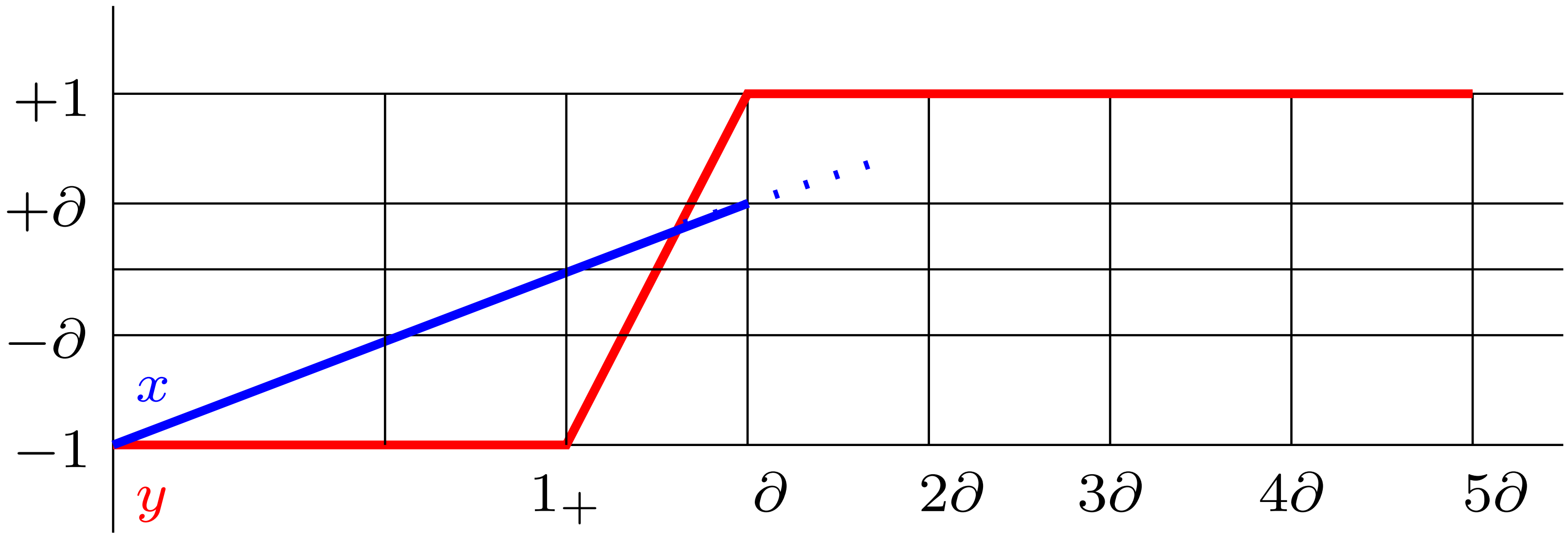
$$
\begin{array}{rcl}
\texttt{up}(e)(0) & = & \texttt{false} \\
\texttt{up}(e)(n) & = & (e(n) > 0)\&(e(n) \leq 0) \\
(\texttt{last}\, x)(n) & = & x(n-1) \\
e_0\, \texttt{fby}\, e(0) & = & e_0(0) \\
e_0\, \texttt{fby}\, e(n) & = & e(n-1)
\end{array}
$$

Note that, from the causality point-of-view, an integrator acts as a delay: it breaks cycles. E.g., the following program is causal:

```
der x = 1.0 -. x init 10.0
```

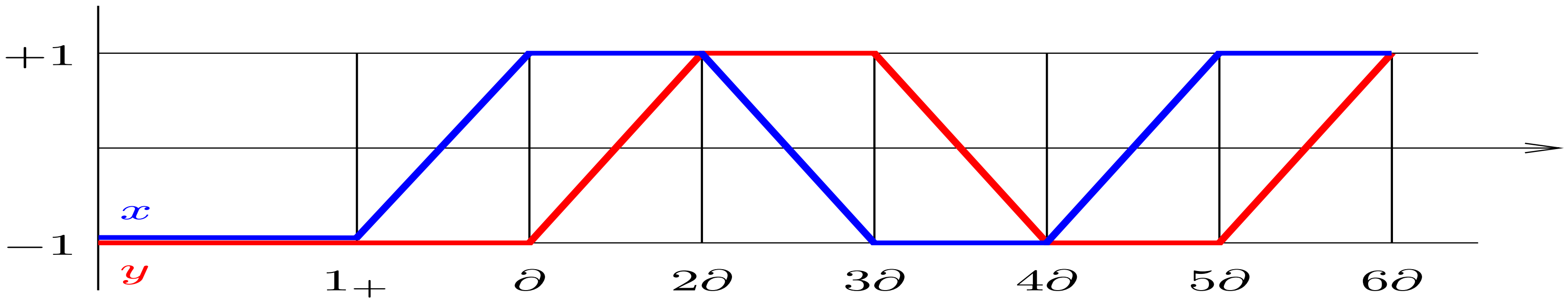
Example 1: reset an integrator on a zero-crossing event

```
let hybrid main () = (x, y) where
  rec der x = 1.0 init -1.0
      and der y = 0.0 init -1.0 reset up(x) → 1.0
```



Example 2: Unbounded cascades of zero-crossing

```
let hybrid main () = (x,y,z) where
  rec der x = 0.0 init -1.0
             reset up(y) → -. 1.0 | up(-. y) → 1.0 | up(z) → 1.0
  and der y = 0.0 init -1.0
              reset up(x) → 1.0 | up(-. x) → -1.0
  and der z = 1.0 init -1.0
```



- ∂ represent a "very small" step size in that finitely many ∂'s sum up to ≈ 0.
- At $t = 1$, x and y starts an infinite cascade of zero-crossing while time remains blocked. **This is certainly pathological**.

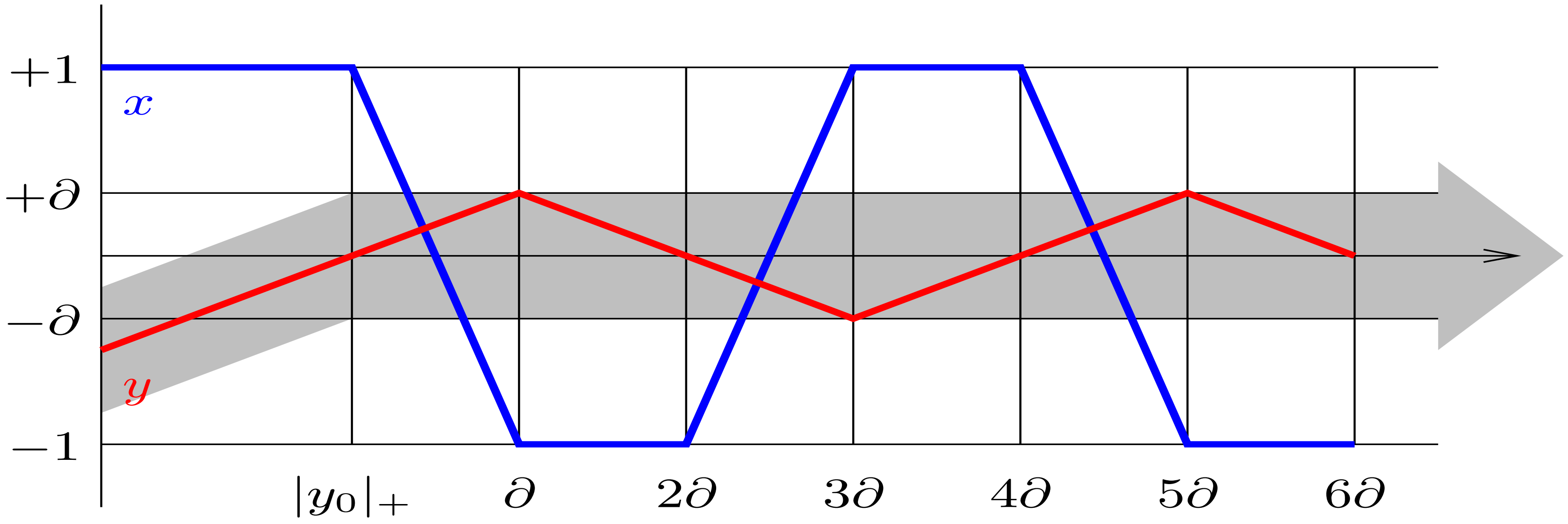
Currently, the `Zélus` compiler rejects it as there is an instantaneous loop between `x` and `y` because `up`(x) depends on x.

It could be made causal by considering the alternative choice for `up`(x): its effect is delayed by one cycle.

Example 3: Sliding mode control

```
let hybrid main() = y where
  rec x = present up(y) → -1.0 | up(-. y) → 1.0 init 1.0
  and der y = x init -. y0

let hybrid main2() = y where
  rec automaton
      | Up → do x = 1.0 until up(y) then Down
      | Down → do x = -1.0 until up(-. y) then Up
  and der y = x init -. y0
```



- y increases at constant speed until its first zero-crossing, just after $t = |y_0|$.
- Then, y chatters infinitesimally around 0 as its speed alternate between -1 and $+1$ with infinitesimal step ∂.

Is-this program pathological?

- It is not Zeno because time progresses strictly but by extremely small steps (with SUNDIALS CVODE).
- In [JCSS'12] we have said it to be equivalent to:

```
let hybrid main(y0) = y where
  rec der y = x init -. y0
  and x = present up(y) → 0.0 init 1.0
```

  We do not intend to obtain it automatically (or to prove the two to be equivalent).

- The causality analysis of `Zélus` does not complain as there is no cycle.

Cascade of zero-crossings

The following program shatters but is perfectly valid: it is never possible to fire twice in a row the zero-crossing `up(y -. 1.0)` without simulation time progressing.

```
let hybrid main2() = y where
  rec automaton
      | Up → do x = 1.0 until up(y -. 1.0) then Down
      | Down → do x = -1.0 until up(-1.0 -. y) then Up
  and der y = x init -. y0
```

But what to say if `y-.1.0` is replaced by `y` and `-1.0-.y` by `-.y`? It is not Zeno: time progresses by a very small step.

Avoid unbounded cascades of zero-crossing: A sufficient condition, checked at runtime: the very same zero-crossing cannot be fired more than once in a row during a sequence of zero-crossing transitions.

Open question: Can a static analysis identify those cases?

Causality analysis of instantaneous zero-crossings

Unbounded cascades of zero-crossings are pathological phenomena. They are a particular case of zeno.

Intuition: the very same zero-crossing must not be taken twice during a sequence of discrete reactions.

If S is a system, denote by Z_S the set of all its variables of zero-crossings. Write ψ_i for expressions $\texttt{up}(e_i)$.

[**Cascaded zero-crossings**] Let Z_S be the directed graph collecting all relations of the form $\psi \longrightarrow \psi'$. If Z_S contains no circuit, then all cascades of successive zero-crossings of S are provably finite.

Examples (1) and (3) are accepted. Example (2) is rejected.

Example (2): Let $z_1 = \texttt{up}(y)$, $z_2 = \texttt{up}(-y)$, $z_3 = \texttt{up}(z)$, $z_4 = \texttt{up}(x)$ and $z_5 = \texttt{up}(-x)$. The causality relation is:

- $y \to z_1$, $y \to z_2$, $z \to z_3$, $x \to z_4$. Then, $z_1 \to x$, $z_2 \to x$, $z_3 \to x$, $z_4 \to y$, $z_4 \to y$.
- Causality cycle: $z_1 \to x \to z_4 \to y \to z_1$

Compilation

The non-standard semantics is not operational. It serves as a reference to establish the correctness of the compilation. Two problems to address:

1. The compilation of the discrete part, that is, the synchronous subset of the language.

2. The compilation of the continuous part which is to be linked to a black-box numerical solver.

Principle

Translate the program into an only discrete one. Compile the result with an existing synchronous compiler such that it verifies the following invariant:

> **The discrete state, i.e., the values of delays, does not change when all of the zero-crossing conditions are false.**

Said differently: when those conditions are false, the function is combinatorial.

Zero-crossings

Does $\texttt{up}(e)$ only detects zero-crossing during integration? We provide several basic operations.

- $\texttt{up}(x)$ is the disjunction of two basic operations:
  1. $\texttt{upc}(x)$ detects a change in sign of x between negative to positive value and it is the responsability of the solver to implement it. When the detection is performed if abstract.
  2. $\texttt{upd}(x)$ is a discrete zero-crossing. It is an instant in $BaseClock(\partial)$ such that:

     $$(x(n-1) \leq 0) \wedge (x(n) > 0) \vee (x(n-1) < 0) \wedge (x(n) \geq 0) \wedge (time(n-1) + \partial = time(n))$$

     This is not implemented by the solver but directly as synchronous code. It is activated during a discrete step only:

```
false → ((last(x) <= 0.0) & (x > 0.0)
           or (last(x) < 0) & (x >= 0.0))
          & (last(time) = time)
```

Some other constructions are provided:

- e_1 `on` e_2 is present when e_1 is present and e_2 is true.
- `disc`(e) is present when e is not left continuous. Is-this construction useful?
- `period`(p) is present according to the period p.

A period is either of the form: (f_2) or $f_1(f_2)$. The corresponding event is present at instants: $k \times f_2 + f_1, (k \in I\!N)$.

Example: Running a process every one milisecond with initial delay of one:

```
present period(1(0.001)) → controller(x)
```

Example (counter)

Add extra input and outputs. The compilation produces a new synchronous program with extra inputs and outputs. Synchronous functions (nodes) stay unchanged. It is parameterized by a Boolean flag d, true in a discrete step, false otherwise[a].

1. `upc`(e) is turned into:
   (a) A fresh Boolean input z;
   (b) an equation $up_z = e$ with up_z as an extra output.
2. `der` $x = e$ `init` e_0 `reset` h is turned into:
   (a) An equation $x' = e$ that computes the instantaneous derivative;
   (b) an equation $x =$ `present` h `else` lx containing the current value for x;
   (c) the initial value $xi = e_0$ `->` lx.
3. The previous value `last` x is replaced by `if` d `then` `pre`(x) `else` x.

[a] An other option is to generate two functions, one used by the solver, the other for the discrete step.

Example (counter)

```
let node counter(top, tick) = o where
  rec der time = 1.0 init 0.0 reset z → 0.0
  and o = present z → counter(top, tick) init 0
 and z = upc(time -. 1.0)

let node counter(d, [z], [ltime], (top, tick)) = (o, [upz], [ntime])
 where
  rec time' = 1.0 and time = present z → 0.0 else ltime
  and o = present z → counter(top, tick) init 0
  and timei = 0.0 → ltime
  and ntime = if d then else timei
  and timeupz = (if d then pre(time) else time) -. 1.0
```

Compilation

For efficiency reason, represent extra inputs and outputs with arrays.

The function `counter` can be processed by any synchronous compiler, and the generated transition function verifies the invariant.

Open questions:

- Should we generate a single function parameterized by d or several (one for $d =$ `true`, one for $d =$ `false`)?
- Is the resulting function causal and in which sense?
- Is the resulting function statically schedulable?

The answer of the last two depends on the selected causality analysis.

Typing

The type language

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

Initial conditions

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

The Type system

Global and local environment

$$G ::= [f_1 : \sigma_1; ...; f_n : \sigma_n] \qquad H ::= [\,] \mid H, x : t$$

Typing predicates

- $G, H \vdash_k e : t$: Expression e has type t and kind k. $G, H \vdash_k e : t$
- $H, H \vdash_k E : H'$: Equation E produces environment H' and has kind k.

Subtyping

An combinatorial function can be passed where a discrete or continuous one is expected:

$$\forall k, \mathtt{A} \leq k$$

Continuous-time signals

Now, we must avoid $x \geq 1.0$, $x \leq 1.0$, $x = 1.0$ in a continuous context if x is not piece-wise constant and, thus, expressions like:

```
rec der time = 1.0 init 0.0
and x = sin(time)
and der t = if x >= 1.0 then 1.0 else -1.0 init 0.0
```

Proposition: Because the only signals that may change during integration are of value `float`, define `real`(k) such that:

- - $k =$ `A` for signals that are constant during integration.
  - $k =$ `C` for continuous signals (during integration).
  - $k =$ `D` for others (that may have discontinuities).
  - The type `float` is a short-cut for `real`(`A`).
- Restrict the use of polymorphism in a continuous context. A type variable α can only be instantiated by a type of piece-wise constant values.

Kind of a type: A type t is of kind k, written $k(t)$ if the following applies:

$$\textsc{(prod)}\ \frac{k(t_1) \qquad k(t_2)}{k(t_1 \times t_2)} \qquad \textsc{(int)}\ k(\texttt{int}) \qquad \textsc{(bool)}\ k(\texttt{bool}) \qquad \textsc{(real)}\ k(\texttt{real}(k)) \qquad \textsc{(alpha)}\ k(\alpha)$$

Instanciation of a polymorphic type:

$$\textsc{(Inst)}\ \frac{k \leq k' \qquad k' \neq \texttt{C}}{(t \xrightarrow{k'} t')[\vec{t_0}/\vec{\alpha}] \in \mathit{Inst}(k')(\forall \vec{\alpha}.t \xrightarrow{k} t')}$$

$$\textsc{(Inst-cont)}\ \frac{k \leq k' \qquad k' = \texttt{C} \qquad \texttt{A}(t_0)}{(t \xrightarrow{k'} t')[\vec{t_0}/\vec{\alpha}] \in \mathit{Inst}(k')(\forall \vec{\alpha}.t \xrightarrow{k} t')}$$

In a continuous context, polymorphic variables can only be instanciated by a type of piece-wise constant signals.

Example: If $x : \texttt{real}(\texttt{C})$, the expression $1.0 = x$ in a context $\texttt{C}$ is wrongly typed. $(=) : \forall \alpha.\alpha \times \alpha \xrightarrow{\texttt{A}} \texttt{bool}$ and α cannot be replaced by $\texttt{real}(\texttt{C})$.

A sketch of Typing rules

(DER)

$$\frac{G, H \vdash_{\mathtt{C}} e_1 : \mathtt{real}(\mathtt{C}) \qquad G, H \vdash_{\mathtt{C}} e_2 : \mathtt{real}(ki_1) \qquad G, H \vdash h : \mathtt{real}(ki_2)}{G, H \vdash_{\mathtt{C}} \mathtt{der}\, x \;=\; e_1 \;\mathtt{init}\; e_2 \;\mathtt{reset}\; h : [x : \mathtt{real}(\mathtt{C})]}$$

(AND)

$$\frac{G, H \vdash_k E_1 : H_1 \qquad G, H \vdash_k E_2 : H_2}{G, H \vdash_k E_1 \;\mathtt{and}\; E_2 : H_1 + H_2}$$

(EQ)

$$\frac{G, H \vdash_k e : t}{G, H \vdash_k x \;=\; e : [x : t]}$$

(APP)

$$\frac{t \xrightarrow{k} t' \in \mathit{Inst}(k)(G(f)) \qquad G, H \vdash_k e : t}{G, H \vdash_k f(e) : t'}$$

(VAR)

$$G, H + [x : t] \vdash_k x : t$$

(VAR)

$$G, H + [x : t] \vdash_k \mathtt{last}\, x : t$$

(EQ-DISCRETE)

$$\frac{G, H \vdash h : t \qquad G, H \vdash_{\mathtt{D}} e : t}{G, H \vdash_{\mathtt{C}} x \ = \ h \ \mathtt{init}\, e : [x : t]}$$

(HANDLER)

$$\frac{\forall i \in \{1, .., n\} \qquad G, H \vdash_{\mathtt{D}} e_i : t \qquad G, H \vdash_{\mathtt{C}} z_i : \mathtt{zero}}{G, H \vdash z_1 \rightarrow e_1 \parallel ... \parallel z_n \rightarrow e_n : t}$$

Causality analysis

Causality

Consider first the simplest case (that of `Lustre` and `Lucid Synchrone`). Any loop must cross a delay (possibly hidden).

Condition: check that the relation between a set of variables is a partial order.

The typing jugment:

$$C, H \vdash e : ct$$

means that under constraint C, type environment H, e gets type t.

Type language:

$$\begin{array}{rcl} \sigma & ::= & \forall \alpha_1, ..., \alpha_n : C.ct \rightarrow ct \\ ct & ::= & ct \times ct \mid \alpha \end{array}$$

Environment and Constraint:

$$H \quad ::= \quad [x_1 : ct_1; ...; x_n : ct_n] \qquad\qquad C \quad ::= \quad \{\alpha_1 < \alpha'_1, ..., \alpha_n < \alpha'_n\}$$

The global environment is left implicit:

$$G \quad ::= \quad [\sigma_1/f_1, ..., \sigma_k/f_k]$$

Relation between types

C must define a partial order, i.e., it is not possible to deduce $C \vdash ct < ct$.

(TAUT)

$$C + \alpha_1 < \alpha_2 \vdash \alpha_1 < \alpha_2$$

(TRANS)

$$\frac{C \vdash ct_1 < ct' \qquad C \vdash ct' < ct_2}{C \vdash ct_1 < ct_2}$$

(PAIR)

$$\frac{C \vdash ct_1 < ct'_1 \qquad C \vdash ct_2 < ct'_2}{C \vdash ct_1 \times ct_2 < ct'_1 \times ct'_2}$$

(ENV)

$$\frac{\forall i \in \{1, .., n\}, C \vdash ct_i < ct'_i}{C \vdash [x_1 : ct_1; ...; x_n : ct_n] < [x_1 : ct'_1; ...; x_n : ct'_n]}$$

Basic operations over types:

Initial environment:

$$
\begin{array}{lcl}
(+) & : & \forall \alpha . \alpha \times \alpha \rightarrow \alpha \\
\texttt{pre}(\cdot) & : & \forall \alpha, \alpha' . \alpha \rightarrow \alpha' \\
\cdot\,\texttt{fby}\,\cdot & : & \forall \alpha, \alpha' . \alpha \times \alpha' \rightarrow \alpha
\end{array}
$$

Instantiation/Generalisation:

(Inst

$$C[\vec{\alpha'}/\vec{\alpha}], (ct_1 \rightarrow ct_2)[\vec{\alpha'}/\vec{\alpha}] \in \mathit{Inst}(\forall \vec{\alpha} : C.ct_1 \rightarrow ct_2)$$

(Gen)

$$\frac{\mathit{Vars}(C) = \{\alpha_1, ..., \alpha_n\}}{\mathit{Gen}(C)(ct_1 \rightarrow ct_2) = \forall \alpha_1, ..., \alpha_n : C.ct_1 \rightarrow ct_2}$$

Typing rules

(Var)

$$C, H + x : ct \vdash_k x : ct$$

(Const)

$$C, H \vdash_k i : ct$$

(App)

$$\frac{C, ct_1 \rightarrow ct_2 \in Inst(G(f)) \qquad C, H \vdash_k e : ct_1}{C, H \vdash_k f(e) : ct_2}$$

(And)

$$\frac{C, H \vdash_k D_1 : H_1 \qquad C, H \vdash_k D_2 : H_2}{C, H \vdash_k D_1 \texttt{ and } D_2 : H_1 + H_2}$$

(Eq)

$$\frac{C, H \vdash_k e : ct}{C, H \vdash_k x = e : [ct/x]}$$

(Sub)

$$\frac{C, H \vdash_k e : t \qquad C \vdash ct < ct'}{C, H \vdash_k e : t'}$$

(Def)

$$\frac{C, H \vdash_k D : H' \qquad C \vdash H' < H \qquad C, H \vdash_k x : ct_1 \qquad C, H \vdash_k y : ct_2}{\vdash \texttt{let } k\, f\ (x) = y \texttt{ where } D : [Gen(C)(ct_1 \rightarrow ct_2)/f]}$$

Example: The first is accepted; the second is rejected.

```
let node f x = y where rec y = 0 → pre y + x
let node g x = y where rec y = x + 1 and x = y + 2
```

since $\{\alpha_x < \alpha_y, \alpha_y < \alpha_x\}$ is not a partial order.

The following two programs are accepted:

```
let node f x = y where rec y = 0 → pre x + 1
let node g x = y where rec y = f(y) + x
```

We get: $f : \forall \alpha_x, \alpha_y . \alpha_x \rightarrow \alpha_y$ and $g : \forall \alpha_x, \alpha_y : \{\alpha_x < \alpha_y\} . \alpha_x \rightarrow \alpha_y$.

Note that this causality analysis is such that the code generation of f cannot be done by generating a single step function.

Adding ODEs

We need a way to get the "value of a signal `x` just before changing it". We write it `last` x. It is the left-limit of x.

Examples of non causal programs:

1. `rec x = x +. 1.0 and der z = x init 0.0`

2. `rec x = last x + 1`

3. ```
   rec der y' = -. g init 0.0 reset up(-.y) → -0.9 *. y'
   and der y = y' init y0
   ```

Examples of causal programs:

1. `rec der x = v -. x init x0`

2. ```
   rec der y' = -. g init 0.0 reset up(-.y) → -0.9 *. last y'
   and der y = y' init y0
   ```

Causality of $\texttt{last}\, x$

What would be the causality type for $\texttt{last}\, x$. A first solution:

$$\frac{(\textsc{Last})}{C, H + x : ct \vdash_{\texttt{D}} \texttt{last}\, x : ct'}$$

$\texttt{last}\, x$ is only possible in a discrete context. Can we do a little better?

Causality analysis

In non-standard semantics [JCSS'12], $\mathtt{der}\, x = e\ \mathtt{init}\ e_0$ defines x so that:

$$x(0) = e_0(0) \quad x(n) = x(n-1) + \partial \times e(n-1) \text{ with } n \in {}^\star\mathbb{N}$$

$x(n)$ is the value of x at time $n \times \partial \in {}^\star\mathbb{R}$.

The left limit $\mathtt{last}\, x$ of a signal

$$\mathtt{last}\, x(n) = x(n-1)$$

- When x is left-continuous, it coincides with the left limit since $\mathtt{last}\, x(n) \approx x(n)$;
- Otherwise, it is the previously computed value of x.

So $\mathtt{last}\, x$ does not always break algebraic loops.

What could be the implementation of $\mathtt{last}\, x$?

$$\mathtt{last}\, x = \mathtt{if}\ d\ \mathtt{then}\ \mathtt{pre}(x)\ \mathtt{else}\ x$$

Examples

The integrator plays the role of the unit delay: it breaks cycles in continuous steps. The following two programs are causally correct:

```
let hybrid f(x) = o where
  der y = x +. y init 0.0 and o = y +. 1.0


let hybrid loop() = y where
  rec y = f(y)
```

`y` does not depends instantaneously of `x`.

`last` x does not necessarily break causality loops, i.e.:

```
let hybrid g(v) = o where
  rec der y = 1.0 init 0.0
  and x = last x +. y and init x = 0.0
```

The reason is this: if $x : \alpha_x + \alpha'_x$ and $y : \alpha_y + \alpha'_y$, then `last` $x : \alpha''_x + \alpha'_x$. We also have: `last` $x +.\ y : \alpha_y + \alpha$ with $\alpha'_y, \alpha'_x < \alpha$. There is a causality cycle since $\alpha'_y, \alpha'_x < \alpha \not< \alpha'_x$, that is, `x` instantaneously depends on itself.

If $\texttt{up}(x)$ instantaneously depends on x during discrete time, we could first give it the signature:

$$\texttt{up}(.) : \forall \alpha . \alpha \rightarrow \alpha$$

Taking this, the following program is accepted:

```
let hybrid h(x) = o where
  der y = 1.0 -. x init 0.0 reset up(x) → last x and o = y +. 1.0


let hybrid loop() = y where rec y = f(y) and init y = 0.0
```

Then: $h : \forall \alpha . \alpha \rightarrow \alpha$ is valid.

By decomposing $\texttt{up}(.)$ into the union of the detection of a discrete zero-crossing (a radical change of x) and the detection of a continuous-one (during integration), we can be a little more precise:

$$\texttt{upc}(.) : \forall \alpha_1, \alpha_2 . \alpha_1 \rightarrow \alpha_2$$

That is, $\texttt{upc}(x)$ does not instantaneously depend on x.

Causality analysis (bis)

Thus, `last` x only breaks an algebraic loop during discrete steps (when x is not left-continuous), that is, d is true.

This reminds conditional dependences of the `Signal` language. $x \xrightarrow{c} y$ states that y depends on x when c is true. We take a simpler solution.

Idea: associate a pair $ct_1 + ct_2$ to every expression e.

- during discrete steps, e only depends on ct_1;
- during integration steps, e only depends on ct_2.

Type language:

$$
\begin{array}{rcl}
\sigma & ::= & \forall \alpha_1, ..., \alpha_n : C.ct \rightarrow ct \\
ct & ::= & ct \times ct \mid ct + ct \mid \alpha
\end{array}
$$

Shortcut: Write $ct_1 + ct_2$ so that $(ct_1 \times ct'_1) + (ct_2 \times ct'_2)$ stands for $(ct_1 \times ct_2) + (ct'_1 \times ct'_2)$.

Intuition:

$$\text{(L}\textsc{ast}\text{)}\quad C, H + [x : ct_1 + ct_2] \vdash \texttt{last}\,x : ct'_1 + ct_2$$

During a discrete step, $\texttt{last}\,x$ does not depend on x but it does depend on x during a continuous step.

If $\texttt{up}(x)$ instantaneously depends on x during discrete time, we could first give it the signature:

$$\texttt{up}(.) : \forall \alpha_1, \alpha_2 . \alpha_1 + \alpha_2 \rightarrow \alpha_1 + \alpha_4$$

Taking this, the following program is accepted:

```
let hybrid h(x) = o where
  der y = 1.0 -. x init 0.0 reset up(x) → x and o = y +. 1.0


let hybrid loop() = y where rec y = f(last y) and init y = 0.0
```

Then: $h : \forall \alpha_1, \alpha_2, \alpha_3 : \alpha_1 + \alpha_2 \rightarrow \alpha_1 + \alpha_3$

is valid, as `last y` breaks the cycle on α .

New typing rules

(Der)

$$\frac{C, H \vdash e : ct \qquad C, H \vdash e_0 : ct'}{C, H \vdash \texttt{der}\, x \ = \ e\ \texttt{init}\ e_0 : [ct'/x]}$$

(Discrete)

$$\frac{C, H \vdash e : ct \qquad C, H \vdash E_1 : H_1 \qquad C, H \vdash E_2 : H_2}{C, H \vdash \texttt{present}\ e\ \texttt{then}\ E_1\ \texttt{else}\ E_2 : H_1 * H_2}$$

(Last)

$$C, H + x : ct_1 + ct_2 \vdash \texttt{last}\, x : ct_1' + ct_2$$

Examples:

The following one is rejected:

```
let hybrid f(z) = y where
  der y = 1.0 init -1.0 reset up(z) → -.1.0

let hybrid loop() = y where
  rec y = f(y)
```

If we consider that `up`(x) instantaneously depends on x.

Types for zero-crossing detections

If $\texttt{upc}(.)$ only detect zero-crossing during integration, its type signature could be:

$$\texttt{upc}(.) : \forall \alpha_1, \alpha_2 . \alpha_1 \rightarrow \alpha_2$$

If $\texttt{upd}(.)$ only detect zero-crossing during a discrete step, its type signature could be:

$$\texttt{upd}(.) : \forall \alpha_1, \alpha_2, \alpha_3 . \alpha_1 + \alpha_2 \rightarrow \alpha_1 + \alpha_3$$

If $\texttt{up}(.)$ do both (union), its type signature would be:

$$\texttt{up}(.) : \forall \alpha_1, \alpha_2 . \alpha_1 + \alpha_2 \rightarrow \alpha_1 + \alpha_3$$

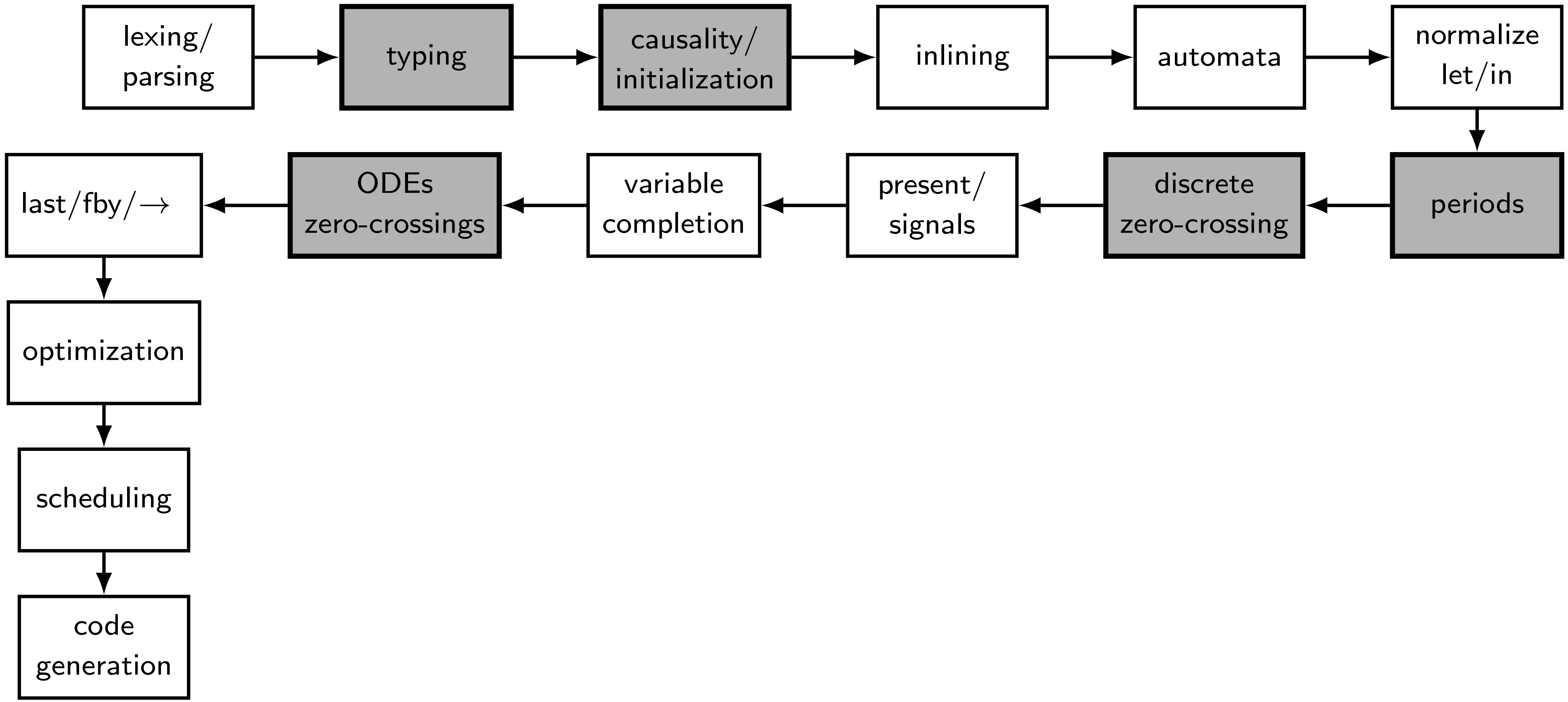
Should we try to avoid discrete cascades? Give type signature:

$$\texttt{up}(.) : \forall \alpha . \alpha + \alpha \rightarrow \alpha + \alpha$$

Conclusion (1)

Zélus is an experimental prototype which mix the expressiveness of synchronous programming with ODEs.

Many ideas borrowed from **Lucid Synchrone**.



Historial note

- Version 1. 2014 (approx).
  - First-order functional language with mixed (discrete/continuous) signals;
  - An ideal semantics based on non standard analysis;
  - A type system to separate discrete-time from continuous-time signals;
  - A type-based causality analysis; initialization analysis.
- Version 2. 2016 (approx).
  - A new compilation technique [BCP+15];
  - Industrial prototype **Scade Hybrid** [BCP+15] built on Scade KCG (ANSYS);
  - Language novelties: higher-order; static values and parameters; code specialisation at compile-time;
  - Probabilitic constructs (ProbZelus): 2020 [BMA+20]
  - First experiment with SISAL array operations; definition of a standard library of control blocks [BCC+17].
- Version 3. 2021 -. Built on the definition of a reference interpreter and constructive semantics (`zrun`). Complete rewriting of the compiler.

Sources of the compiler, doc, examples and manual

Webpage (doc, examples, papers):

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

Source code, examples:

`https://github.com/inria/zelus/tree/main`

The (very begining) of version 3:

`https://github.com/inria/zelus/tree/work`

References

[BBC+14] 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.

[BBCP11a] 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.

[BBCP11b] 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.

[BBCP12] 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.

[BCC+17] 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.

[BCP10] 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.

[BCP+15] 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.

[BMA+20] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Reactive Probabilistic Programming. In *International Conference on Programming Language Design and Implementation (PLDI)*, London, United Kingdom, June 15-20 2020. ACM.

[BP13] 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.