A synchronous approach to designing and compiling hybrid modelling languages

Timothy Bourke[1,2]

1. Inria Paris-Rocquencourt

2. École normale supérieure (DI)

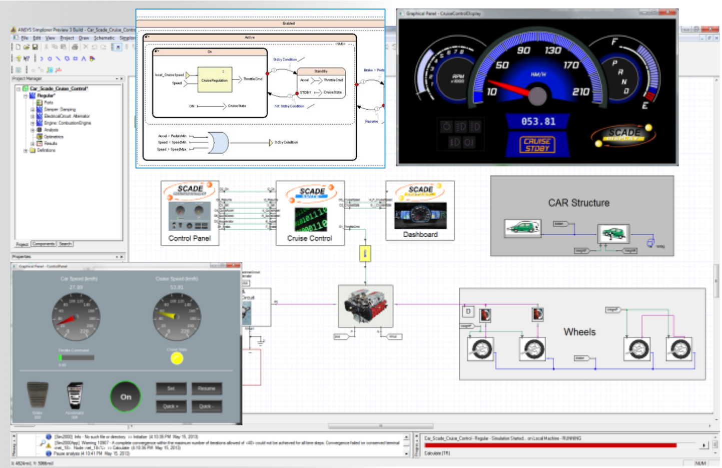
Joint work with Albert Benveniste, Benoît Caillaud, and Marc Pouzet
In collaboration with Jean-Louis Colaço, Bruno Pagano, and Cédric Pasteur

6 February 2015, Chalmers FP Talk

Hybrid Modelling Languages

Embedded software interacts with physical devices.

The whole system has to be modeled: the controller and the plant.[1]



[1] Image by Esterel-Technologies/ANSYS.

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?

Approach: add ODEs to a synchronous dataflow language

Reuse existing tools 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.

Dataflow programming: Lustre/SCADE

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

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

Dataflow programming: Lustre/SCADE

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

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

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 : $int \times bool \xrightarrow{D} bool$

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.

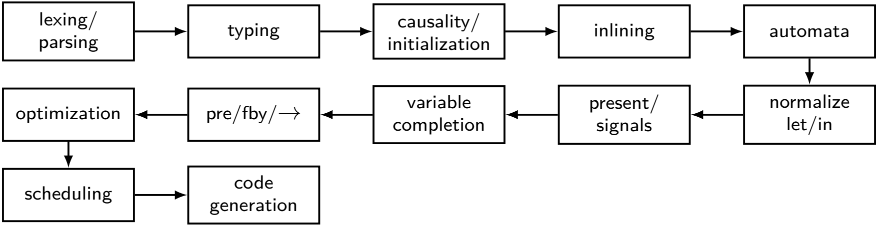
let node f x = y **where**
  rec y = p + x
  and p = y + 1

```
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 a an example of a cycle:[p --> y y --> p p --> p]
```

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).



Discrete programs

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

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

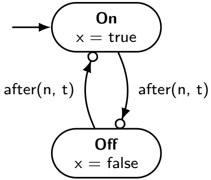
Discrete programs

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

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

```
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
```

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



Alarm

Done

Cancelled

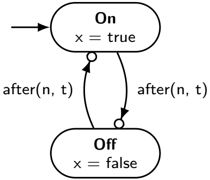
Discrete programs

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

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

```
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
```

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



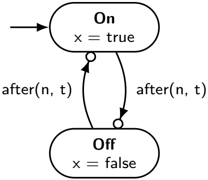
Discrete programs

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

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

```
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
```

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




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

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

Support for (some) 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.

ODEs: still functional, still SSA, enough problems for us initially.

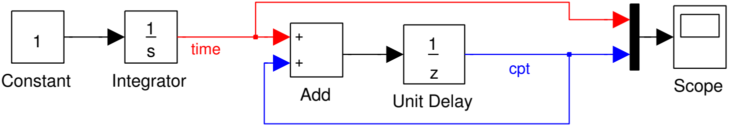
Immediately raises several interrelated questions

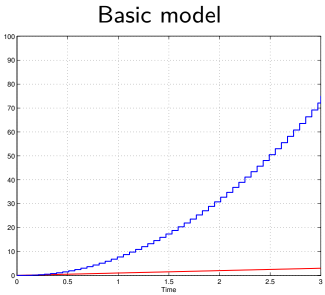
1. Which compositions make sense?
2. What do they mean?
3. How should causality (loops) be handled?
4. How to compile programs?

We study them in a prototype language (Zélus, [BP13]) and by looking at existing languages (Simulink and Modelica).

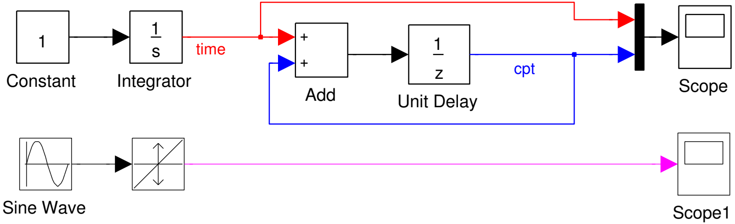
❶ Which compositions make sense?

Typing issue 1: Mixing continuous & discrete components

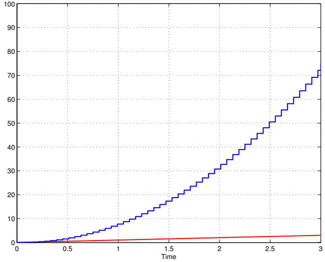




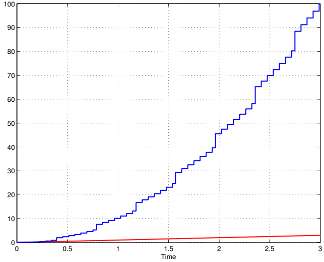
Typing issue 1: Mixing continuous & discrete components



Basic model

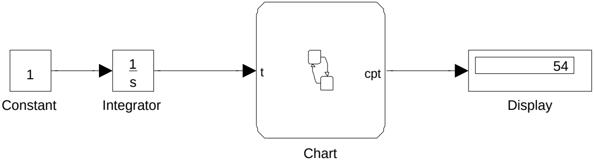


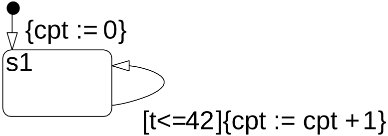
with Sine Wave



- 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.

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)
```

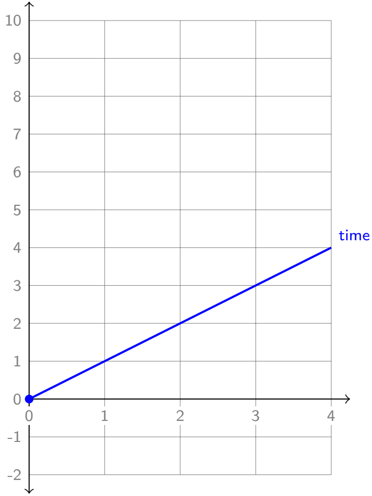
Which compositions make sense?

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

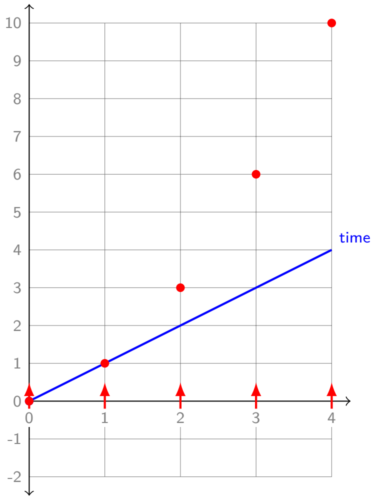
Which compositions make sense?

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

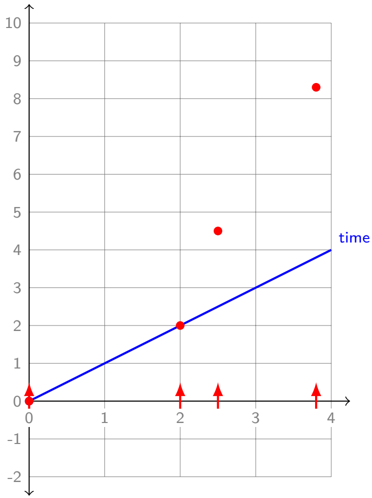
Which compositions make sense?

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

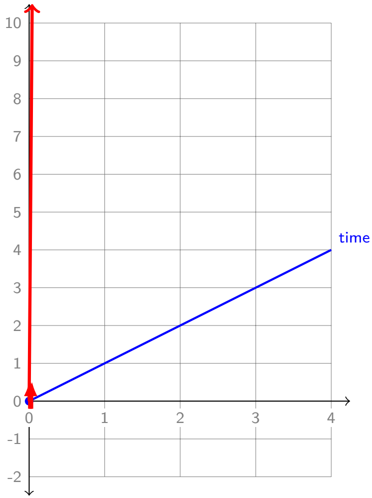
Which compositions make sense?

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

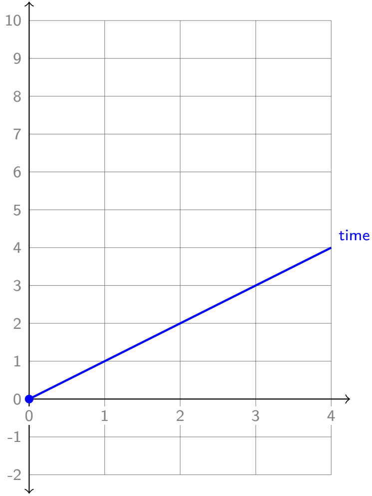
Which compositions make sense?

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?

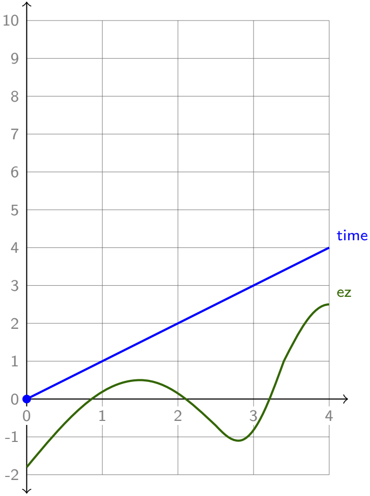
Given:

> **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?

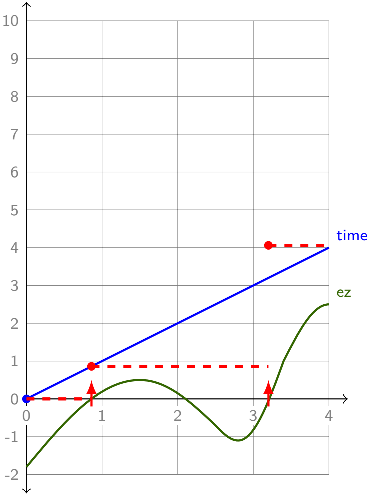
Given:

```
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)

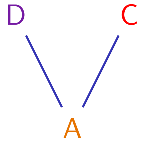
Try to minimize the effects of solver parameters and choices

Basic typing [BBCP11a]

A simple ML type system with effects.

The type language

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



Initial conditions

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

Typing rules (extract)

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

$$\text{(AND)}\ \frac{G, H \vdash_k E_1 : H_1 \qquad G, H \vdash_k E_2 : H_2}{G, H \vdash_k E_1\ \mathtt{and}\ E_2 : H_1 + H_2}$$

$$\text{(EQ)}\ \frac{G, H \vdash_k e : t}{G, H \vdash_k x \ = \ e : [x : t]}$$

$$\text{(EQ-DISCRETE)}\ \frac{G, H \vdash h : t \qquad G, H \vdash_{\mathsf{C}} e : t}{G, H \vdash_{\mathsf{C}} x \ = \ h\ \mathsf{init}\ e : [\mathsf{last}\ x : t]}$$

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

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

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

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

Less expressive but simpler than ‘per-wire’ kinds, e.g. Simulink

$$j : (\mathsf{float}_{\mathsf{D}}) \times (\mathsf{float}_{\mathsf{C}}) \longrightarrow (\mathsf{float}_{\mathsf{D}}) \times (\mathsf{float}_{\mathsf{C}})$$

Extends naturally to hybrid automata [BBCP11b].

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).

  rec x = x + 1

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

  rec x = 1 − x

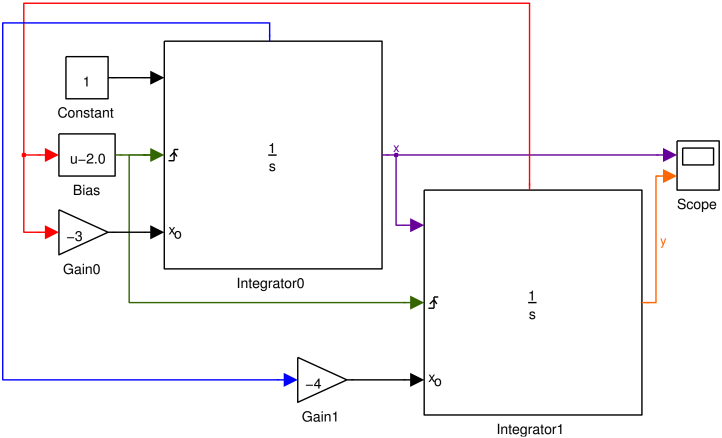
  But NOK for sequential code generation.

- **last** x does not necessarily break causality loops!

  rec x = **last** x + 1

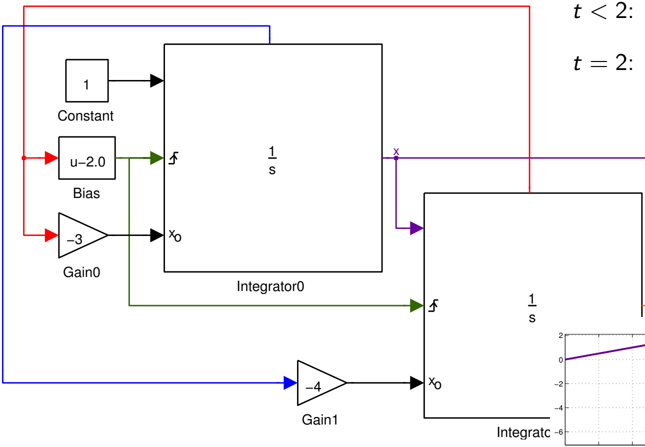
How can we check in a simple and uniform way, that every cycle is broken by a unit delay, or an integrator?

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)

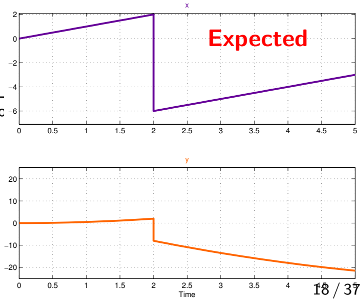
Causality issue: the Simulink state port



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

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

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



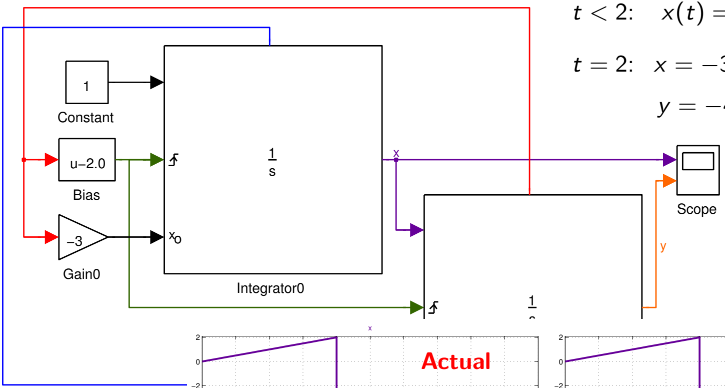
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.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

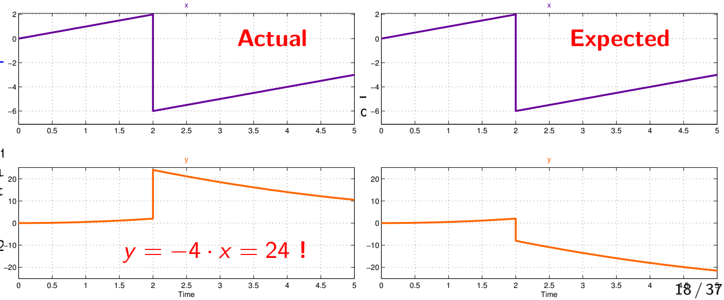
Causality issue: the Simulink state port

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

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

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





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.

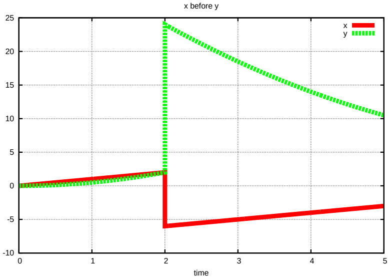
Causality: Modelica example

```
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);
    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);
    reinit(y, -4 * x);
  end when;

end scheduling;
```

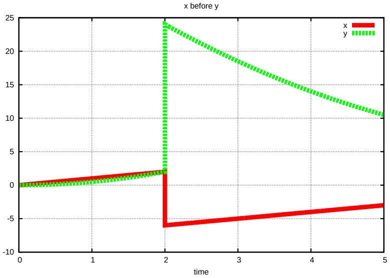


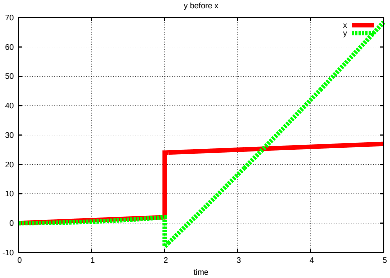
OpenModelica 1.9.2beta1 (r24372)
Also in Dymola

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);
    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, $-3 *$ pre y)) is scheduled properly.
  Normally, everything works correctly.
- But, this non-causal program is accepted and the result is not well defined (it seems to us).
  What is the semantics of this program?
- It's not about forbidding algebraic loops, but the expressions here are clocked (not relational).
  Should the solver be left to resolve the non-determinism?
- Such problems are certainly not easy to solve, but
  the semantics of a model must not depend on its layout!
- Studying causality can help to understand the detail of interactions between discrete and continuous code.

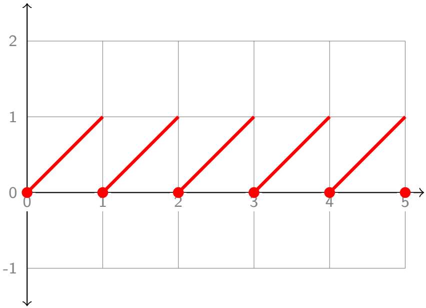
ODEs with reset

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) → 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) → 0.0

The ideal non-standard semantics is:

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

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.

der y = 1.0 **init** 0.0 **reset** up(**last** y −. 1.0) → 0.0

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.

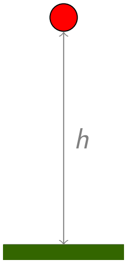
All major tools thus calculate approximate solutions numerically.

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.

Bouncing ball

model



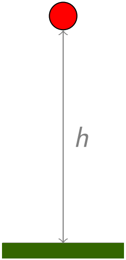
$$F = m \cdot a$$

$$m \cdot -g = m \cdot \frac{d^2 h(t)}{dt^2}$$

$$\frac{d^2 h(t)}{dt^2} = -g$$

Bouncing ball

model



$$F = m \cdot a$$

$$m \cdot -g = m \cdot \frac{d^2 h(t)}{dt^2}$$

$$\frac{d^2 h(t)}{dt^2} = -g$$

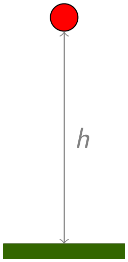
$$\dot{v} = -g \qquad v(0) = v_0$$

$$\dot{h} = v \qquad h(0) = h_0$$

First-order ODE

Bouncing ball

model



$$F = m \cdot a$$

$$m \cdot -g = m \cdot \frac{d^2 h(t)}{dt^2}$$

$$\frac{d^2 h(t)}{dt^2} = -g$$

$$\dot{v} = -g \qquad v(0) = v_0$$

$$\dot{h} = v \qquad h(0) = h_0$$

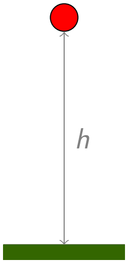
First-order ODE

$$v(t) = v_0 + \int_0^t -g \, .d\tau$$

$$h(t) = h_0 + \int_0^t v(\tau) \, .d\tau$$

Bouncing ball

model



$$F = m \cdot a$$

$$m \cdot -g = m \cdot \frac{d^2 h(t)}{dt^2}$$

$$\frac{d^2 h(t)}{dt^2} = -g$$

$[\dot{v}; \dot{h}] = \mathbf{f}(t, [v; h])$

Solver

approximation

$\mathbf{y_i} = [v_0; h_0]$

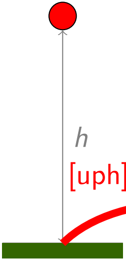
$$\begin{aligned} \dot{v} &= -g \\ \dot{h} &= v \end{aligned} \qquad \begin{aligned} v(0) &= v_0 \\ h(0) &= h_0 \end{aligned}$$

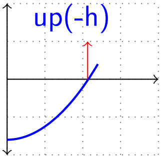
First-order ODE

$$\begin{aligned} v(t) &= v_0 + \int_0^t -g \, .d\tau \\ h(t) &= h_0 + \int_0^t v(\tau) \, .d\tau \end{aligned}$$

Bouncing ball

model





$$F = m \cdot a$$

$$m \cdot -g = m \cdot \frac{d^2 h(t)}{dt^2}$$

$$\frac{d^2 h(t)}{dt^2} = -g$$

$$[\text{uph}] = \mathbf{g}(t, [v; h])$$

$$[\dot{v}; \dot{h}] = \mathbf{f}(t, [v; h])$$

Solver

event!

approximation

$$\mathbf{y_i} = [v_0; h_0]$$

$$\boxed{\begin{matrix} \dot{v} \\ \dot{h} \end{matrix}} \begin{matrix} = -g \\ = v \end{matrix} \qquad \boxed{\begin{matrix} v(0) \\ h(0) \end{matrix}} \begin{matrix} = v_0 \\ = h_0 \end{matrix}$$

$$\boxed{\begin{matrix} v(t) \\ h(t) \end{matrix}} \begin{matrix} = v_0 + \int_0^t -g \,.d\tau \\ = h_0 + \int_0^t v(\tau) \,.d\tau \end{matrix}$$

First-order ODE

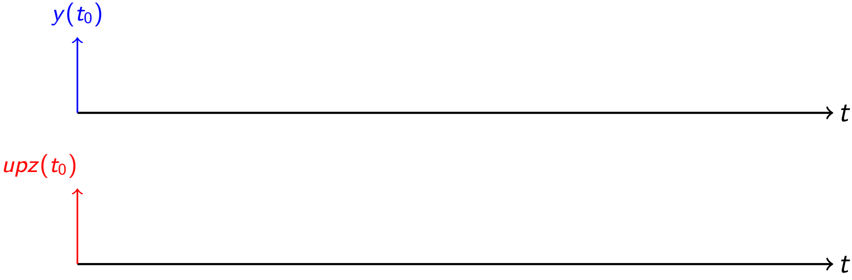
Solver execution (e.g., LLNL Sundials CVODE)

Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$

- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
    - No side-effects within f_σ or g_σ

Solver execution (e.g., LLNL Sundials CVODE)

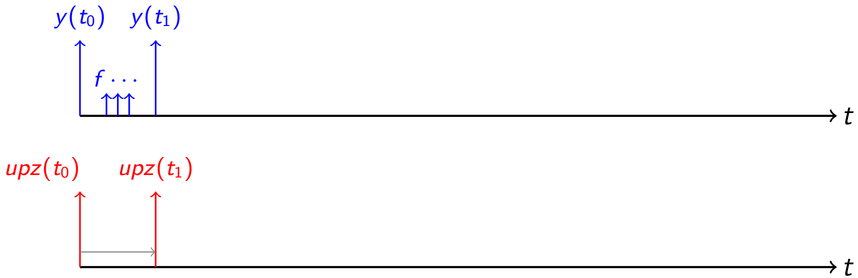
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
    - No side-effects within f_σ or g_σ

Solver execution (e.g., LLNL Sundials CVODE)

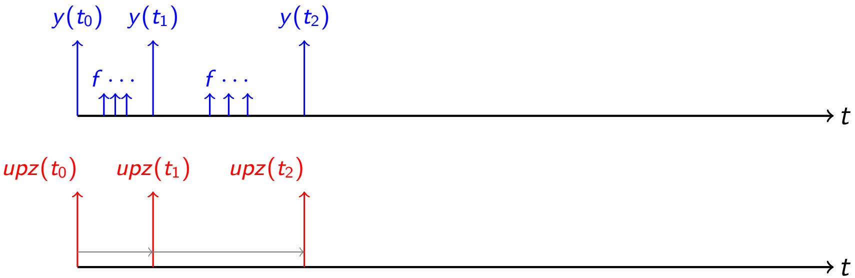
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f_σ or g_σ

Solver execution (e.g., LLNL Sundials CVODE)

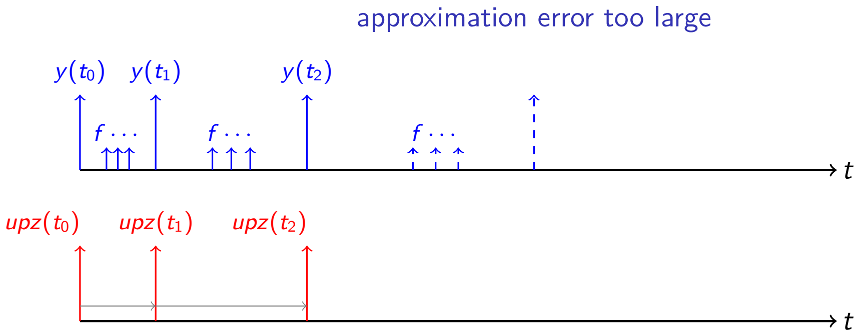
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f_σ or g_σ

Solver execution (e.g., LLNL Sundials CVODE)

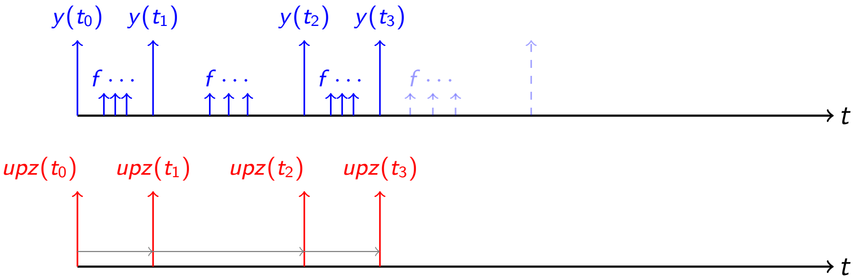
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f_σ or g_σ

Solver execution (e.g., LLNL Sundials CVODE)

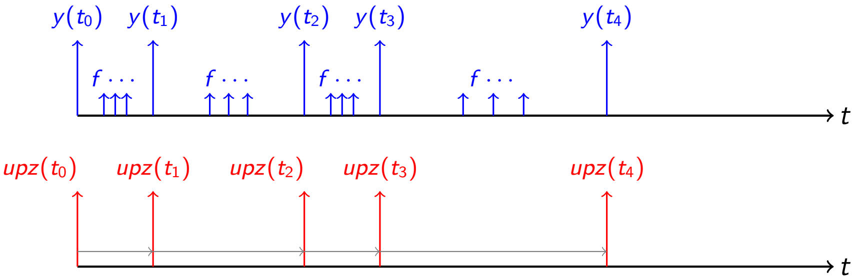
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f_σ or g_σ

Solver execution (e.g., LLNL Sundials CVODE)

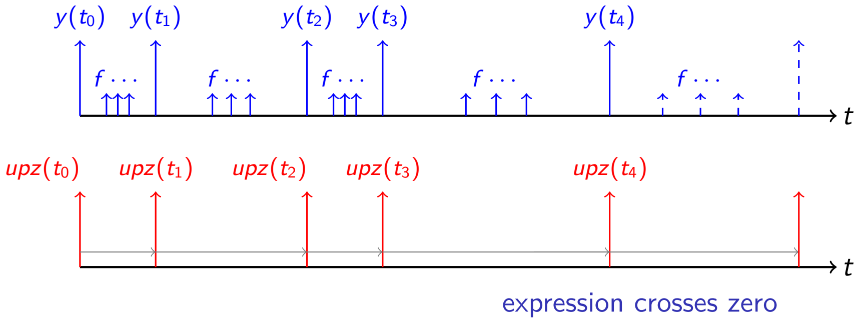
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f_σ or g_σ

Solver execution (e.g., LLNL Sundials CVODE)

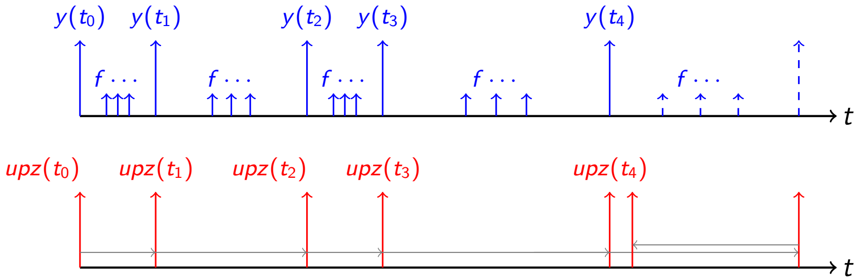
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f_σ or g_σ

Solver execution (e.g., LLNL Sundials CVODE)

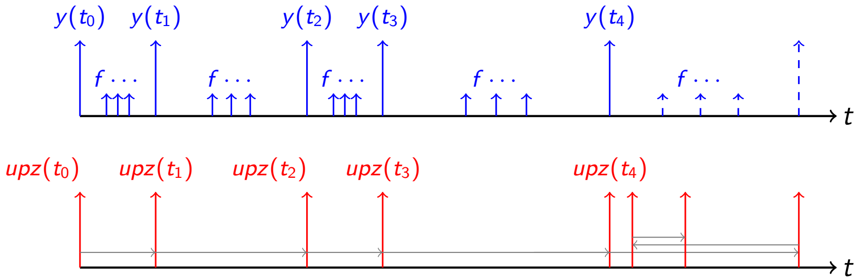
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
    - No side-effects within f_σ or g_σ

Solver execution (e.g., LLNL Sundials CVODE)

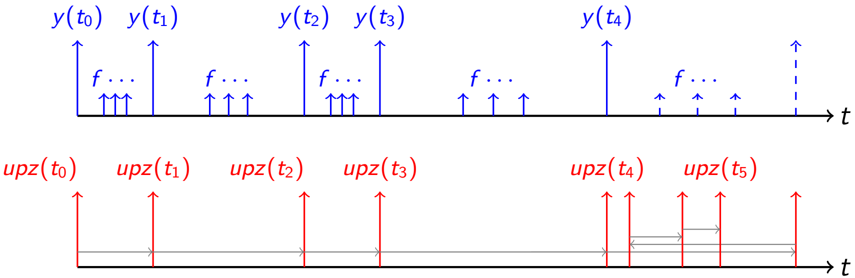
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f_σ or g_σ

Solver execution (e.g., LLNL Sundials CVODE)

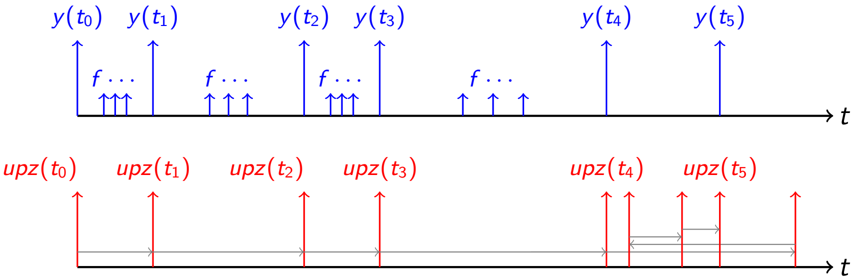
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f_σ or g_σ

Solver execution (e.g., LLNL Sundials CVODE)

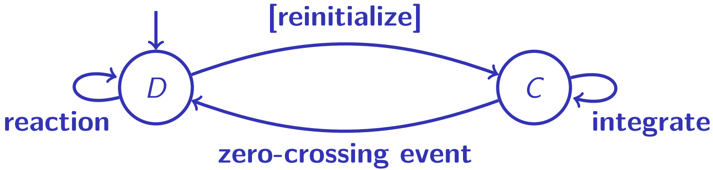
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - No side-effects within f_σ or g_σ

The Simulation Engine of Hybrid Systems

Alternate discrete steps and integration steps



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

Properties of the three functions

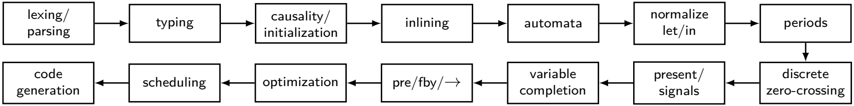
- d_σ gathers all discrete changes.
- g_σ defines signals for zero-crossing detection.
- f_σ and g_σ should be free of side effects and, better, continuous.

How to compile programs? Two experiments

[BCP+15]

❶ 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.

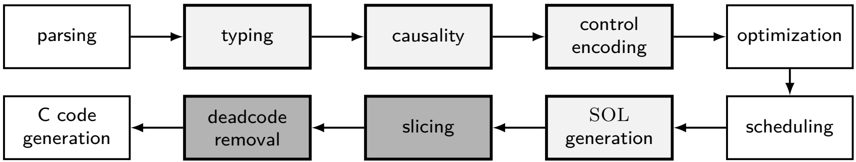


❷ Prototype SCADE Suite KCG Generator

- ...

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

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

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

- 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

- In C after other simplifying transformations.
- Plus code to distribute solver values throughout the 'node hierarchy'.
- Triplicate and apply dead code elimination to get d_σ, f_σ, and g_σ.

KCG prototype: Scade + der

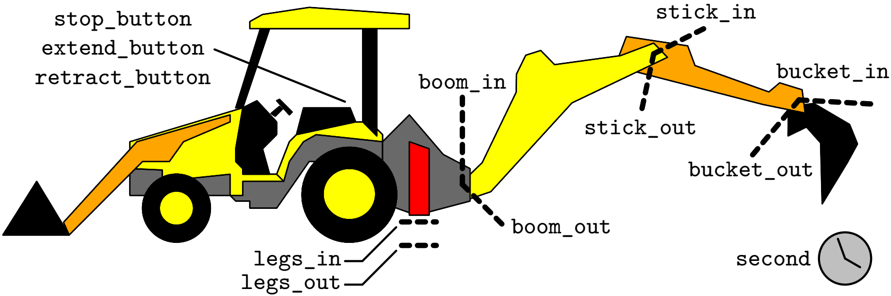
```
const g:real = 9.81;

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;
}
```

Backhoe example: discrete controller + plant model

Sensors (plant outputs / controller inputs)



Actuators (plant inputs / controller outputs)

alarm_lamp	legs_extend	boom_pull	stick_pull	bucket_pull
done_lamp	legs_retract	boom_push	stick_push	bucket_push
cancel_lamp	legs_stop	boom_drive	stick_drive	bucket_drive

(pull = in; push = out)

Zélus: hybrid programs (Plant)

let hybrid integrator(yd, y0) = y **where**
  der y = yd **init** y0

$\textit{\textbf{val} integrator : float} \times \textit{float} \xrightarrow{\texttt{C}} \textit{float}$

Zélus: hybrid programs (Plant)

```
let hybrid integrator(yd, y0) = y where
  der y = yd init y0
```

val integrator : float × float $\xrightarrow{\texttt{C}}$ float

```
let hybrid pi_controller(v_r, i) = angle where
  rec der angle = v init i
  and der v = k_p *. error +. k_i *. z init 0.0
  and der z = error init 0.0
  and error = v_r -. v
```

val pi_controller : float × float $\xrightarrow{\texttt{C}}$ float

$$angle(t) = i + \int_0^t v(\tau)\ .d\tau$$

$$v(t) = 0 + \int_0^t (k_p(v_r(\tau) - v(\tau)) + k_i z(\tau))\ .d\tau$$

$$z(t) = 0 + \int_0^t (v_r(\tau) - v(\tau))\ .d\tau$$

Zélus: hybrid programs (Plant)

```
let hybrid integrator(yd, y0) = y where
  der y = yd init y0
```

val integrator : float × float $\xrightarrow{\texttt{C}}$ float

```
let hybrid pi_controller(v_r, i, hit) = angle where
  rec der angle = v init i
  and der v = k_p *. error +. k_i *. z init 0.0 reset hit(v0) -> v0
  and der z = error init 0.0 reset hit(_) -> 0.0
  and error = v_r -. v
```

val pi_controller : float × float × float signal $\xrightarrow{\texttt{C}}$ float

```
  present up(angle -. max) -> do
    emit hit = -0.8 *. last v
  done
```

Plant model = 3 segments + legs

```
let hybrid model (boom_ctl, stick_ctl, bucket_ctl, leg_ctl, lamp_ctl) =

  let  (boom_inout, boom) = segment (boom_range, boom_rate, boom_ctl)
  and (stick_inout, stick) = segment (stick_range, stick_rate, stick_ctl)
  and (bucket_inout, bucket) = segment (bucket_range, bucket_rate, bucket_ctl)

  and (leg_inout, leg_pos) = legsegment (leg_range, leg_rate, leg_ctl)

  in
  (leg_inout, boom_inout, stick_inout, bucket_inout)
```

Composing Controller and Plant

```
let hybrid main () = () where
  rec init sensors = ((false, false), (false, false), (false, false), (false, false))

  and init (boom_drive, stick_drive, bucket_drive) = (false, false, false)
  and init (alarm_lamp, done_lamp, cancel_lamp) = (false, false, false)

  and present sample → do
      (boom_ctl, stick_ctl, bucket_ctl, legs_ctl, lamps) =
                sampled_controller (last sensors, buttons (), second)
    done
  and sensors = model (boom_ctl, stick_ctl, bucket_ctl, legs_ctl, lamps)

  and sample = period (0.5)
  and second = present sample → not (last second) init true
```

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.

Our approach

- Hybrid dataflow language with hierarchical automata.
- System of kinds for rejecting unreasonable programs.
- Relate discrete to continuous via zero-crossings.
- Rigorous treatment of causality.
- Compilation via source-to-source transformations.
- Simulation using off-the-shelf numerical solvers.

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.

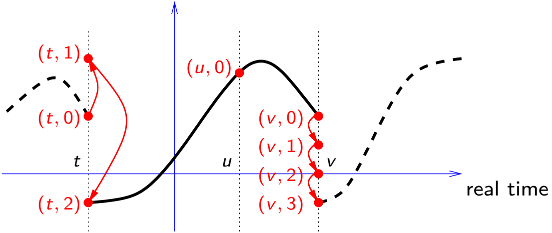
Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet.
A synchronous-based code generator for explicit hybrid systems languages.
In *Proc. 24th Int. Conf. on Compiler Construction (CC)*, page to appear, London, UK, April 2015.

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.

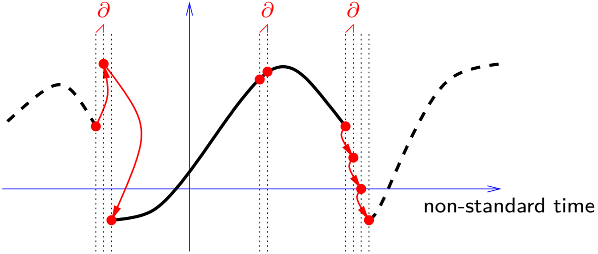
❷ What do compositions mean?

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 with reset on clock T: **der** x = e **init** e0 **reset** z1 → e1

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

- **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

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

- Smooth definitions at the expense of technical machinery.
- Tricky issues: standardization and link to simulations.