Zelus, a Programming Language for Hybrid Systems

Marc Pouzet

ENS

Marc.Pouzet@ens.fr

Cours “Programmation synchrone”, M2
Université de Paris
December 8, 2020

Synchronous data-flow programming

Synchronous Dataflow programming

E.g., Lustre/Heptagon/Lucid Synchrone/SCADE/Zelus [1]

Basic primitives: time is discrete and logical (indices in $\mathbb{N}$)

- $o = 1$ means $o(n) = 1$ $\forall n \in \mathbb{N}$
- $o = x + y$ means $o(n) = x(n) + y(n)$ $\forall n \in \mathbb{N}$
- $o =$ **pre** x means $o(n) = x(n-1)$ $\forall n \in \mathbb{N}$ when $n > 0$
- $o = x \rightarrow y$ means $o(0) = x(0)$ and $o(n) = y(n)$ $\forall n \in \mathbb{N}$ when $n > 0$
- $o = x$ **fby** y means $o = x \rightarrow$ (**pre** y)

[1] https://zelus.di.ens.fr

Synchronous Dataflow programming

E.g., Lustre/Heptagon/Lucid Synchrone/SCADE/Zelus [1]

Basic primitives: time is discrete and logical (indices in $\mathbb{N}$)

- o = 1 means $o(n) = 1$ $\forall n \in \mathbb{N}$
- o = x + y means $o(n) = x(n) + y(n)$ $\forall n \in \mathbb{N}$
- o = **pre** x means $o(n) = x(n-1)$ $\forall n \in \mathbb{N}$ when $n > 0$
- o = x $\rightarrow$ y means $o(0) = x(0)$ and $o(n) = y(n)$ $\forall n \in \mathbb{N}$ when $n > 0$
- o = x **fby** y means o = x $\rightarrow$ (**pre** y)

Programs = sets of mutually-recursive equations defining sequences

let boom_rate = 0.4

val boom_rate : float

let inc x = x + 1

val inc : int $\xrightarrow{A}$ *int*

let node after (n, t) = (c = n) **where**
rec c = 0 **fby** min ((**if** t **then** c + 1 **else** c), n)

val after : int $\times$ *bool* $\xrightarrow{D}$ *bool*

[1] https://zelus.di.ens.fr

Dataflow programming: causality (à la Lustre)

- Programs are functional and causal: single variable at left
- No instantaneous feedback.
- Every loop must be broken by a delay.
- This is ensured by a static causality analysis.
- Allows compilation to statically-scheduled code.
- Kahn semantics: all valid schedules give the same result.

let node f x = y **where**
  rec y = p + x
  and p = y + 1

```
File "prog.zls", line 1-3, characters 15-54:
>...............y where
>  rec y = p + x
>  and p = y + 1
Causality error: this expression may instantaneously depend on itself.
Here is an example of a cycle:[p --> y y --> p p --> p]
```

Stream programs (that is, discrete-time systems)

let node after (n, t) = (c = n) **where**
  rec c = 0 **fby** min ((**if** t **then** c + 1 **else** c), n)

$\texttt{val}\ \mathit{after}\ \texttt{:}\ \mathit{int} \times \mathit{bool} \xrightarrow{\texttt{D}} \mathit{bool}$

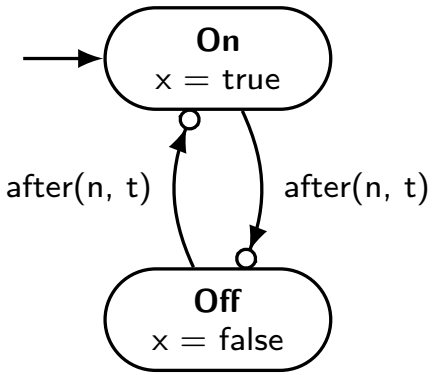
Stream programs (that is, discrete-time systems)

let node after (n, t) = (c = n) **where**
  rec c = 0 **fby** min ((**if** t **then** c + 1 **else** c), n)

$\mathit{val\ after : int \times bool \xrightarrow{D} bool}$

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

$\mathit{val\ blink : int \times bool \xrightarrow{D} bool}$



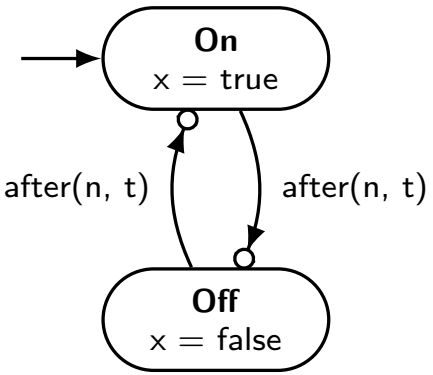
Stream programs (that is, discrete-time systems)

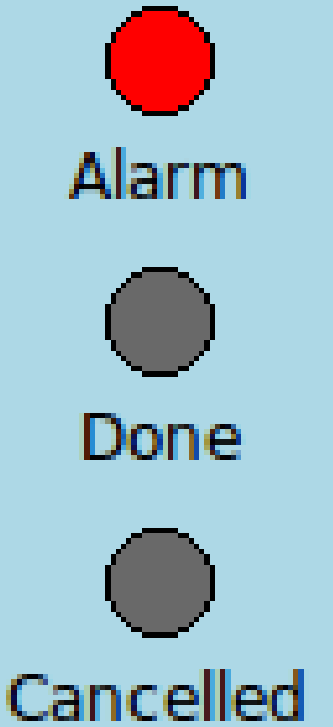
let node after (n, t) = (c = n) **where**
  rec c = 0 **fby** min ((**if** t **then** c + 1 **else** c), n)

$\mathit{val\ after : int \times bool \xrightarrow{D} bool}$

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

$\mathit{val\ blink : int \times bool \xrightarrow{D} bool}$





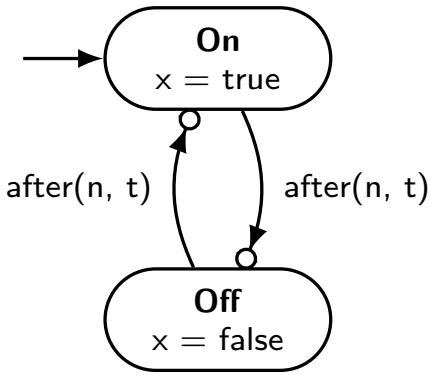
Stream programs (that is, discrete-time systems)

let node after (n, t) = (c = n) **where**
  rec c = 0 **fby** min ((**if** t **then** c + 1 **else** c), n)

$$\texttt{val}\ after : int \times bool \xrightarrow{D} bool$$

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

$$\texttt{val}\ blink : int \times bool \xrightarrow{D} bool$$



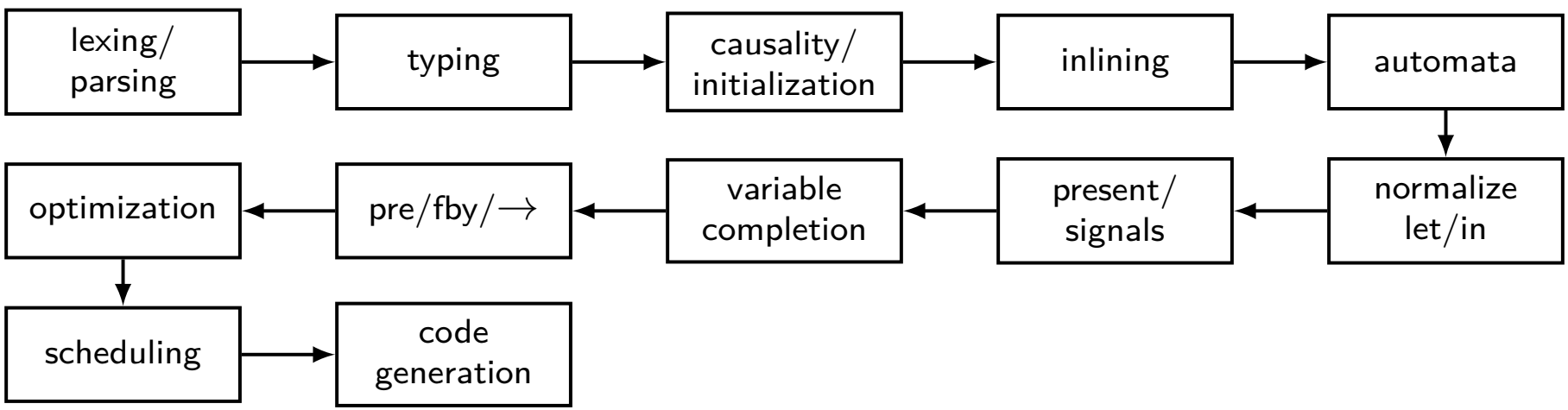
let node main second =
  let alarm = blink (3, second) **in**
  show (alarm, false, false)

$$\texttt{val}\ main : bool \xrightarrow{D} unit$$

Dataflow programming: control structures

- Add conditional activation conditions (clocks):
  - sub-sampling, over-sampling, and merging.
  - static clock calculus guarantees bounded time and memory.

- Build more sophisticated constructs:
  - modular resets, automata, signals, etc.
  - Successive compilation into smaller and smaller subsets.
  - Finally, convert base primitives into sequential code.
  - Modular compilation is possible (if a little tricky).



What to retain

The Synchronous Model

Write a mathematical, ideal and portable specification of a reactive system.

Supposing that time is global, discrete and logical as if all processes were running in parallel synchronously.

Check important safety properties on this ideal model: is-it deterministic? deadlock free?, can it be implemented into code that runs in bounded time and space?

Finally check that the implementation is fast enough.

This approach is not limited to synchronous languages but is of paramount importance in all modeling tools for embedded systems.

E.g., Simulink is not a "synchronous language" per se; but provides means to write a synchronous model.

Time is logical: $time = \mathbb{N}$.

With no reference, in the semantics, about the duration of a reaction.

In a data-flow functional language (e.g., Lustre/Heptagon/Scade/Lucid Synchrone/Zelus):

- A signal x is a sequence or stream $(x_n)_{n \in \mathbb{N}}$;
- a system is a stream function $I^{\mathbb{N}} \to O^{\mathbb{N}}$.

Specify the static semantics, compilation steps and the dynamic semantics to give strong confidence that the generated code implement the same function.

E.g., Scade whose compiler is qualified for certified software (read [CPP17]).

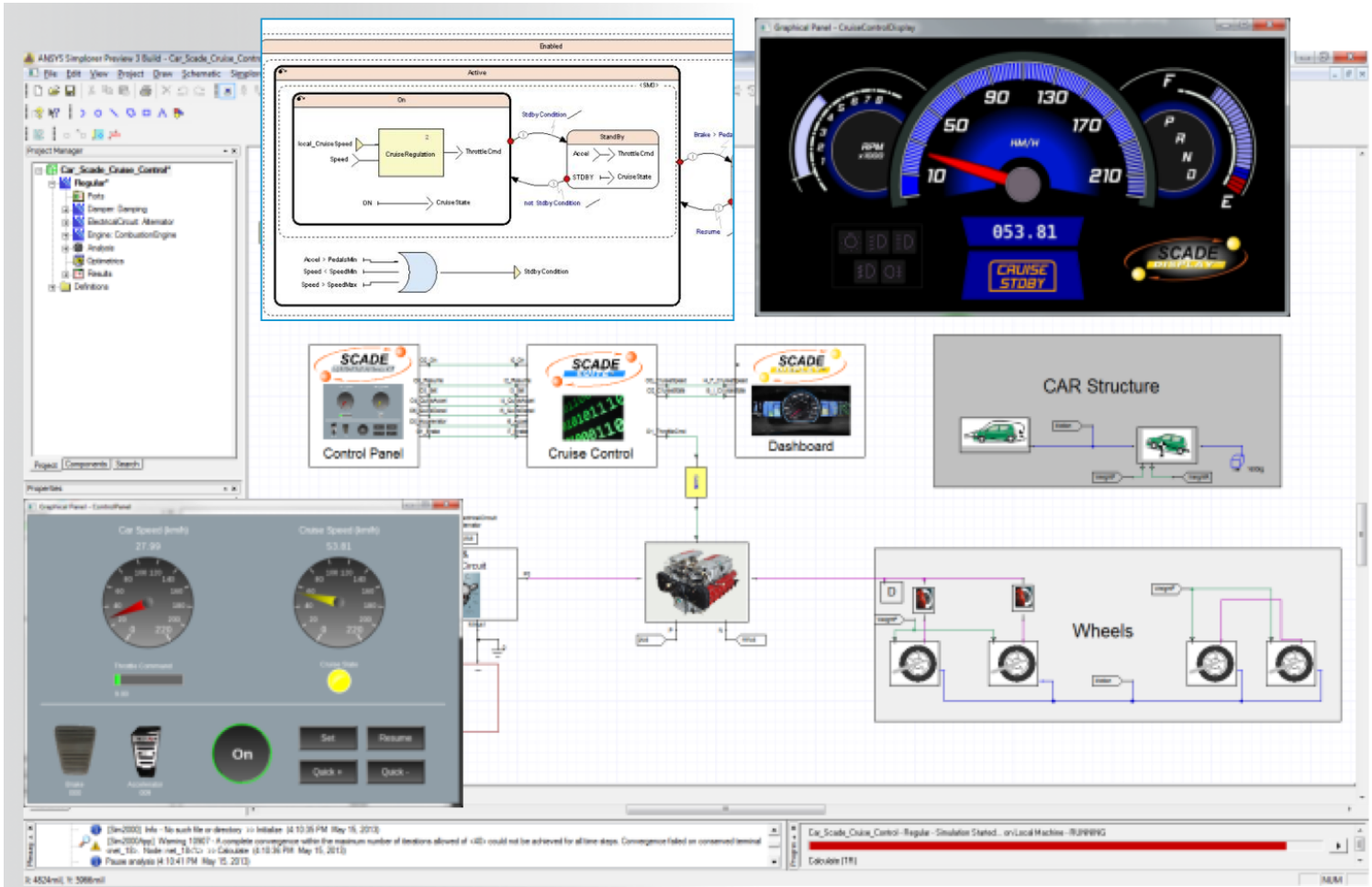
E.g., The formally verified Velus [2] compiler of Lustre [BBD+17, BBP20]

[2] velus.inria.fr

What about mixed or hybrid discrete and continuous-time models?

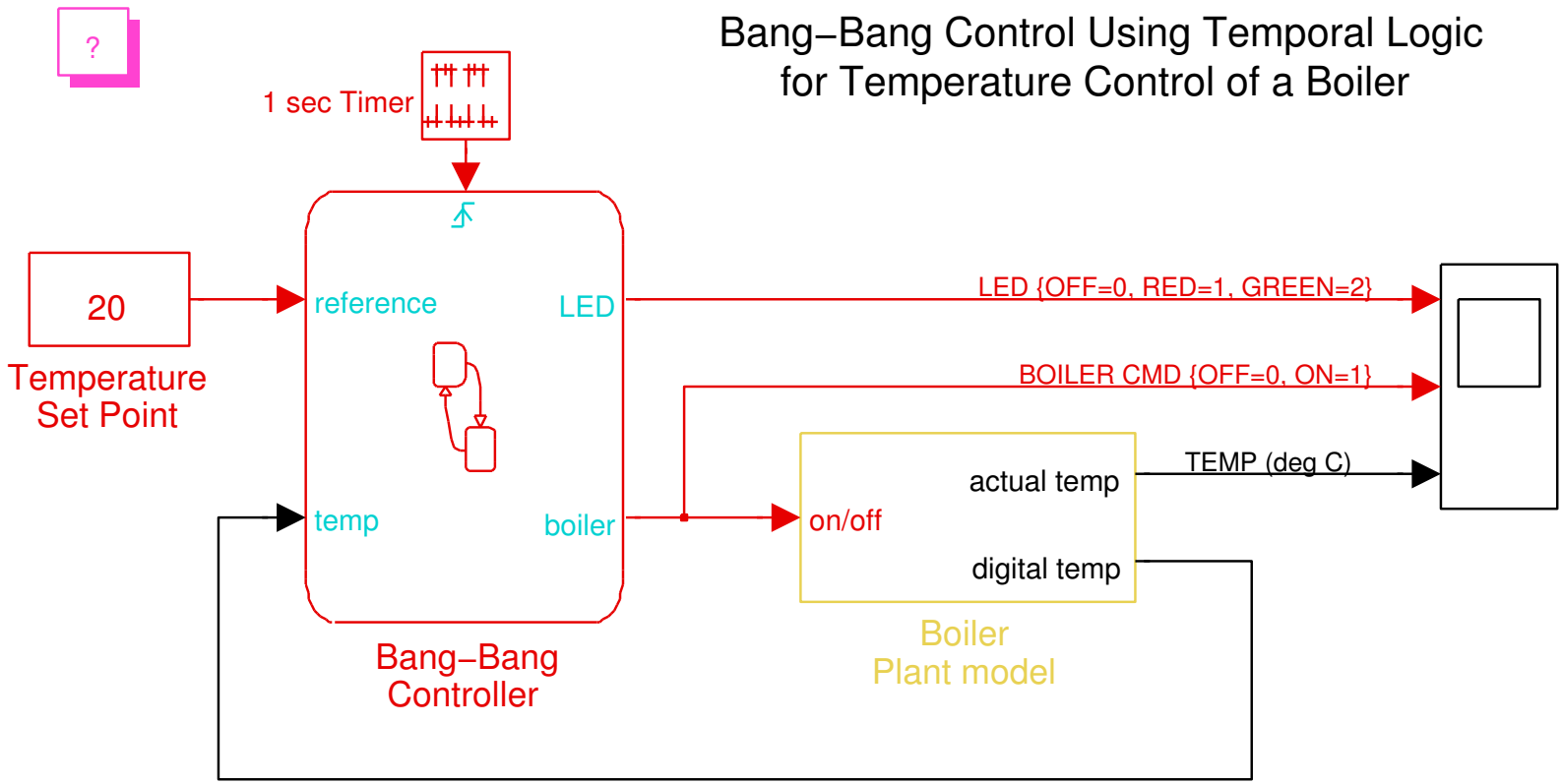
Example: the cruise control in its environment[3]

Model the whole system: the controller and the plant.



[3] Image from ANSYS/Esterel-Technologies

Example: a Bang-bang controller [4]



[4] https://www.mathworks.com/help/stateflow/ug/bang-bang-control-using-temporal-logic.html

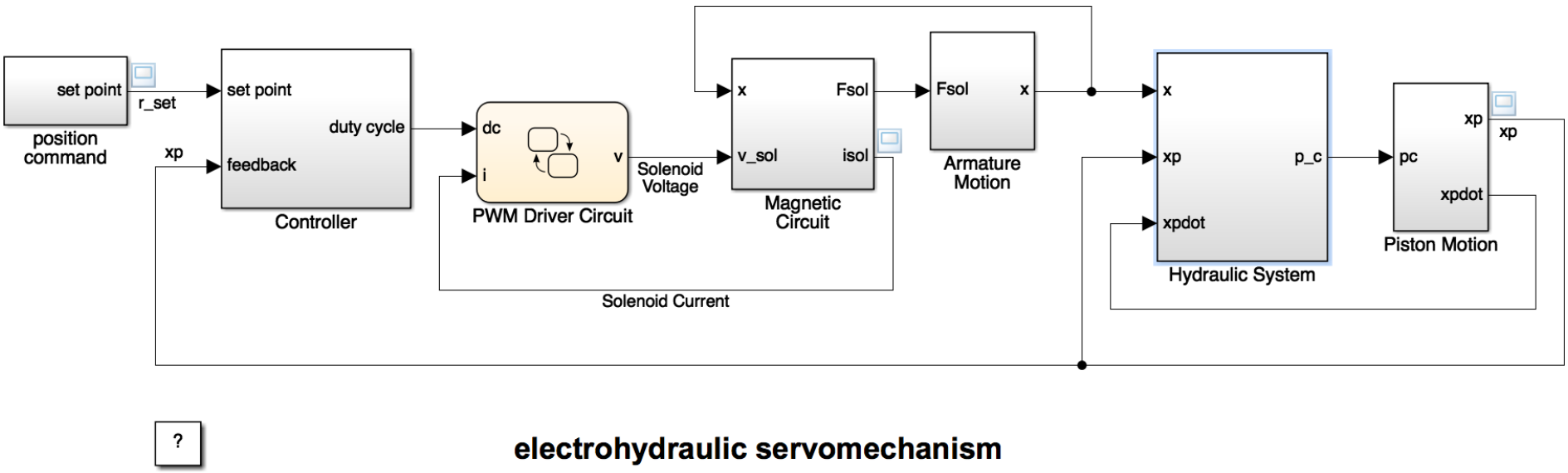
What is hybrid?

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.

For example [5]



[5] Image taken from the standard distribution of Simulink

Why?

It is possible to write a discrete-time approximation of a continuous-time model.

that is, to write, difference equations instead of differential equations.

but it is either slow or unprecise (and possibly both).

Why?

It is possible to write a discrete-time approximation of a continuous-time model.

that is, to write, difference equations instead of differential equations.

but it is either slow or unprecise (and possibly both).

Why?

It is possible to write a discrete-time approximation of a continuous-time model.

that is, to write, difference equations instead of differential equations.

but it is either slow or unprecise (and possibly both).

Three simple examples [DEMO]

The Van der Pool oscillator.

Complicated integrals.

Bouncing ball (hybrid models).

What to retain

For complicated (stiff) dynamics, a solver of differential equations works far better that the synchronous encoding.

It is fast when the dynamics is slow (smooth).

Slow when the dynamics is fast (stiff).

Overall, the simulation is faster, more faithful.

A Wide Range of Hybrid Systems Modelers Exist

Ordinary Differential Equations + discrete time

Simulink/Stateflow ($\geq 10^6$ licences), LabView, Ptolemy II, etc.

Differential Algebraic Equations + discrete time

Modelica, VHDL-AMS, VERILOG-AMS, etc.

Dedicated tools for multi-physics

Mechanics, electro-magnetics, fluid, etc.

Co-simulation/combination of tools

Common formats/protocols: FMI/FMU, S-functions, etc.

So what's to study?

- Tools normally excel in a single domain.
- Continuous: well understood. Discrete: well understood. Hybrid?
- Why do they work (or not)? What are the underlying principles?

In those languages, the compiler has a central role

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

A complex compilation chain;

E.g., static typing and rejection of certain models, static scheduling, inlining, rewriting, separation of the continuous/discrete-time part, data representations, link with a solver.

For hybrid models, the run-time system may involve one (or several) differential equations solver(s).

Can we trust the compiler in order to ensure that what is simulated is what is written and execute on the target?

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

Can we mix (and how) best of both worlds?

Synchronous models to describe the software and the discrete-time dynamics.

Differential equations, zero-crossing events to model continuous-time.

Use of-the-shelf numerical solvers in the (simulation) loop.

Can we mix (and how) best of both worlds?

Synchronous models to describe the software and the discrete-time dynamics.

Differential equations, zero-crossing events to model continuous-time.

Use of-the-shelf numerical solvers in the (simulation) loop.

Can we mix (and how) best of both worlds?

Synchronous models to describe the software and the discrete-time dynamics.

Differential equations, zero-crossing events to model continuous-time.

Use of-the-shelf numerical solvers in the (simulation) loop.

Can we mix (and how) best of both worlds?

Synchronous models to describe the software and the discrete-time dynamics.

Differential equations, zero-crossing events to model continuous-time.

Use of-the-shelf numerical solvers in the (simulation) loop.

Approach: add ODEs to a synchronous dataflow language

Reuse existing principles and techniques

Synchronous languages (SCADE/Lustre)

Expressive language for both discrete controllers and mode changes.

Off-the-shelf ODE numerical solvers

Simulate with external, off-the-shelf, variable-step numerical solvers

Two concrete reasons

- Increase modeling power *(hybrid programming)*.
- Exploit existing compiler *(target for code generation)*.

Conservative: any synchronous program must be compiled, optimized, and executed as per usual.

Support for hybrid modelling

The Bouncing Ball

```
let g = 9.81

let hybrid ball(y0, y'0) = (y, y', z) where
    rec der y = y' init y0
    and der y' = -. g init y'0 reset z -> -. 0.9 *. last y'
    and z = up(-. y)
```

```
val g : float
val ball : float × float -C-> float × float × zero
```

- When $-.$ y crosses zero, re-initialize the speed y' with $-$ 0.9 $*.$ **last** y'.
- **last** y' stands for the previous value of y'.
- As y' is immediately reset, writing **last** y' is mandatory —otherwise, y' would instantaneously depend on itself.

Bouncing ball with automata

```
let eps = 0.001

let hybrid bouncing_with_two_modes(y0, y'0) = (y, y') where
rec automaton
    | Bouncing →
      do y, y', z = ball(y0, y'0)
      until z on (y' < eps) then Sliding
    | Sliding →
      do y = 0.0 and y' = 0.0 done
    end
```

```
val eps : float
val bouncing_with_two_modes : float × float -C-> float × float
```

Continuous and discrete PI controller

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

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

let ts = 0.05

let node disc_integr(x, y0) = y where
  rec y = y0 -> last y +. ts *. x

let node disc_pi(kp, ki, i) = cmd where
  rec cmd = kp *. i +. ki *. disc_integr(i, 0.0)
```

$$\texttt{val } integr : float \times float \xrightarrow{C} float$$
$$\texttt{val } pi : float \times float \times float \xrightarrow{C} float$$
$$\texttt{val } ts : float$$
$$\texttt{val } disc\_integr : float \times float \xrightarrow{D} float$$
$$\texttt{val } disc\_pi : float \times float \times float \xrightarrow{D} float$$

Support for hybrid modelling

der o = x **init** v **reset** z0 → v0 | z1 → v1 | ...

means $o(0) = v$

$$o(t) = v(0) + \int_0^t x(\tau)\, d\tau \qquad \forall t \in \mathbb{R}$$

reset to v_i on event z_i; also up(e) and **last** x.

Raises several interrelated questions

1. Which compositions make sense?
2. What do they mean?
3. How should causality (loops) be handled?
4. How to compile programs?

Entering the land of real languages...

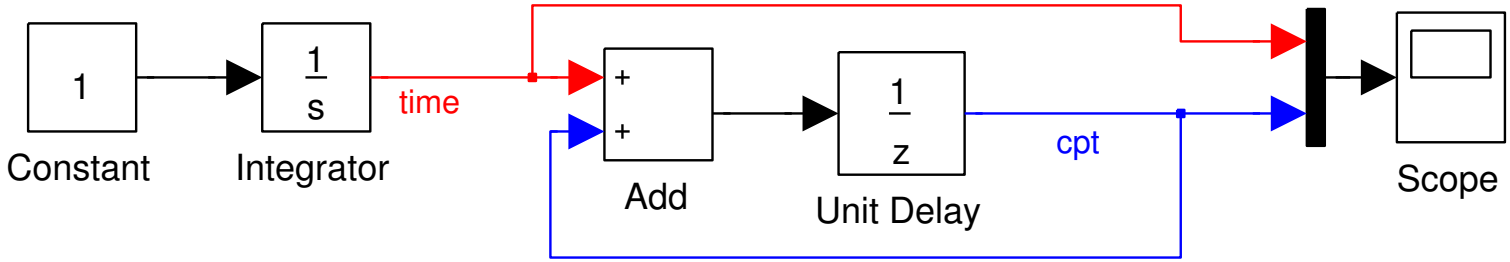
where are the monsters?

Entering the land of real languages...
where are the monsters?

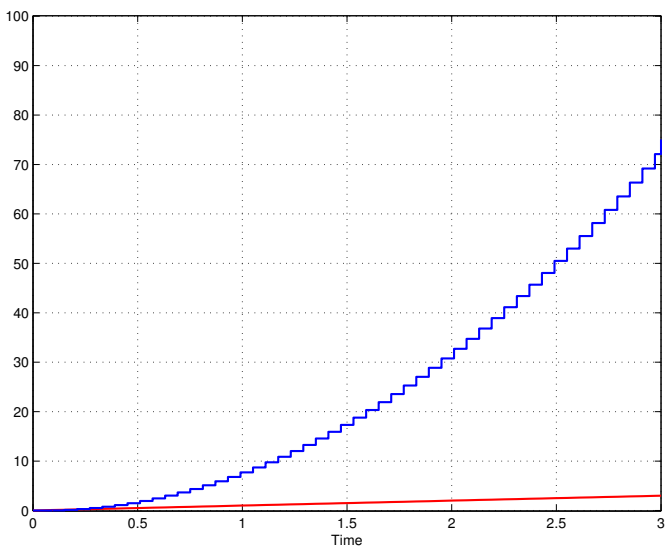
❶ Which compositions make sense?

Examples in Simulink

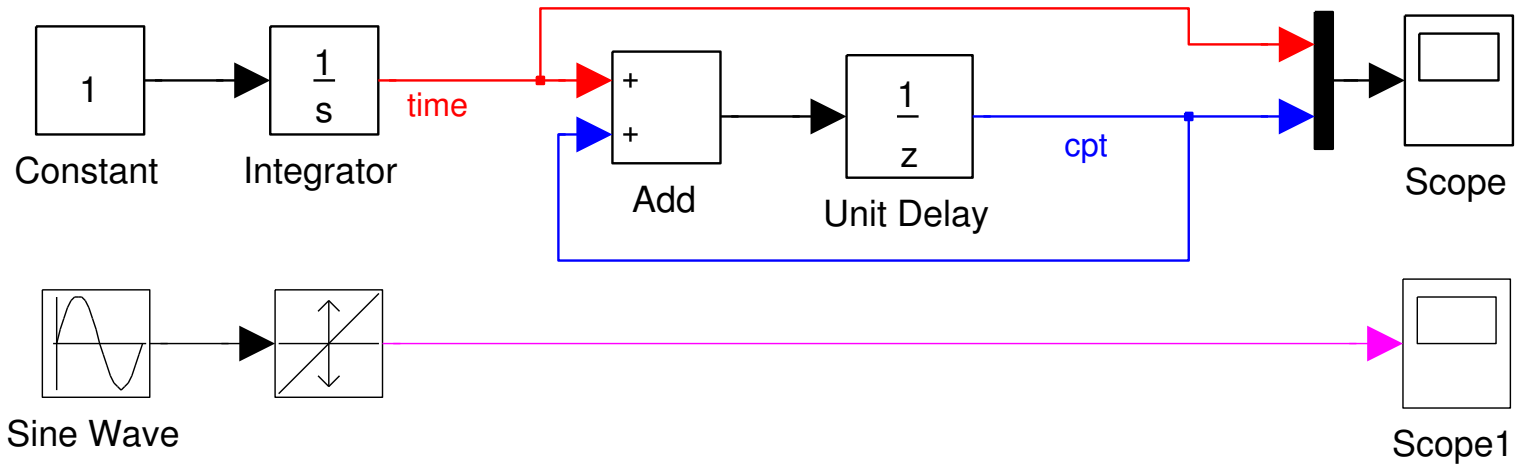
Typing issue 1: Mixing continuous & discrete components



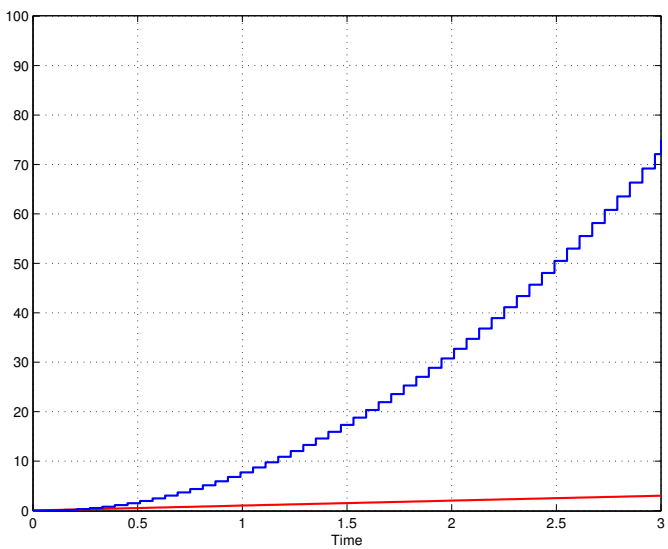
Basic model



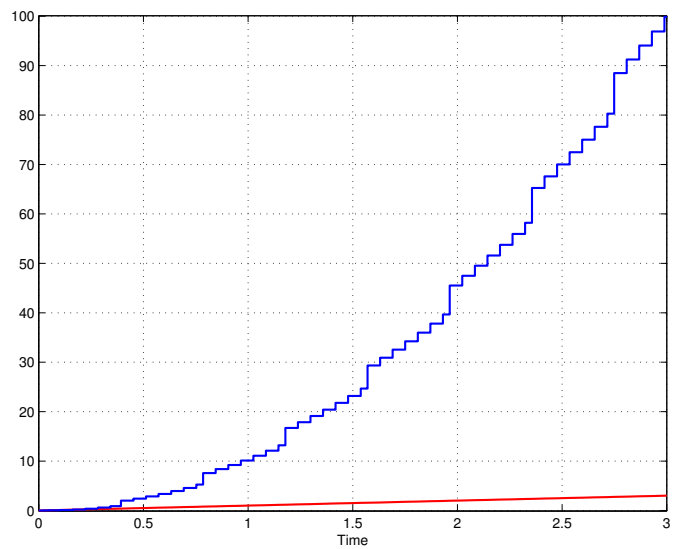
Typing issue 1: Mixing continuous & discrete components



Basic model

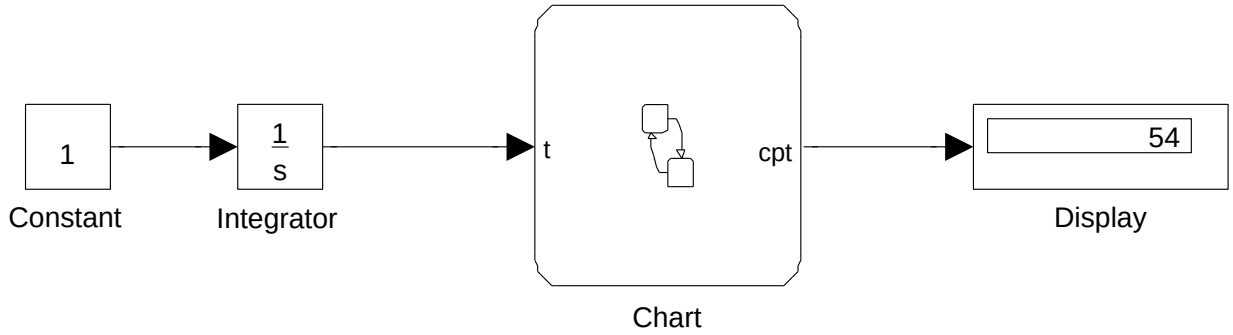


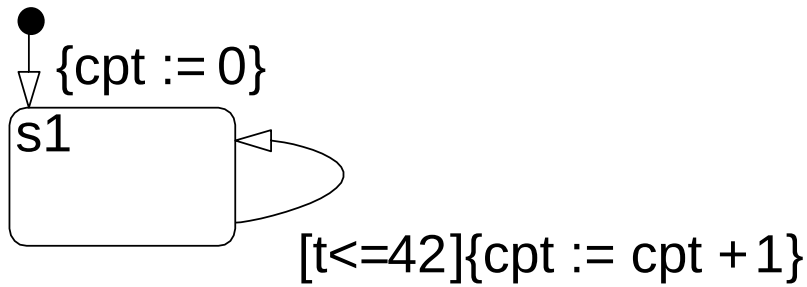
with Sine Wave



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

Typing issue 2: Boolean guards in continuous automata





How long is a discrete step?

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

Here discrete time is not logical
—it comes from the simulation engine.

❸ How should causality (loops) be handled?

How should causality (loops) be handled? [BBC+14]

Some programs are well typed but have algebraic loops.

Which programs should we accept?

- OK to reject (no solution).

  ```
  let hybrid f () = x where rec x = x + 1
  ```

- OK as an algebraic constraint (e.g., Simulink and Modelica).

  ```
  rec x = 1 - x
  ```

  But NOK for sequential code generation.

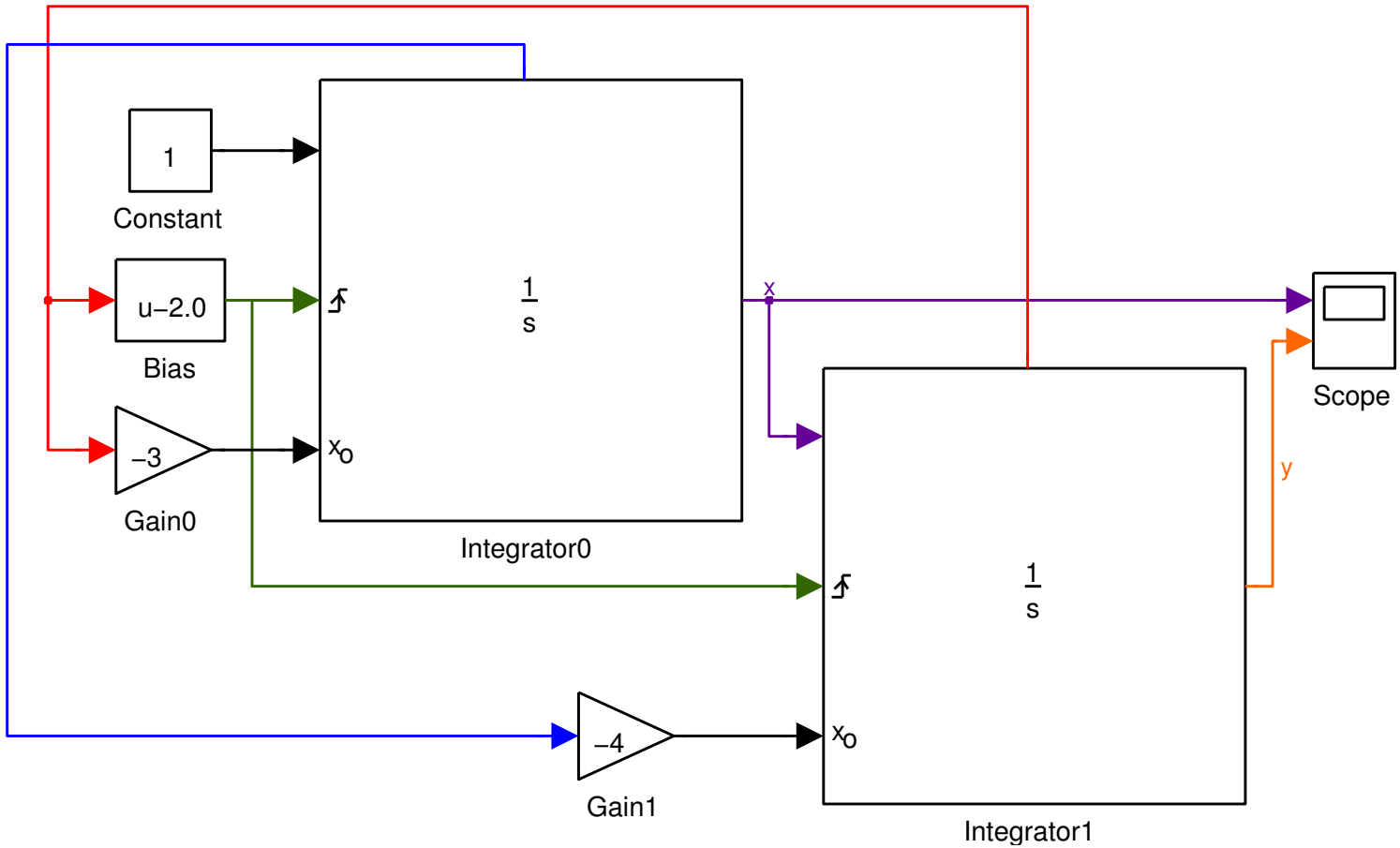
- We need an way to refer to the previous value of a mixed signal. E.g., the following is not causally correct.

  ```
  let hybrid ball (y0, y'0) = y where
    rec der y = y' init y0
    and der y' = -. g init y'0 reset up(-y) -> 0.9 *. y'
  ```

How can we check in a simple and uniform way, that every cycle is broken by a unit delay, or an integrator?

Fdedback loops are a source of bugs in compiler. E.g., an example in Simulink.

Causality issue: the Simulink state port



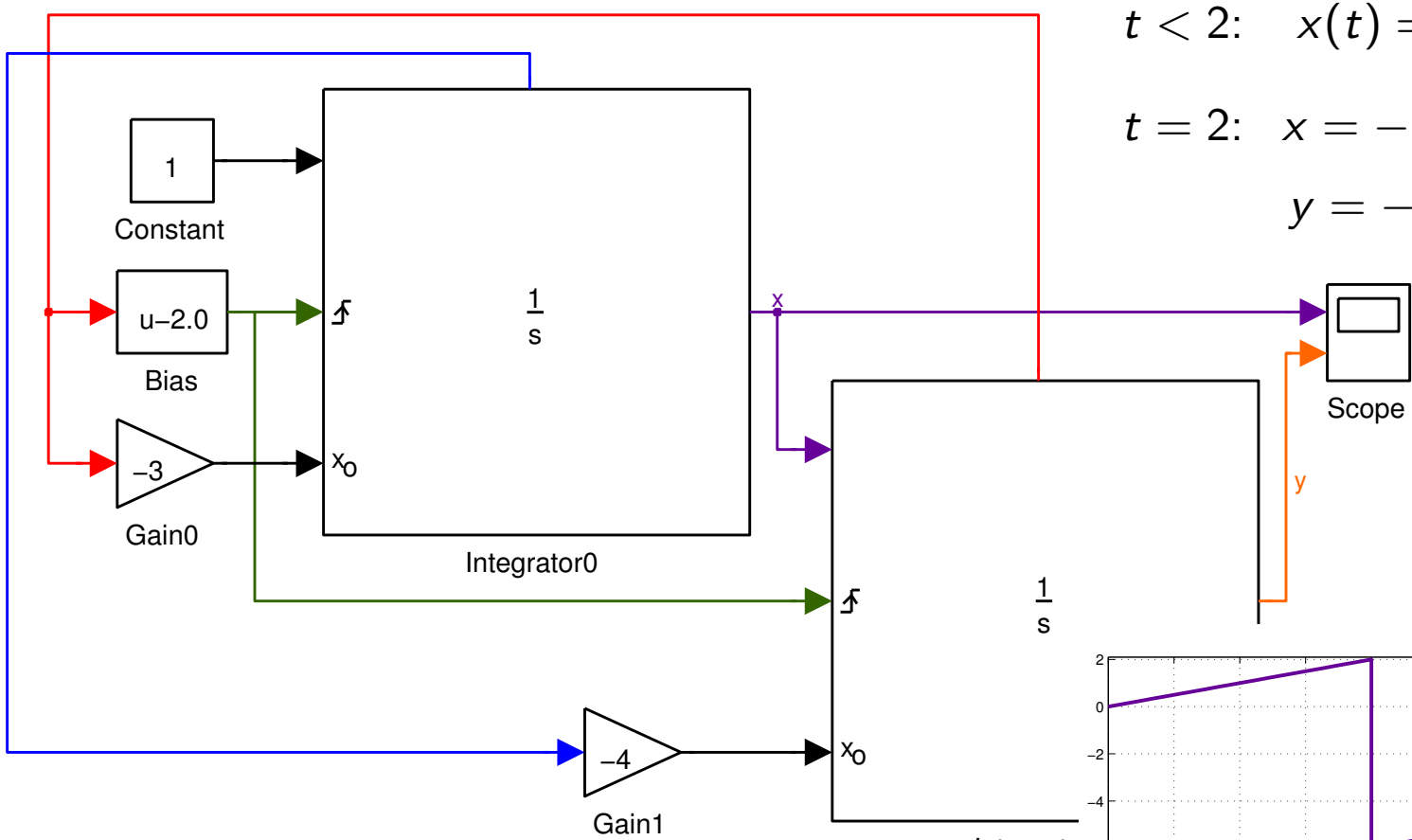
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.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

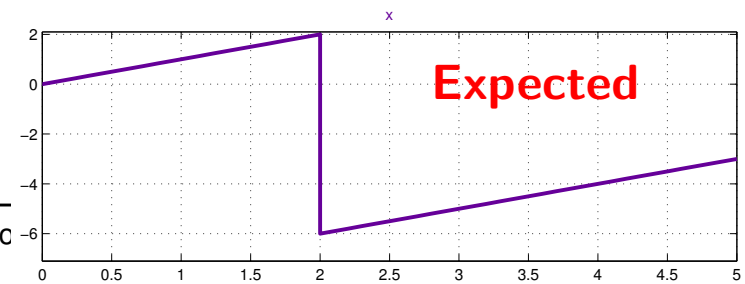
Causality issue: the Simulink state port

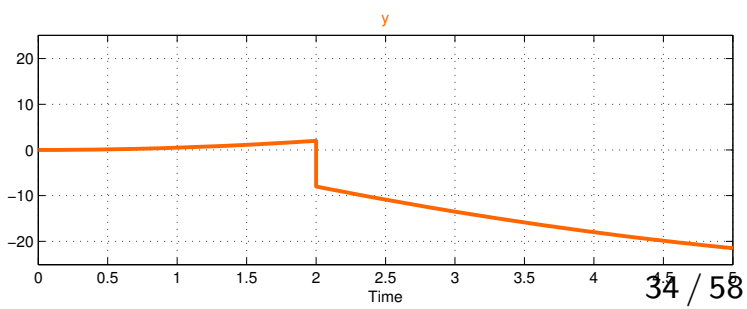
$$t < 2: \quad x(t) = t,\ y(t) = \frac{t^2}{2}$$

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

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







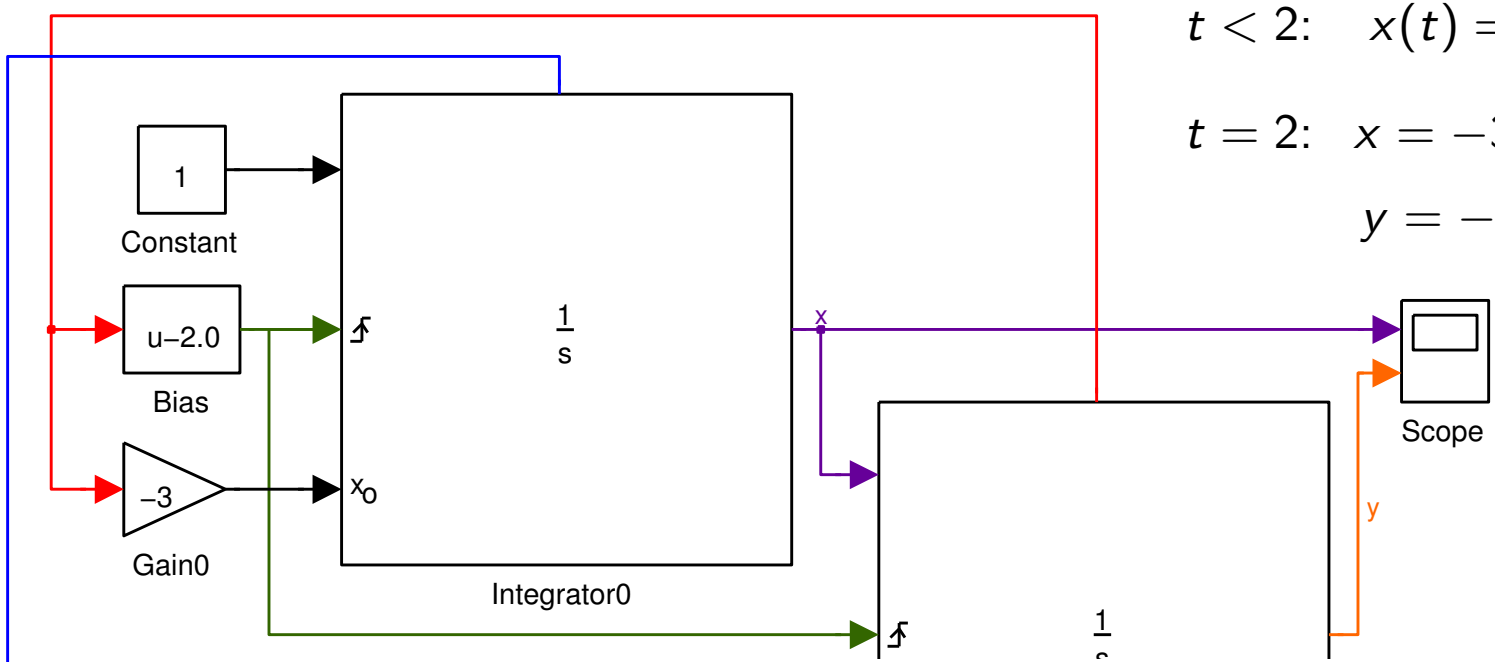
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.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

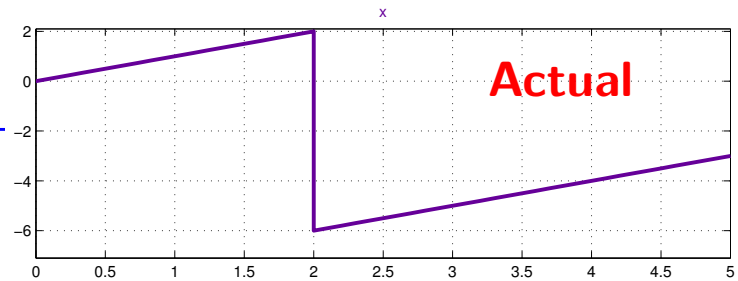
Causality issue: the Simulink state port

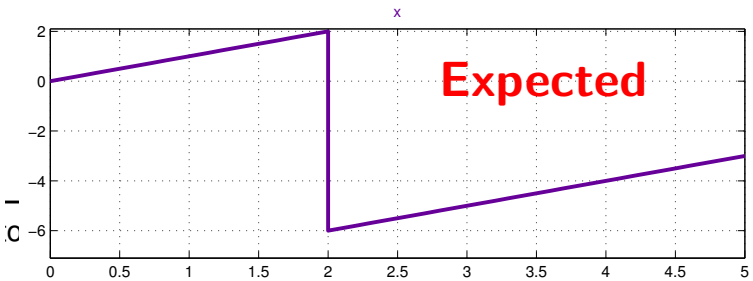
$$t < 2: \quad x(t) = t,\ y(t) = \frac{t^2}{2}$$

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

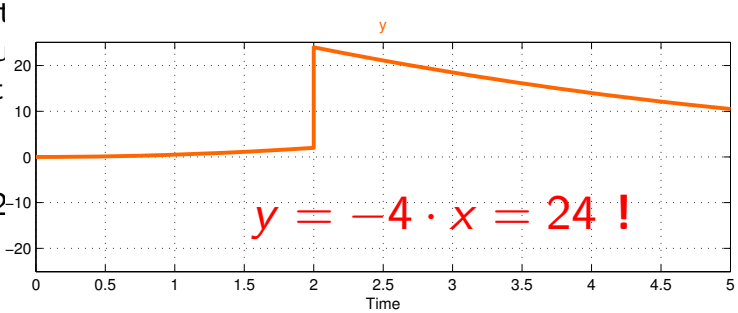
$$y = -4 \cdot \text{last}\, x = -8$$

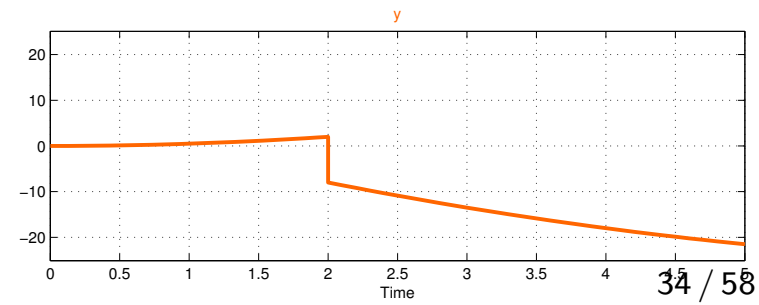






The output of the stat
block's standard outpu
the block is reset in t
state port is the value
standard output if the
–Simulink Reference (2





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;
```

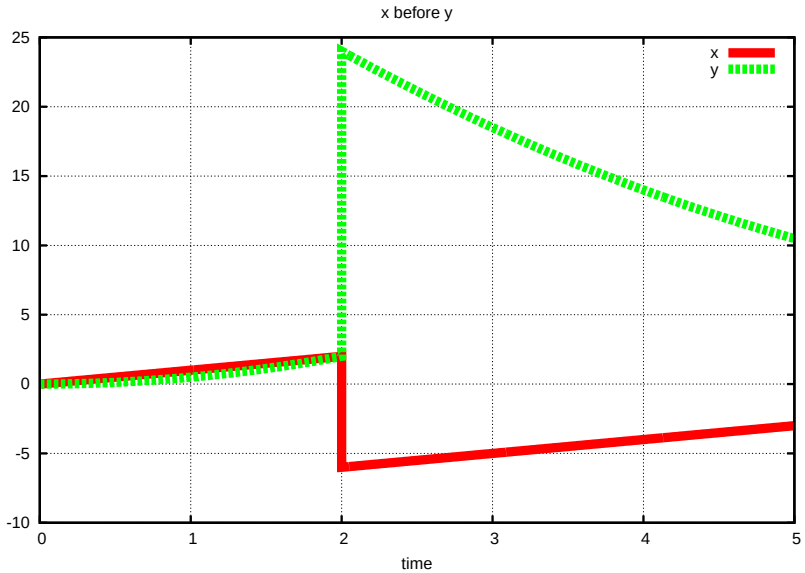
Causality: Modelica example

```
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

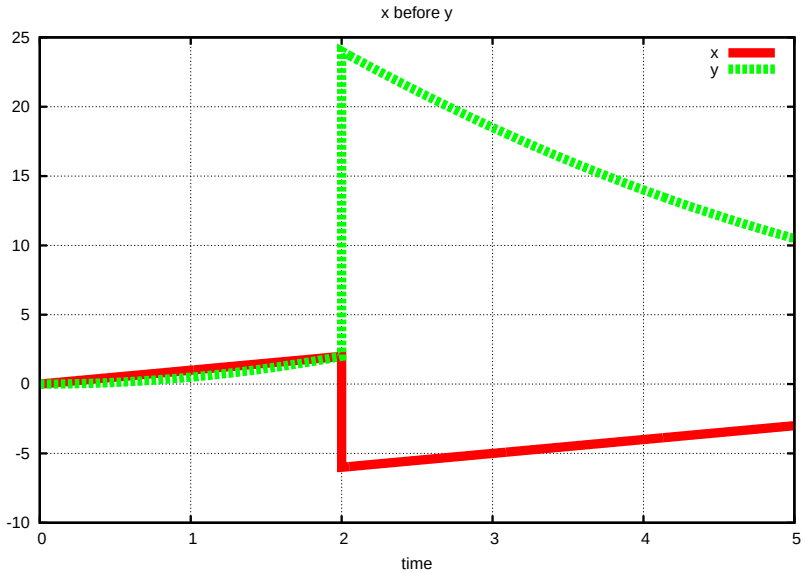
Causality: Modelica example

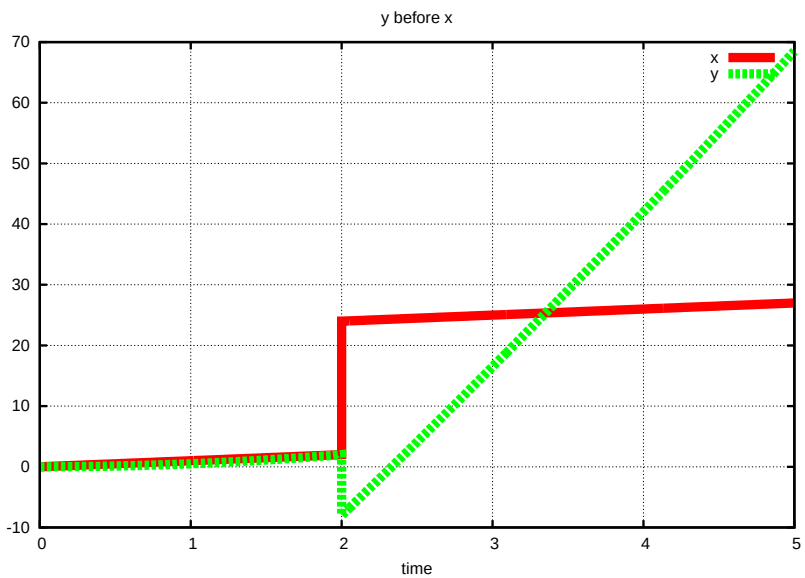
```
model scheduling
  Real x(start = 0);
  Real y(start = 0);
equation

  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 (cont.)

- A causal version (i.e., reinit(x, $-3 * \text{pre } y$)) is scheduled properly (pre y denotes the left limit of y).
  Normally, everything works correctly.

- But, this non-causal program is accepted and the result is not well defined.
  What is the semantics of this program?

- It's not about forbidding algebraic loops, but the expressions here are clocked (discrete-time).
  Should the solver be left to resolve the non-determinism?

- Such problems are not easy to solve, but
  the semantics of a program must not depend on its layout.

- Studying causality can help to understand the detail of interactions between discrete and continuous code.

Which compositions make sense?

Given:

let node sum(x) = cpt **where**
rec cpt = (0.0 **fby** cpt) +. x

Which compositions make sense?

Given:

```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Define:

```
let wrong () = ()
  where rec
    der time = 1.0 init 0.0
    and y = sum (time)
```

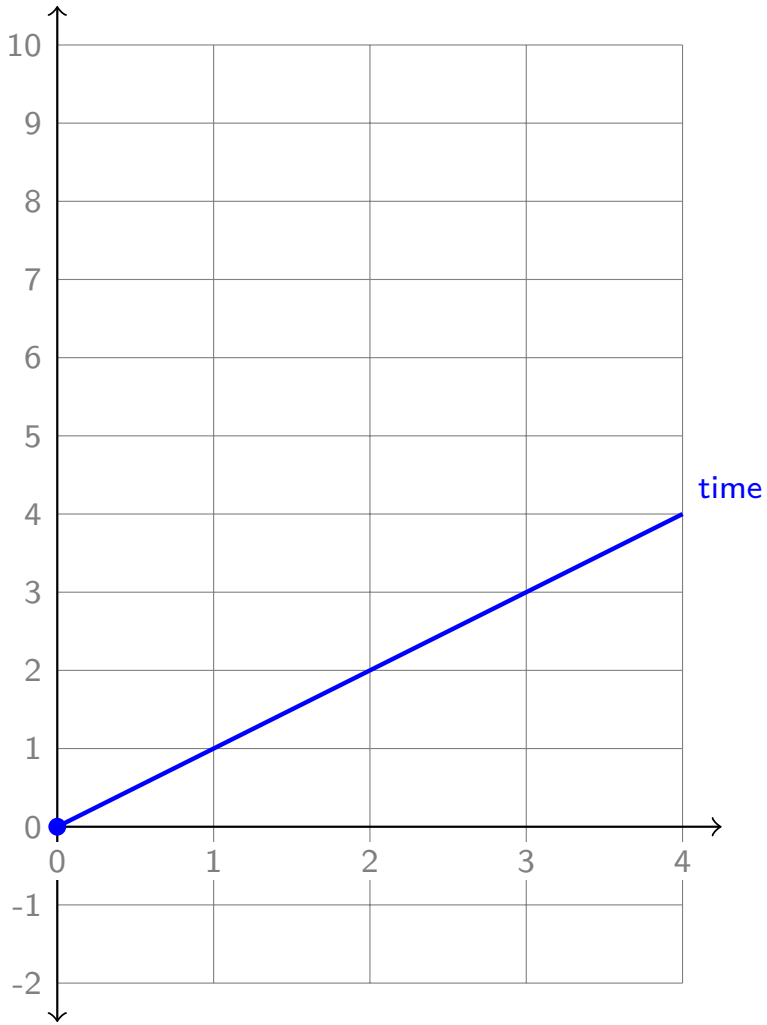
Which compositions make sense?

Given:

```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Define:

```
let wrong () = ()
  where rec
    der time = 1.0 init 0.0
    and y = sum (time)
```



Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject

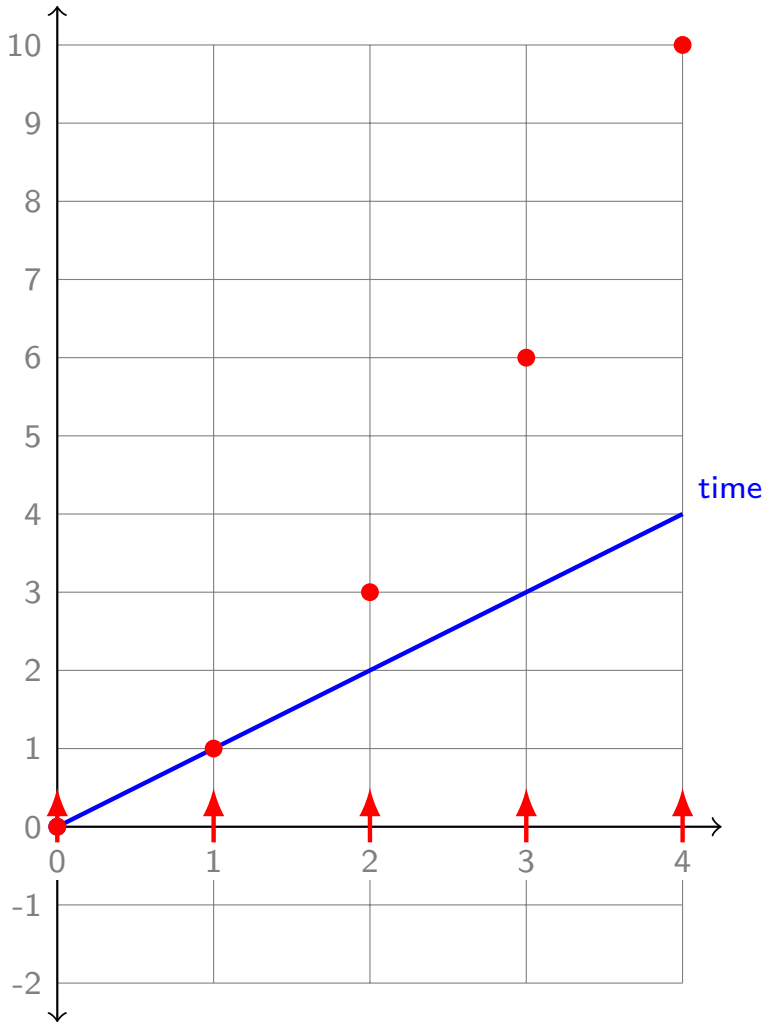
Which compositions make sense?

Given:

```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Define:

```
let wrong () = ()
  where rec
    der time = 1.0 init 0.0
    and y = sum (time)
```



Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject

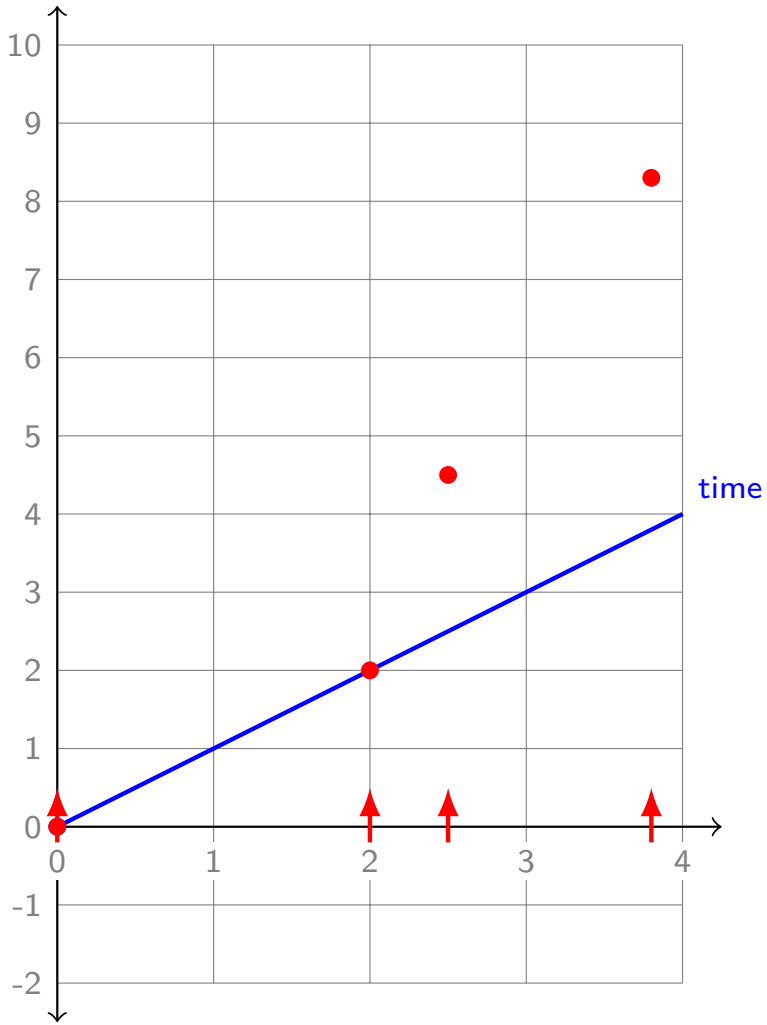
Which compositions make sense?

Given:

```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Define:

```
let wrong () = ()
  where rec
    der time = 1.0 init 0.0
    and y = sum (time)
```



Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject

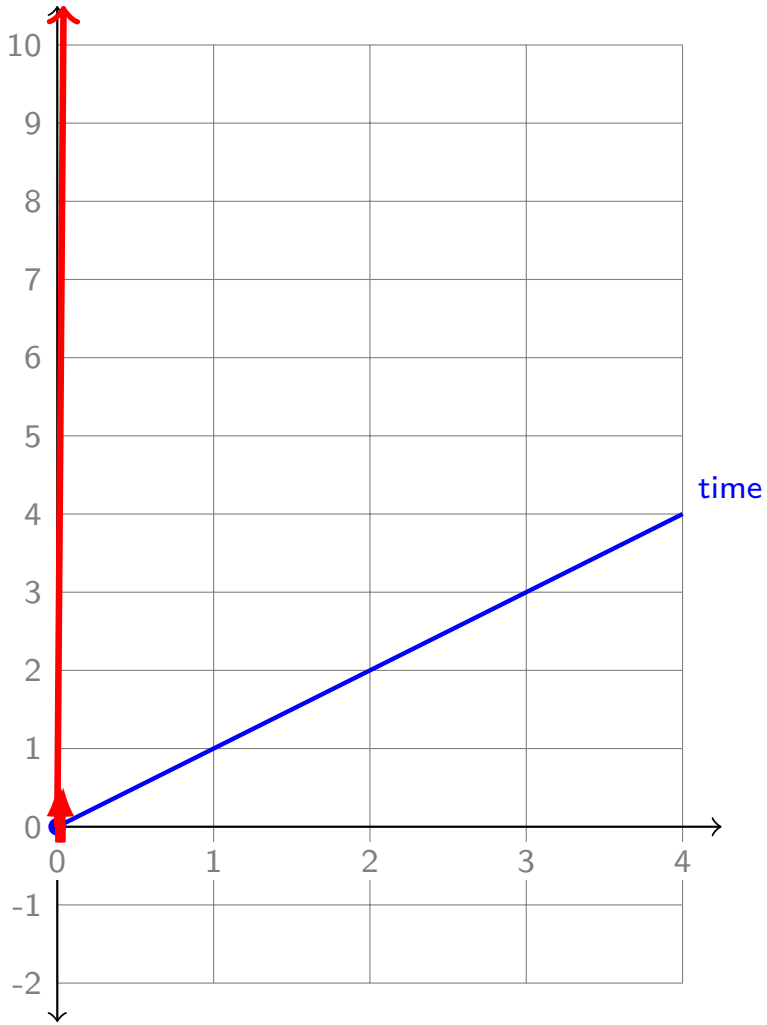
Which compositions make sense?

Given:

```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Define:

```
let wrong () = ()
  where rec
    der time = 1.0 init 0.0
    and y = sum (time)
```



Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject

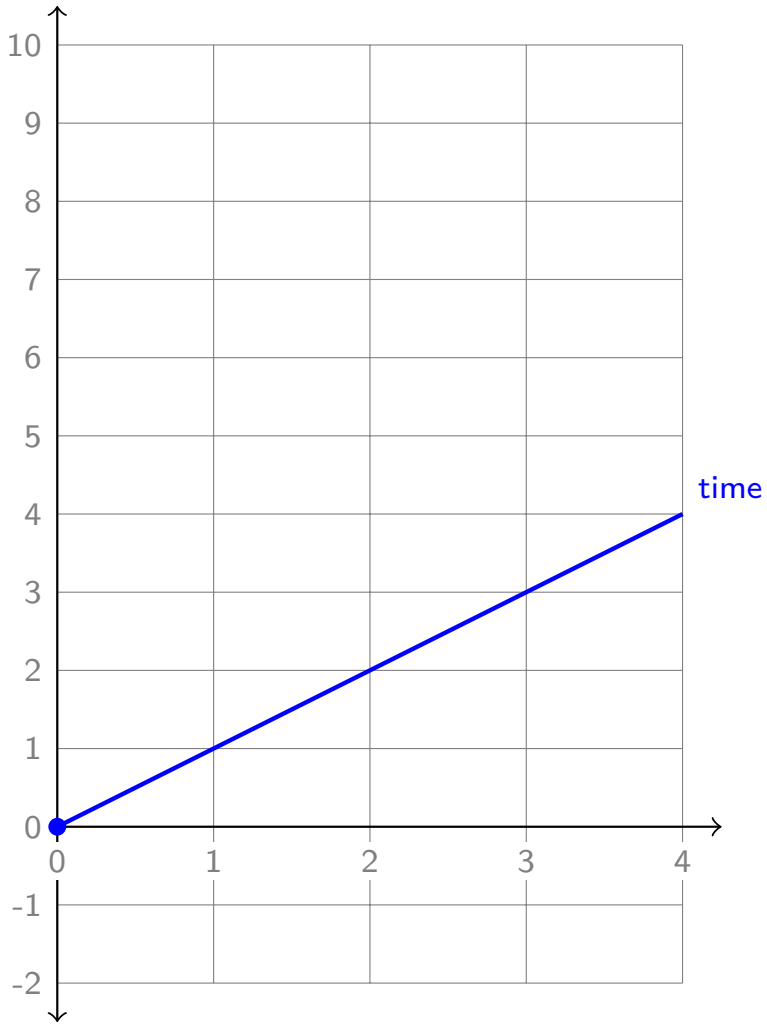
Which compositions make sense?

Given:

```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Define:

```
let wrong () = ()
  where rec
    der time = 1.0 init 0.0
    and y = sum (time)
```



Interpretation:

- Option 1: $\mathbb{N} \subseteq \mathbb{R}$
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject

Which compositions make sense?

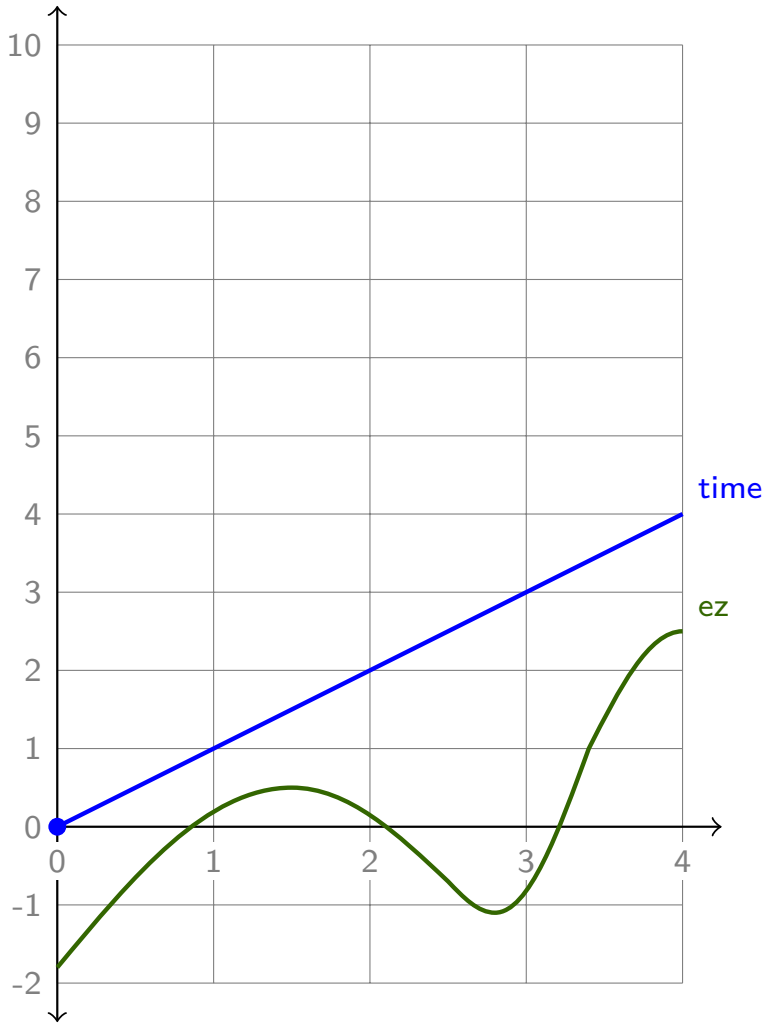
Given:

```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Define:

```
let hybrid correct () = ()
  where rec
    der time = 1.0 init 0.0
    and y = present up(ez) → sum (time)
            init 0.0
```

- node:
  function acting in discrete time
- hybrid:
  function acting in continuous time



Which compositions make sense?

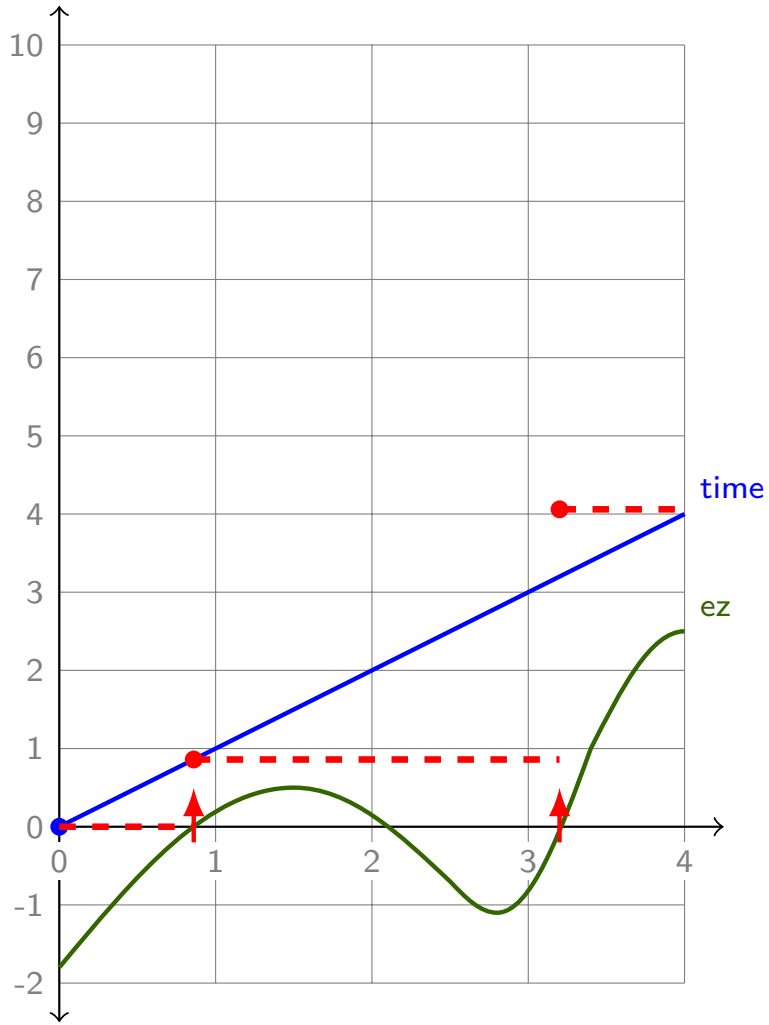
Given:

```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Define:

```
let hybrid correct () = ()
  where rec
    der time = 1.0 init 0.0
    and y = present up(ez) → sum (time)
            init 0.0
```

- node:
  function acting in discrete time
- hybrid:
  function acting in continuous time



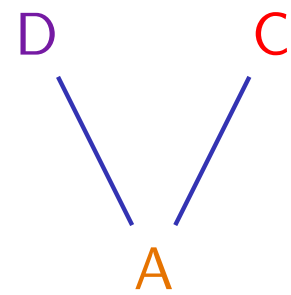
Explicitly relate simulation and logical time (using zero-crossings)

Basic typing [BBCP11a]

A simple ML type system with effects.

The type language

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



Initial conditions

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

Typing rules (extract)

$$\frac{G, H \vdash_{\mathsf{C}} e_1 : \mathsf{float} \qquad G, H \vdash_{\mathsf{C}} e_2 : \mathsf{float} \qquad G, H \vdash h : \mathsf{float}}{G, H \vdash_{\mathsf{C}} \mathsf{der}\ x\ =\ e_1\ \mathsf{init}\ e_2\ \mathsf{reset}\ h : [\mathsf{last}\, x : \mathsf{float}]}\ (\textsc{der})$$

$$\frac{G, H \vdash_k E_1 : H_1 \qquad G, H \vdash_k E_2 : H_2}{G, H \vdash_k E_1\ \mathsf{and}\ E_2 : H_1 + H_2}\ (\textsc{and}) \qquad \frac{G, H \vdash_k e : t}{G, H \vdash_k x\ =\ e : [x : t]}\ (\textsc{eq})$$

$$G, H + [\mathsf{last}\, x : t] \vdash_k \mathsf{last}\, x : t\ (\textsc{last})$$

$$\frac{\forall i \in \{1, .., n\} \qquad G, H \vdash_{\mathsf{C}} z_i : \mathsf{zero} \qquad G, H \vdash_{\mathsf{D}} e_i : t}{G, H \vdash z_1\ \rightarrow\ e_1 \mid \ldots \mid z_n\ \rightarrow\ e_n : t}\ (\textsc{handler})$$

Typing of function body gives its kind $k \in \{\mathsf{C}, \mathsf{D}, \mathsf{A}\}$:

$$h : \mathsf{float} \times \mathsf{float} \xrightarrow{k} \mathsf{float} \times \mathsf{float}$$

Extends naturally to hybrid automata [BBCP11b].

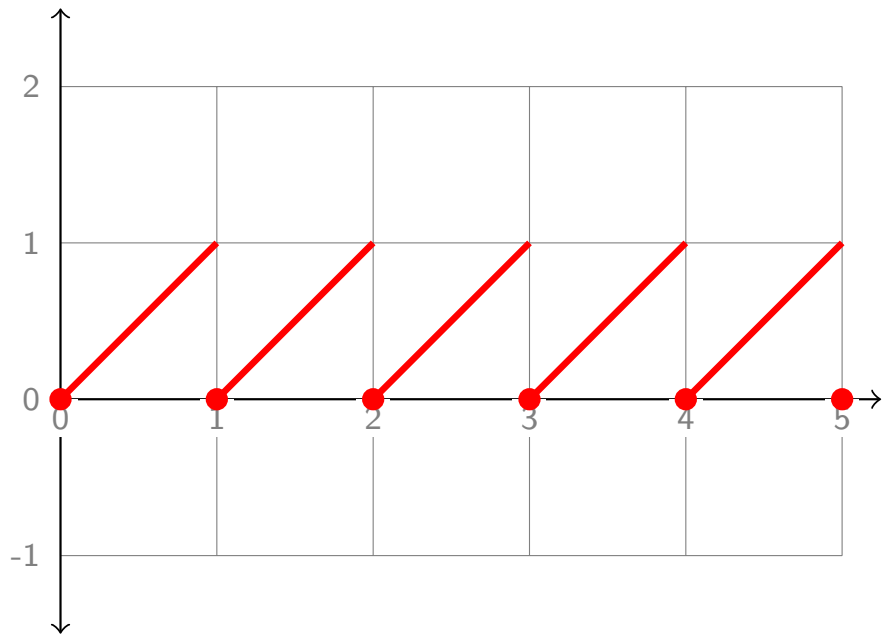
ODEs with reset

Consider the sawtooth signal $y : \mathbb{R}^+ \mapsto \mathbb{R}^+$ such that:

$$\frac{dy}{dt}(t) = 1 \quad y(t) = 0 \text{ if } t \in \mathbb{N}$$

written:

der y $= 1.0$ **init** 0.0 **reset** up(y $-.$ 1.0) $\rightarrow$ 0.0



ODEs with reset

Consider the sawtooth signal $y : \mathbb{R}^+ \mapsto \mathbb{R}^+$ such that:

$$\frac{dy}{dt}(t) = 1 \quad y(t) = 0 \text{ if } t \in \mathbb{N}$$

written:

der y $= 1.0$ **init** 0.0 **reset** up(y $-.$ $1.0) \to 0.0$

The ideal non-standard semantics [BBCP12] is:

$$\begin{aligned}
{}^\star y(0) &= 0 & {}^\star y(n) &= \text{if } {}^\star z(n) \text{ then } 0.0 \text{ else } {}^\star ly(n) \\
{}^\star ly(n) &= {}^\star y(n-1) + \partial & {}^\star c(n) &= ({}^\star y(n) - 1) \geq 0 \\
{}^\star z(0) &= \text{false} & {}^\star z(n) &= {}^\star c(n) \wedge \neg {}^\star c(n-1)
\end{aligned}$$

This set of equation is not causal: ${}^\star y(n)$ depends on itself.

Accessing the left limit of a signal

There are two ways to break this cycle:

- consider that the effect of a zero-crossing is delayed by one cycle, that is, the test is made on ${}^\star z(n-1)$ instead of on $z(n)$, or,

- distinguish the current value of ${}^\star y(n)$ from the value it would have had were there no reset, namely ${}^\star ly(n)$.

Testing a zero-crossing of ly (instead of y),

$$ {}^\star c(n) = ({}^\star ly(n) - 1) \geq 0, $$

gives a program that is causal since ${}^\star y(n)$ no longer depends instantaneously on itself.

```
let hybrid f () = y where rec
  der y = 1.0 init 0.0 reset up(last y -. 1.0) -> 0.0
```

Causality Analysis

Every feedback loop must cross a delay.

Intuition: associate a 'time stamp' to every expression and require that the relation $<$ between time stamps is a strict partial order.

The type language

$$\begin{array}{lll} \sigma & ::= & \forall \alpha_1, ..., \alpha_n : S.\ ct \xrightarrow{k} ct \\ ct & ::= & ct \times ct \mid \alpha \\ k & ::= & \mathsf{D} \mid \mathsf{C} \mid \mathsf{A} \end{array}$$

Precedence relation:

$$S ::= \{\alpha_1 < \alpha_1', ..., \alpha_n < \alpha_n'\}$$

$S \vdash ct_1 < ct_2$ means that ct_1 precedes ct_2 according to S.

The Type System

Type Judgments

$$\frac{}{S \mid G, H \vdash_k e : ct}\ (\text{TYP-EXP}) \qquad \frac{}{S \mid G, H \vdash_k E : H'}\ (\text{TYP-ENV})$$

$$G ::= [\sigma_1/f_1, ..., \sigma_k/f_k] \qquad H ::= [ct_1/x_1, ..., ct_n/x_n]$$

Initial Conditions

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

❹ How to compile programs?

Interacting with a numerical solver

It is not always feasible, nor even possible, to calculate the behavior of a hybrid model analytically.

All major tools thus calculate approximate solutions numerically.

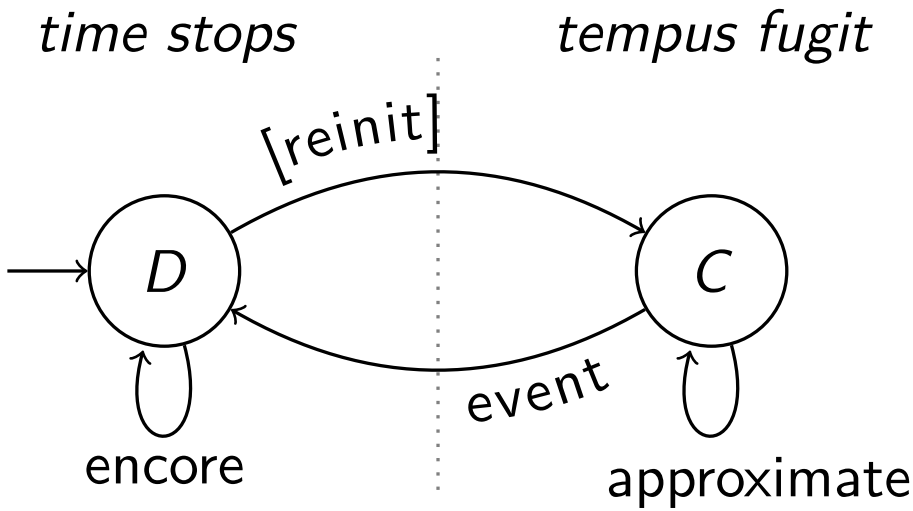
Numerical solvers

- Designed by experts in numerical analysis.
- We treat them as black boxes. We must conform to their interface/restrictions.
- Subtle: must extrapolate continuous dynamics accurately using a finite number of (variable size) discrete steps and floating point numbers.

The Simulation Loop of Hybrid Systems

Define it as a synchronous function that computes four streams:

- $t(n)$ is the increasing sequence of instants where the solver stops
- $lx(n)$ is the value at time $t(n) \in \mathbb{R}$ of the *continuous state variables*
- $y(n)$ is the value at time $t(n)$ of the *discrete state*
- $z(n)$ signals any *zero-crossing* occurring at instant $t(n)$



The Continuous Mode (C)

Abstract the solver as a stream function $solve(f)(g)$ with:

- $x'(\tau) = f(y(n), \tau, x(\tau))$ is the IVP
- $upz(\tau) = g(y(n), \tau, x(\tau))$ defines zero-crossings

The mode C computes the sequences:

$$(lx, z, t, s)(n+1) = solve(f)(g)(s, y, lx, t, step)(n)$$

$s(n)$ is the internal state of the solver at instant $t(n) \in \mathbb{R}$.

$lx(n)$ is the value of x at instant $t(n)$, that is, $lx(n) = x(t(n))$.

$t(n+1)$ is bounded by the horizon $t(n) + step(n)$:

$$t(n) \leq t(n+1) \leq t(n) + step(n)$$

x is a solution of an ODE. $\forall T \in [t(n), t(n+1) + margin]$:

$$x(T) = lx(n) + \int_{t(n)}^{T} f(y(n), \tau, x(\tau))\, d\tau$$

The Discrete Mode (D)

All discrete changes occur in this mode. Execute:

$$(y, lx, step, encore)(n+1) = next(y, lx, z, t)(n)$$

$$z(n+1) = false$$

$$t(n+1) = t(n)$$

with:

$y(n+1)$ is the new discrete state (outside of mode D, $y(n+1) = y(n)$);

$lx(n+1)$ is the new continuous state, which may be directly changed in the discrete mode;

$step(n+1)$ is the new step size;

$encore(n+1)$ is true if an additional discrete step must be performed.

$t(n)$ (the simulation time) is unchanged during a discrete phase.

Compilation

The Compiler has to produce:

1. Inititialization function *init* to define $y(0)$ and $lx(0)$.
2. Functions f and g.
3. Function *next*.

The Runtime System

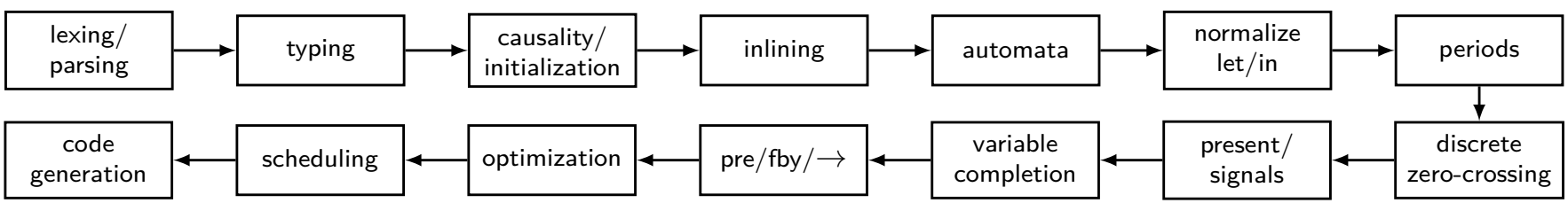
1. Program the simulation loop, using a black-box solver;
2. Or rely on an existing infrastructure.

`Zélus` follows (1); `SCADE Hybrid` follows (2), targetting Simplorer FMIs.

How to compile programs? Two experiments

[BCP+15]

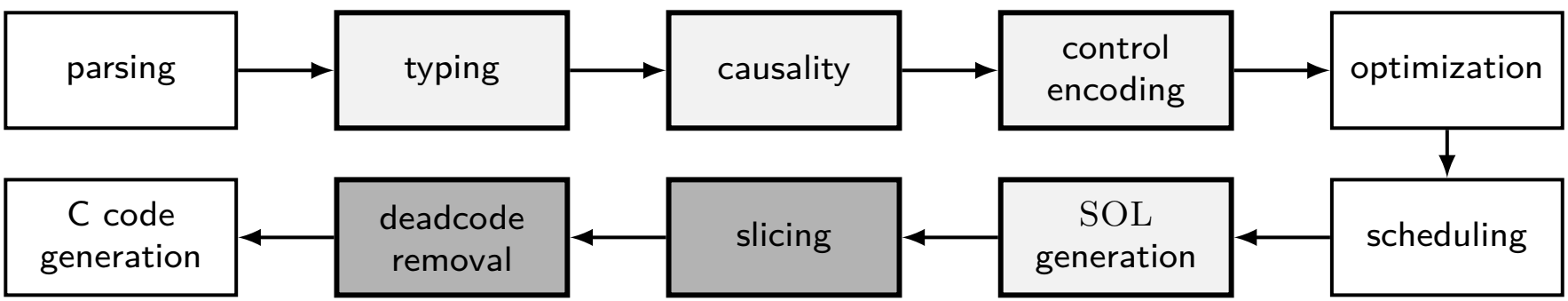
1. Prototype Zélus compiler
   - Simulate with an off-the-shelf, variable-step numerical solver: Sundials CVODE (with OCaml binding).
   - Add source-to-source passes.
   - First version: early removal of ODEs and zero-crossings.
     - Additional inputs and outputs to communicate with the solver.
     - Good for defining and refining the semantics.
     - Not especially efficient.
     - Hard to optimize continuous states and zero-crossings across modes.
   - Second version: later remove of ODEs and zero-crossings.
     - Makes code generation easier.



2. Prototype SCADE Suite KCG Generator
   - ...

Experiment ❷: Prototype SCADE Suite KCG Generator

- A prototype built on KCG 6.4 (release July 2014).
- Generates FMI 1.0 model-exchange FMUs for Simplorer.
- Only 5% of the compiler modified:
  - Small tweaks in static analysis (typing, causality).
  - Small changes in automata translation.
  - Small changes in code generation.
  - FMU generation (XML description, wrapper).
- FMU integration loop: about 1000 LoC.



Compiling ODEs and Zero-crossing

Given a function definition, a synchronous compiler generates a step function. Reuse the existing compiler infrastructure.

- Turn ODEs into data-flow equations.
- Continuous states are distinguished state variables.
- zero-crossing are distinguished state variables.

ODE with Reset: **der** o = x **init** v **reset** z0 → v0 | ...

becomes three equations:

```
last_o = if init then v else o.last
o.val = if z0 then v0
        else if z1 then v1
        ...
        else last_o
o.der = x
```

and the substitutions [last_o/**last**(o)], [o.val/o]

Zero-crossing: z = up(e)

becomes two equations:

z.zout = e
z_event = z.zin **or** (discrete **and** upd(z.zout))

and the substitution [z_event/z]

Finally

1. Translate as usual to produce functions `step` and `reset`.
2. For hybrid nodes, copy-and-paste the step function.
3. Either into a `cont` function activated during the continuous mode, or two functions `derivatives` and `crossings`.
4. Apply the following:
   - During the continuous mode, all variables of type *zero* are surely false.
   - During the discrete step, all continuous state variables are useless.
   - Remove dead-code.
5. That's all.

Examples at: `zelus.di.ens.fr/cc2015`

KCG prototype: Scade + ODEs + Xing

```
const g:real = 9.81;

hybrid bouncing(y0, y_v0:real)
  returns (y:real last = y0)
var  y_v:real last = y_v0;
let
  der y = y_v;
  activate if down y then
    y_v = - 0.8 * last 'y_v;
  else
    der y_v = - g;
  returns ..;
tel
```

```
void bouncing_cont(inC_bouncing *inC,
                   outC_bouncing *outC)
{
  outC->y = outC->der_out.last;
  outC->zc_cb_clock.out = outC->y;
  outC->y_v.der = - g;
  outC->der_out.der = outC->y_v.last;
}
```

```
void bouncing(inC_bouncing *inC,
              outC_bouncing *outC) {
  kcg_real last_y_v;
  kcg_bool cb_clock;

  if (outC->init) {
    outC->init = kcg_false;
    last_y_v = inC->y_v0;
    outC->der_out.val = inC->y0;
  } else {
    last_y_v = outC->y_v.last;
    outC->der_out.val = outC->der_out.last;
  }

  outC->y = outC->der_out.val;
  outC->zc_cb_clock.out = outC->y;

  cb_clock = outC->zc_cb_clock.up | kcg_down(
      outC->zc_cb_clock.last,
      outC->zc_cb_clock.out);

  if (cb_clock) {
    outC->y_v.val = - 0.8 * last_y_v;
  } else {
    outC->y_v.val = last_y_v;
  }
  outC->horizon = kcg_infinity;
}
```

Conclusion

Zelus is a functional data-flow language.

Its discrete-time kernel is essentially Lustre.

It provides important features borrowed from Lucid Synchrone.

E.g., automata, signals, higher-order functions, type inference, parametric polymorphism.

Equations can define discrete-time signals (stream) or continuous-time signals.

The mix of the two must must respect specific typing constraints.

The compilation technique is essentially that of a synchronous compiler.

When the model has ODEs/zero-crossing, the target code is paired with an ODE/zero-crossing detection solver.

Conclusion

Synchronous languages should and can properly treat hybrid systems

To exploit existing compilers and techniques.

To program the discrete subcomponents.

To clarify underlying principles and guide language design/semantics.

Current/further work

Define a more expressive type system, with polymorphism on the sort (k); and the sort put on signals instead of blocks.

Define a more expressive causality analysis.

Mix deterministic and probabilistic constructs (e.g., to model systems with noise and perform inference in the loop) [BMA+20].

Provide means to specify assume/guaranty conditions and prove them. Specify refinement relations (e.g., a continuous-time model versus a discrete-time one).

http://zelus.di.ens.fr

Zélus Download Examples Publications Contact

Compiler

Zélus is a synchronous language extended with Ordinary Differential Equations (ODEs) to model systems with complex interaction between discrete-time and continuous-time dynamics. It shares the basic principles of Lustre with features from Lucid Synchrone (type inference, hierarchical automata, and signals). The compiler is written

Research

Zélus is used to experiment with new techniques for building hybrid modelers like Simulink/Stateflow and Modelica on top of a synchronous language. The language exploits novel techniques for defining the semantics of hybrid modelers, it provides dedicated type systems to ensure the absence of discontinuities during integration and the

References I

Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet.
A type-based analysis of causality loops in hybrid modelers.
In Martin Fränzle and John Lygeros, editors, *Proc. 17th Int. Conf. on Hybrid Systems: Computation and Control (HSCC 2014)*, pages 71–82, Berlin, Germany, April 2014. ACM Press.

Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet.
Divide and recycle: Types and compilation for a hybrid synchronous language.
In Jan Vitek and Bjorn De Sutter, editors, *Proc. 2011 ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES '11)*, pages 61–70, Chicago, USA, April 2011. ACM Press.

Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet.
A hybrid synchronous language with hierarchical automata: Static typing and translation to synchronous code.
In *Proc. 11th ACM Int. Conf. on Embedded Software (EMSOFT'11)*, pages 137–147, Taipei, Taiwan, October 2011. ACM Press.

References II

Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet.
Non-standard semantics of hybrid systems modelers.
J. Computer and System Sciences, 78(3):877–910, May 2012.

Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg.
A Formally Verified Compiler for Lustre.
In *International Conference on Programming Language, Design and Implementation (PLDI)*, Barcelona, Spain, June 19-21 2017. ACM.

Timothy Bourke, Lélio Brun, and Marc Pouzet.
Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset.
In *International Conference on Principles of Programming Languages (POPL)*, New Orleans, Louisiana, United States, January 19-25 2020. ACM.

Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet.
A synchronous-based code generator for explicit hybrid systems languages.
In *Proc. 24th Int. Conf. on Compiler Construction (CC)*, London, UK, April 2015.

References III

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.

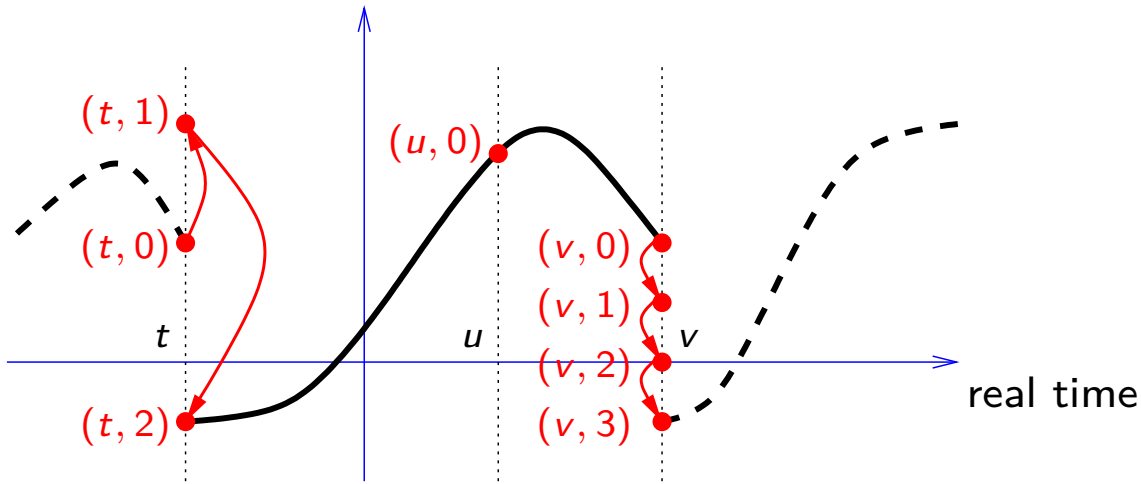
Timothy Bourke and Marc Pouzet.
Zélus: A synchronous language with ODEs.
In Calin Belta and Franjo Ivancic, editors, *Proc. 16th Int. Conf. on Hybrid Systems: Computation and Control (HSCC 2013)*, pages 113–118, Philadelphia, USA, April 2013. ACM Press.

Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet.
Scade 6: A Formal Language for Embedded Critical Software Development.
In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15 2017.

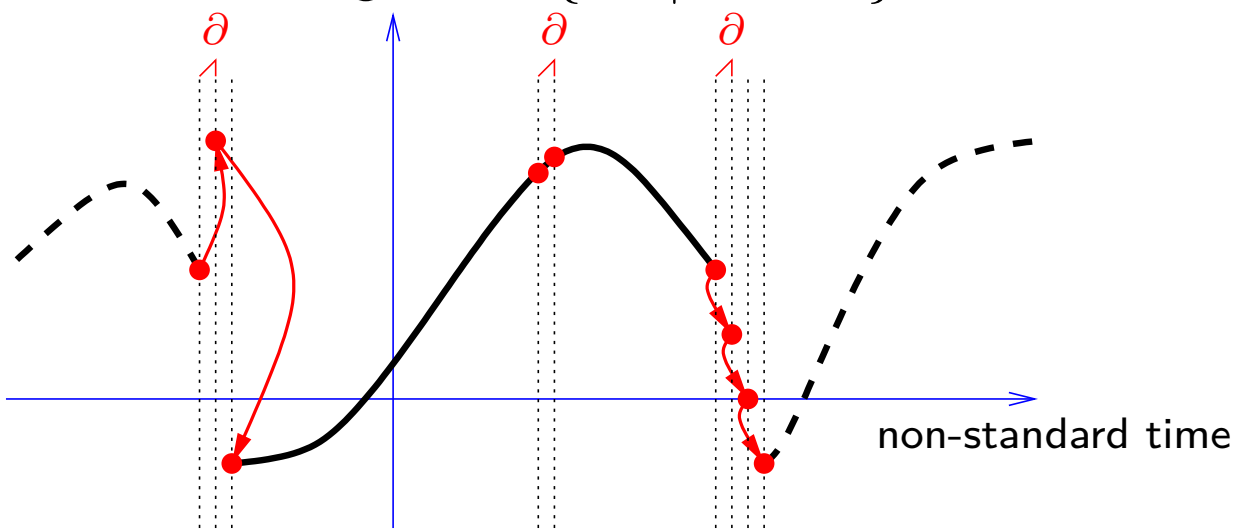
❷ What do compositions mean?

❷ What do compositions mean?

- super-dense time modeling $\mathbb{R} \times \mathbb{N}$



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



Non standard semantics [BBCP12]

- Let ${}^\star\mathbb{R}$ and ${}^\star\mathbb{N}$ be the non-standard extensions of $\mathbb{R}$ and $\mathbb{N}$.
- Let $\partial \in {}^\star\mathbb{R}$ be an infinitesimal, i.e., $\partial > 0, \partial \approx 0$.
- Let the global time base or base clock be the infinite set of instants:

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

  $\mathbb{T}_\partial$ inherits its total order from ${}^\star\mathbb{N}$. 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.
- A signal is a partial function from $\mathbb{T}$ to a set of values.

Semantics of basic operations

- Replay the classical semantics of a synchronous language.
- An ODE activated on clock T: **der** x = e **init** e0 **reset** z1 $\rightarrow$ e1

$$^\star x(t_0) = {}^\star e_0(0) \quad \textit{if } t_0 = \min T$$
$$^\star x(t) = \text{if } {}^\star z(t) \text{ then } {}^\star e_1(t) \text{ else } {}^\star x(^\bullet T(t)) + \partial \cdot {}^\star e(^\bullet T(t)) \quad \textit{if } t \in T$$

- **last** x if x is defined on clock T

$$^\star \text{last}\, x(t) = {}^\star x(^\bullet T(t))$$

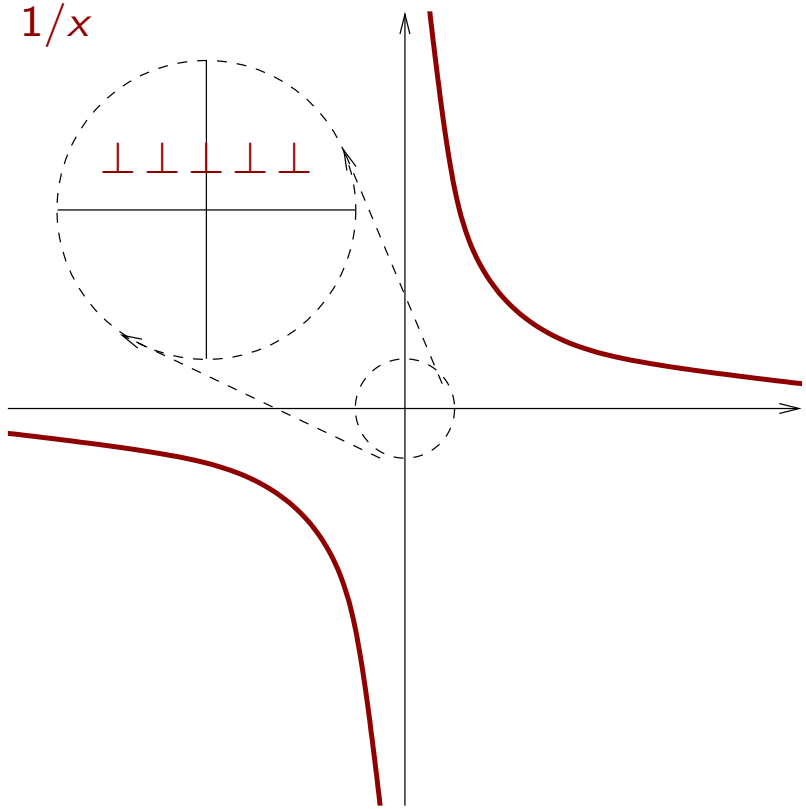
- Zero-crossing up(x) on clock T

$$^\star \text{up}(x)(t_0) = \text{false if } t_0 = \min T$$
$$^\star \text{up}(x)(t) = ({}^\star x(^\bullet T(t) < 0) \wedge ({}^\star x(t) \geq 0) \text{ if } t \in T$$

Assumption on operators and external functions

Assumption 1

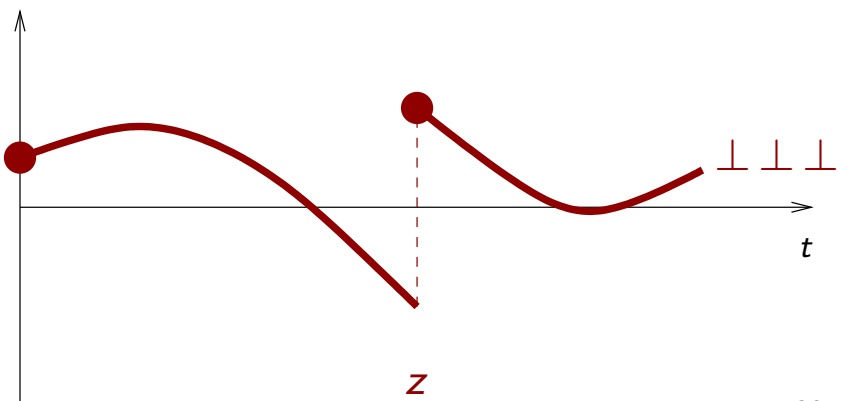
The semantics of operators of kind C or A is undefined ($\perp$) on every infinitesimal neighborhood of a discontinuity or singularity



Assumption 2

For every compact set of dates $K \subseteq \mathbb{T}$, and every input $u : S(^\star V)$, continuous and without zero-crossing on K, the semantics of every external function of kind C or A is either:

1. continuous on K or,
2. triggers a zero-crossing at some $t \in K$ or,
3. is undefined ($\perp$) at some $t \in K$



Main Theorem [BBC+14]

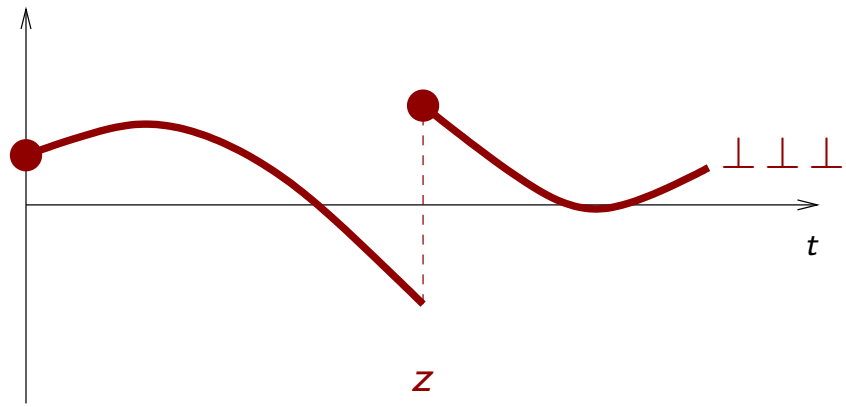
Theorem

The semantics of every causally correct programm is:

1. standardizable,
2. independent of ∂,
3. continuous

on every compact set of dates not containing:

1. a zero crossing, or
2. an undefined value ($\bot$)



The proof deeply rely on the use of the non-standard synchronous semantics.