Zélus, a Synchronous Language with ODEs [1]

Marc Pouzet

École normale supérieure (DI)
Univ. Pierre et Marie Curie
INRIA
Paris, France

Seminar, Berkeley Univ.
Feb. 11, 2014

[1] Joint work with Benveniste, Bourke, Caillaud (INRIA) and Pagano (Esterel-Tech.).

Hybrid Systems Modelers

Program complex discrete systems and
their physical environments in a single language

Many tools exist

- Simulink/Stateflow, LabVIEW, Modelica, Ptolemy, ...

Focus on programming language issues to improve safety

Our proposal

- Build a hybrid modeler on top of a synchronous language
- Recycle existing techniques and tools
- Clarify underlying principles and guide language design/semantics

Reuse existing tools and techniques

Synchronous languages (SCADE/Lustre)

- Widely used for critical systems design and implementation
  - mathematically sound semantics
  - certified compilation (DO178C)
- Expressive language for both discrete controllers and mode changes
- Do not support modelling continuous dynamics

Off-the-shelf ODEs numeric solvers

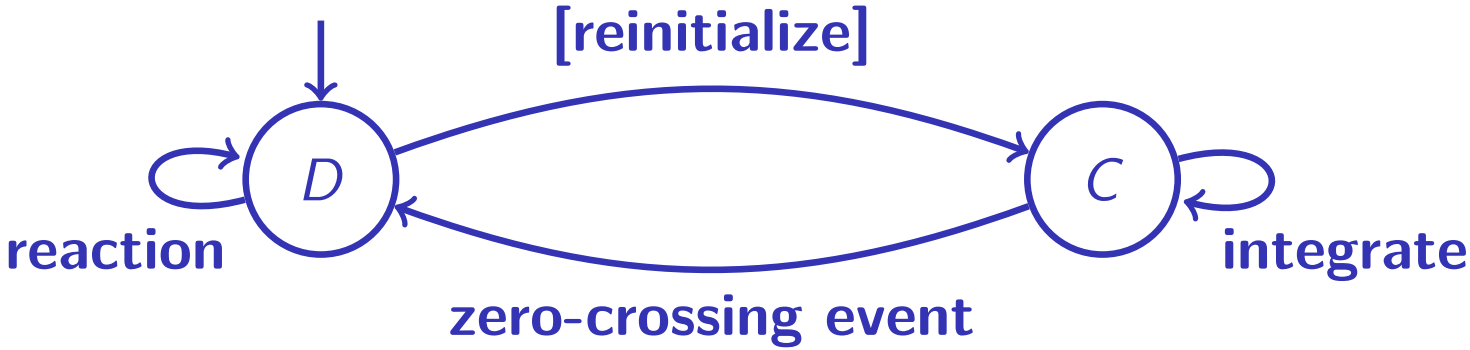
- Sundials CVODE (LLNL) among others, treated as black boxes
- Exploit existing techniques and (variable step) solvers

A conservative extension:

Any synchronous program must be compiled, optimized, and executed as per usual

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.

Causality Issues

Find sufficient conditions for the compiler to generate d, g and f.

The classical solution for difference equations (Lustre)

$$\mathsf{x} = 0 \rightarrow \text{pre } \mathsf{y} \quad \text{and} \quad \mathsf{y} = \text{if } \mathsf{c} \text{ then } \mathsf{x} + 1 \text{ else } \mathsf{x}$$

defines the two sequences $(x_n)_{n\in\mathbb{N}}$ and $(y_n)_{n\in\mathbb{N}}$:

$$x(0) = 0 \qquad y(n) = \texttt{if}\, c(n)\, \texttt{then}\, x(n) + 1\, \texttt{else}\, x(n)$$
$$x(n) = y(n-1)$$

If every feedback loop crosses a delay, elements can be computed sequentially. E.g.,an excerpt of the C code:

```
if (self->v_1) {x = 0;} else {x = self->v_2;};
if (c) {y = x+1;} else {y = x;};
self->v_2 = y; self->v_1 = false;
```

Can we reproduce the very same argument when mixing ODEs and discrete transitions and generate sequential code?

The Non Standard Interpretation of Hybrid Systems

We proposed in [CDC 2010, JCSS 2012] to build the semantics on Non-standard analysis.

der y = z init 4.0 and z = 10.0 − 0.1 ∗ y and k = y + 1.0

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

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

Consider the value that y would have if computed by an ideal solver taking an infinitesimal step of duration ∂.
${}^\star y(n)$ stands for the values of y at instant $n\partial$, with $n \in {}^\star\mathbb{N}$ a non-standard integer.

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

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:

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

${}^\star y(n)$ depends on itself so this set of equations is not causal.

Acessing the "left limit" of a signal (last y)

Two ways to break this cycle

- consider that the effect of the 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*)

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

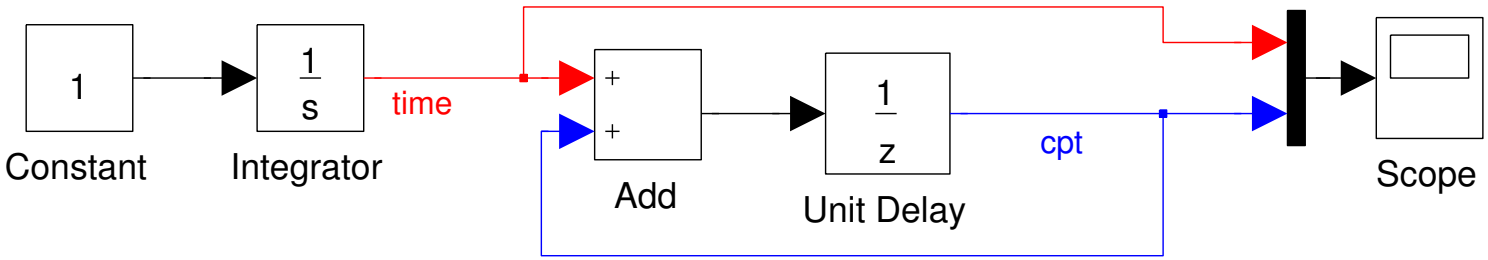
gives a program that is causal since ${}^\star y(n)$ no longer depends instantaneously on itself.

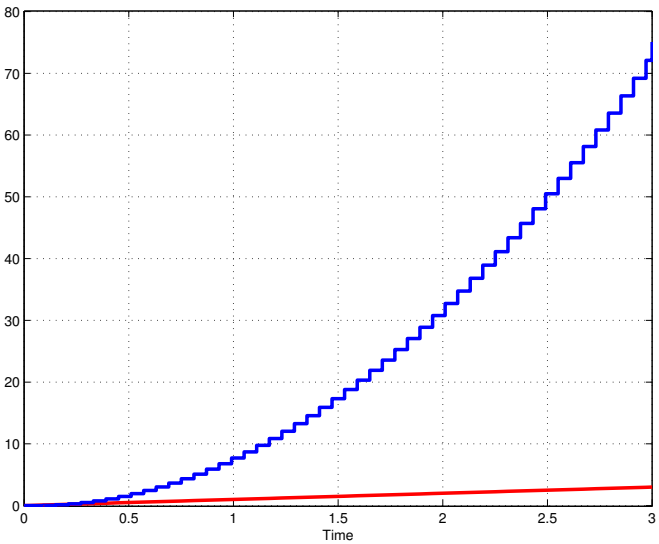
Write

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

Strange beasts. . .

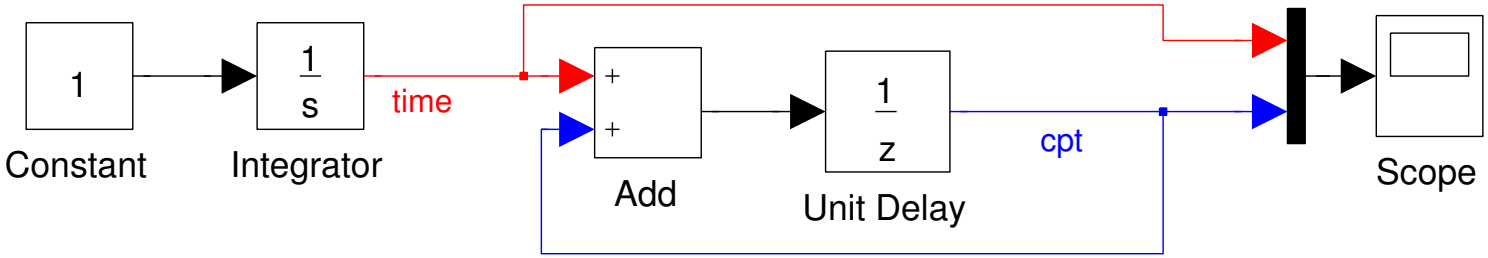
Typing issue: Mixing continuous and discrete components

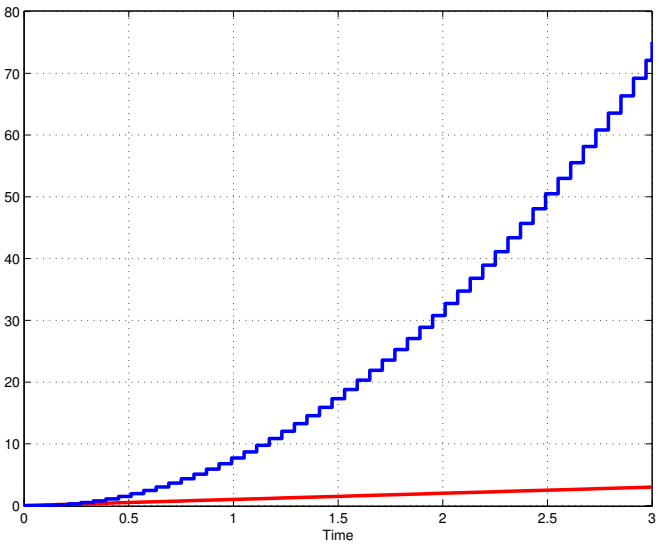




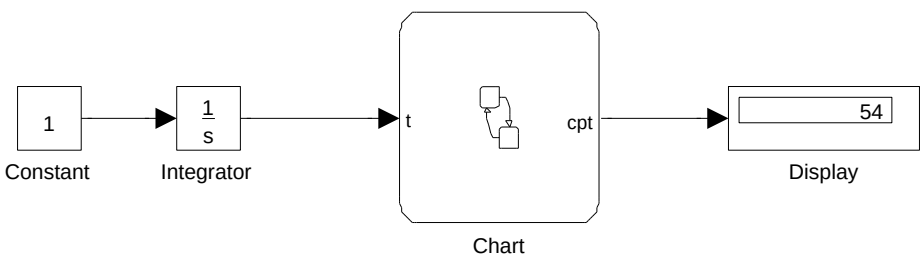
- Warning with 'Unit Delay' but not with 'Memory'.
- The shape of cpt depends on the steps chosen by the solver.
- Putting another component in parallel can change the result.

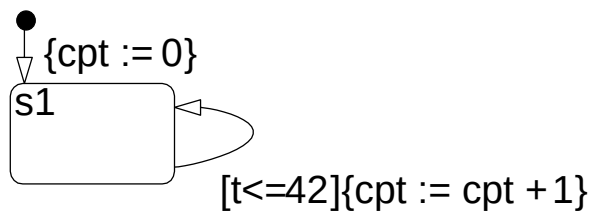
Typing issue: Mixing continuous and discrete components



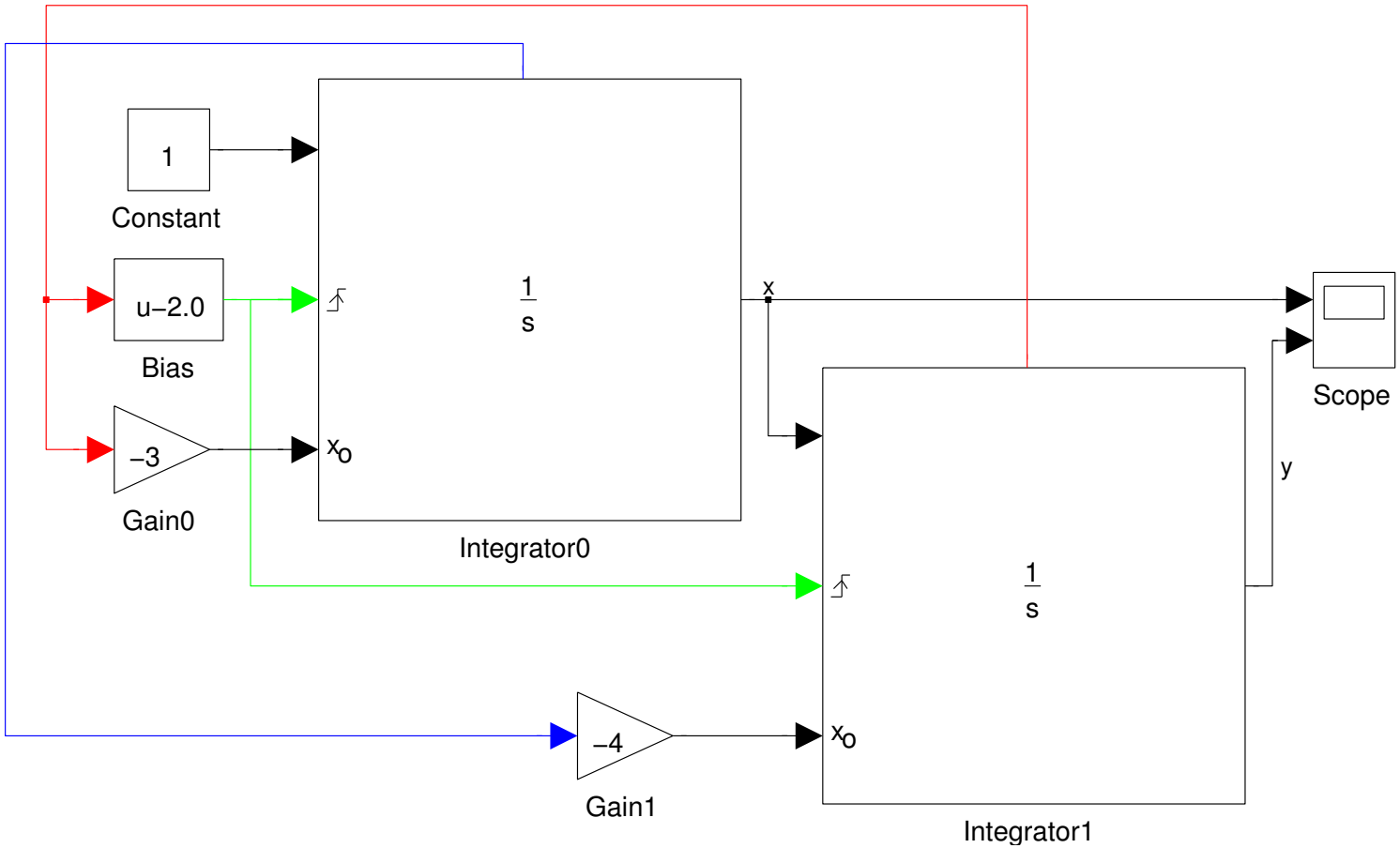


- Warning with 'Unit Delay' but not with 'Memory'.
- The shape of cpt depends on the steps chosen by the solver.
- Putting another component in parallel can change the result.
- Similar issues with Stateflow.





Causality issue: the Simulink state port



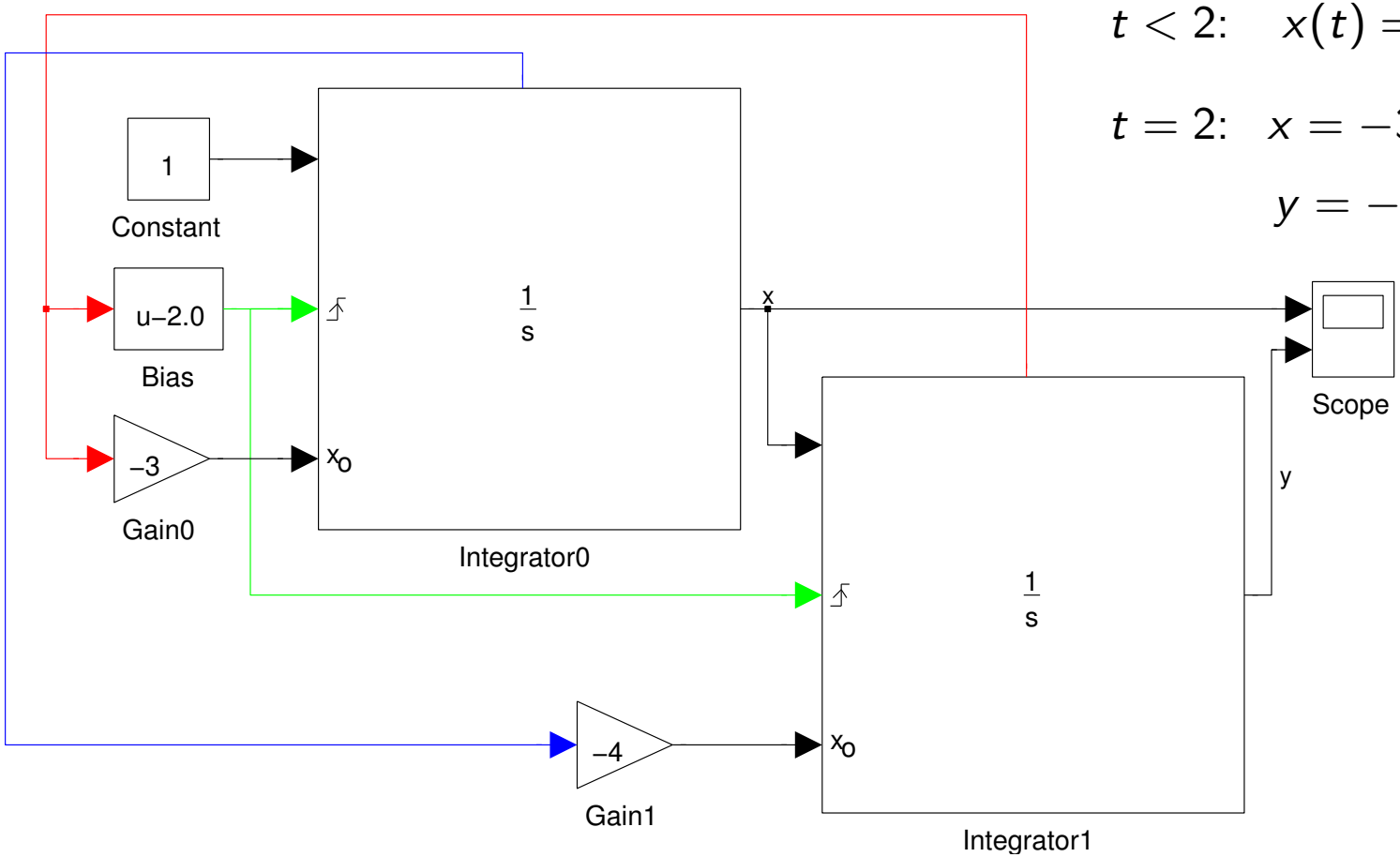
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.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

Causality issue: the Simulink state port

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

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

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



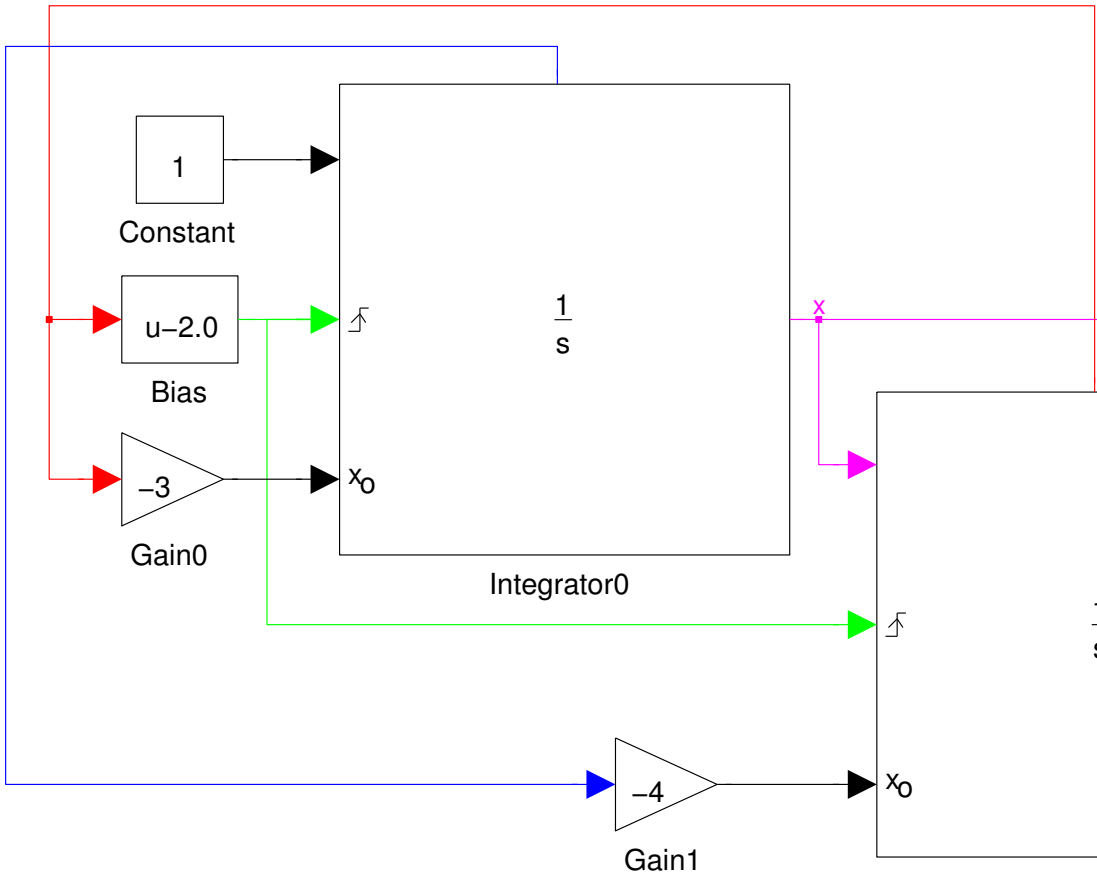
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.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

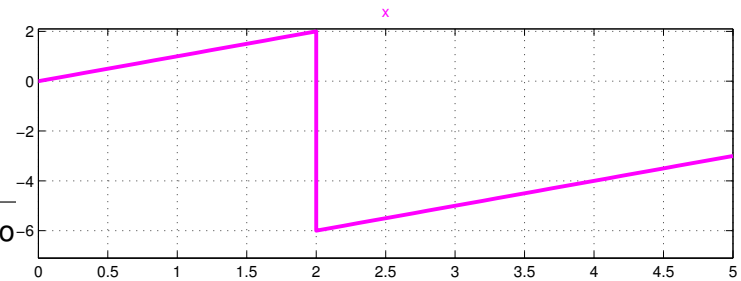
Causality issue: the Simulink state port

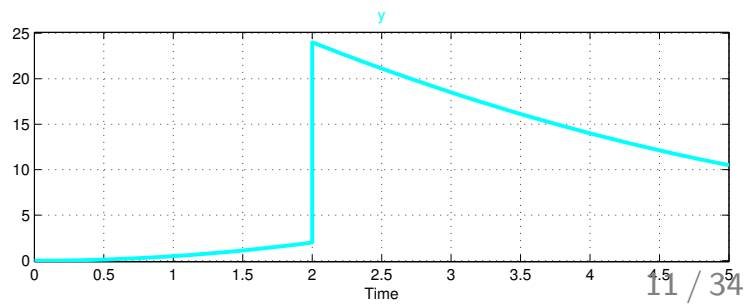
$$t < 2: \quad x(t) = t,\ y(t) = \frac{t^2}{2}$$

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

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







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

Excerpt of C code produced by RTW (release R2009)

```
// P_0 = -2.0 P_1 = -3.0 P_2 = -4.0 P_3 = 1.0
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;
       }
      ...
    } ... }
```

An explanation of the bug

The source program

$$
\begin{aligned}
&\text{der x} = 1.0 \text{ init } 0.0 \text{ reset z} \rightarrow -3.0 * \text{last y} \\
&\text{and der y} = \text{x init } 0.0 \text{ reset z} \rightarrow -4.0 * \text{last x} \\
&\text{and z} = \text{up}(\text{last x} - 2.0)
\end{aligned}
$$

Its non-standard interpretation

$$
\begin{aligned}
{}^\star x(n) &= \texttt{if } {}^\star z(n) \texttt{ then } -3 \cdot {}^\star y(n-1) \texttt{ else } {}^\star x(n-1) + \partial \\
{}^\star y(n) &= \texttt{if } {}^\star z(n) \texttt{ then } -4 \cdot {}^\star x(n-1) \texttt{ else } {}^\star y(n-1) + \partial \cdot {}^\star x(n-1) \\
&\ldots
\end{aligned}
$$

Explanation

- The first two equations are scheduled this way so ${}^\star x(n-1)$ is lost.
- This is a scheduling bug: the sequential code lacks a copy variable.

Zélus

`zelus.di.ens.fr`

Combinatorial and sequential functions

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

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

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

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

When feed into the compiler, we get:

```
val add : int × int  --A--> int
val mix_max : α× α --D--> α× α
val after : int × int  --D--> bool
```

x, y, etc. are infinite sequences of values.

Chronograms

time	=	0	1	2	3	4	5	6
x	=	2	4	2	1	2	3	2
y	=	3	6	5	1	1	9	0
$min\_max(x, y)$	=	$(2, 3)$	$(4, 6)$	$(2, 5)$	$(1, 1)$	$(1, 2)$	$(3, 9)$	$(0, 2)$
$pre(x)$	=	*nil*	2	4	2	1	2	3
$x \rightarrow y$	=	2	6	5	1	1	9	0
t	=	*false*	*true*	*true*	*false*	*false*	*true*	*true*
$after(2, t)$	=	*false*	*false*	*true*	*true*	*true*	*true*	*true*

- A signal is a sequence of values or stream.
- A system is function from streams to streams.
- Operations apply pointwise to their arguments.
- All streams progress synchronously.

The counter can be instantiated twice in a two state automaton,

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

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

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

The type signatures inferred by the compiler are:

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

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

Examples

Up to syntactic details, these are `Scade 6` or `Lucid Synchrone` programs. Now, a simple heat controller. [2]

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

let hybrid hysteresis_controller(temp) = active where
  rec automaton
      | Idle →   do active = false until (up(t_min −. temp)) then Active
      | Active → do active = true until (up(temp −. t_max)) then Idle

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

[2] This is the hybrid version of one of Nicolas Halbwachs' examples with which he presented Lustre at the Collège de France, in January 2010.

The Bouncing ball

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

The type signature is:

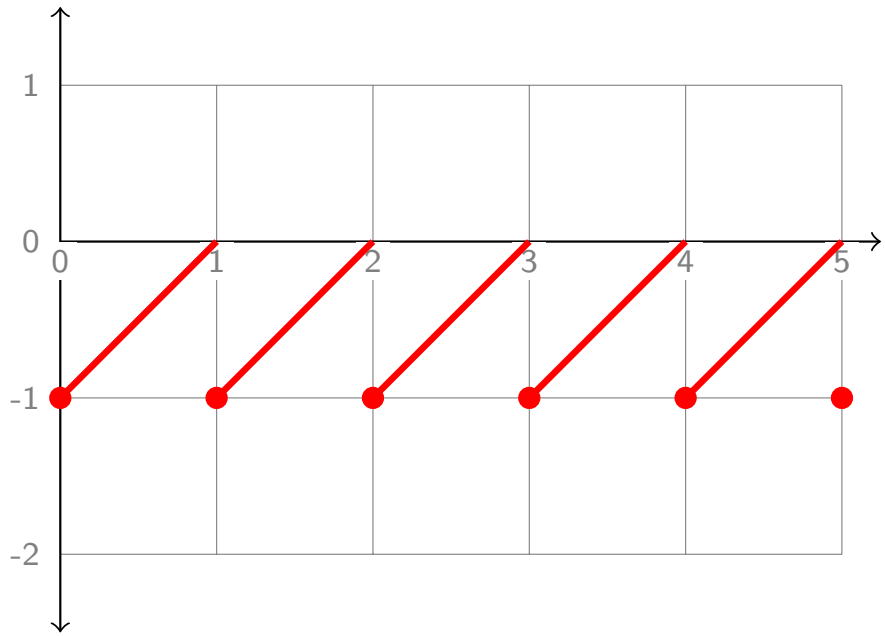
$$\texttt{val bouncing} : \texttt{float} \times \texttt{float} \times \texttt{float} \xrightarrow{c} \texttt{float} \times \texttt{float}$$

- When $-.$ y crosses zero, re-initialize the speed y' with $-. 0.9 * \texttt{last y'}$.
- 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.

ODEs and Zero-crossings

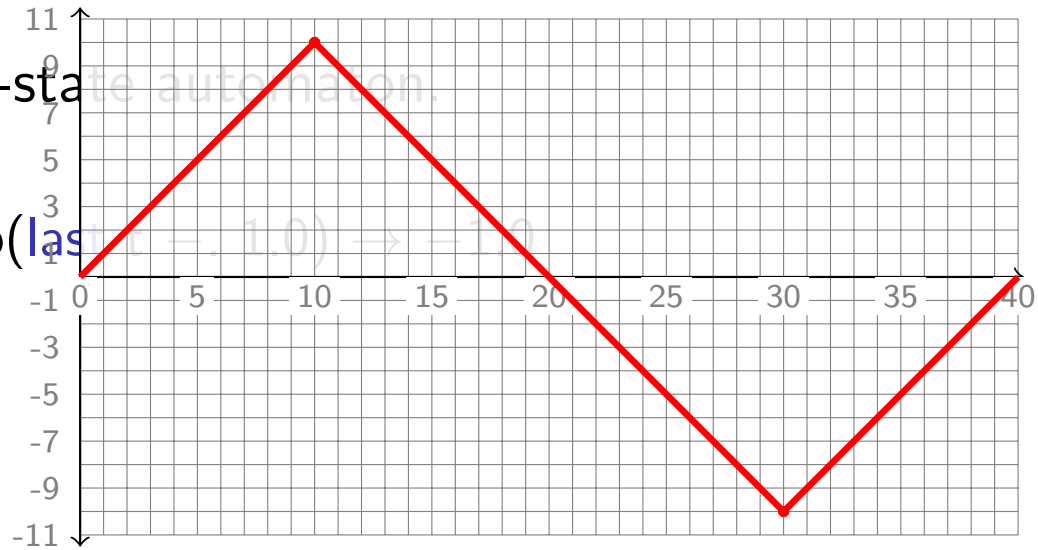
E.g., the sawtooth signal, the two-state automaton.

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



ODEs and Zero-crossings

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




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

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

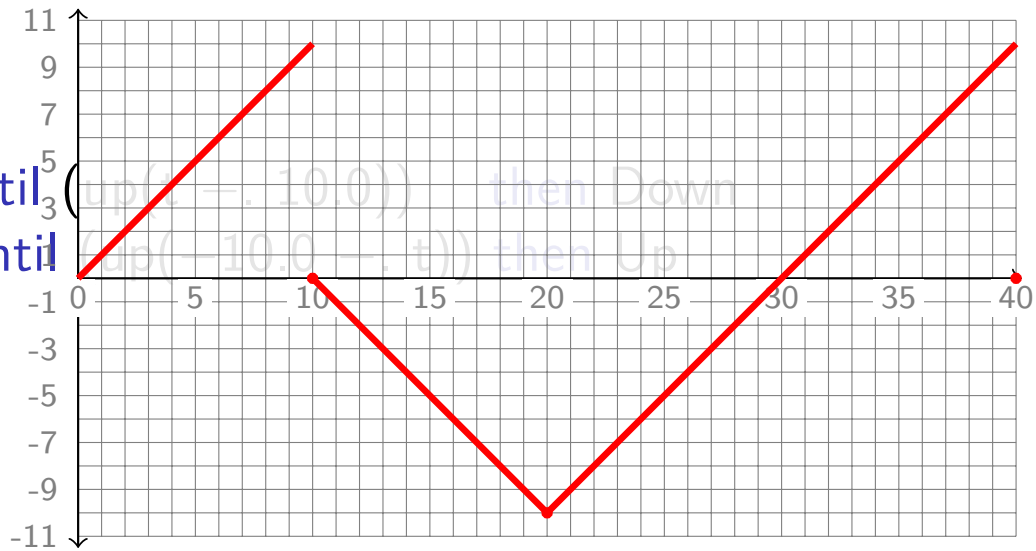
```
let hybrid fm'() = t where
  rec init t = 0.0
  and automaton
      | Up → do der t = 1.0
                until (up(t −. 10.0)) then do t = last t −. 10.0 in Down
      | Down → do der t = −1.0 until (up(−10.0 −. t)) then Up
```

ODEs and Zero-crossings

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

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

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




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

Two difficulties

Ensure that continuous and discrete time signals interfere correctly

- Discrete time should stay logical and independent on when the solver decides to stop.
- Otherwise, we get the same monsters as `Simulink`/`Stateflow` have.
- Rely on a dedicated type system.

Ensure that fix-point exist and code can be scheduled

- Algebraic loops must be statically detected.
- Introduce the operator last(x) as the "left limit" of a signal.
- Ensure that signals are left-continuous during integration.
- Rely on a dedicated type system.

Mixing discrete (logical) time and continuous time

Given:

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

Mixing discrete (logical) time and continuous time

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)
```

Mixing discrete (logical) time and continuous time

Given:

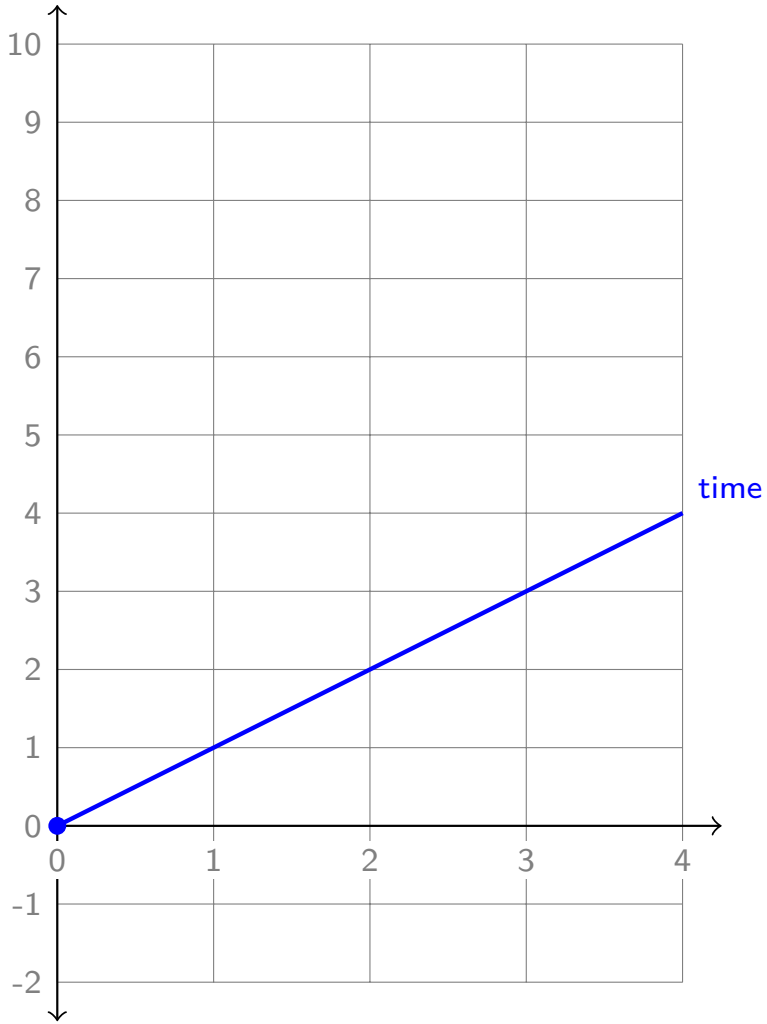
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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



Mixing discrete (logical) time and continuous time

Given:

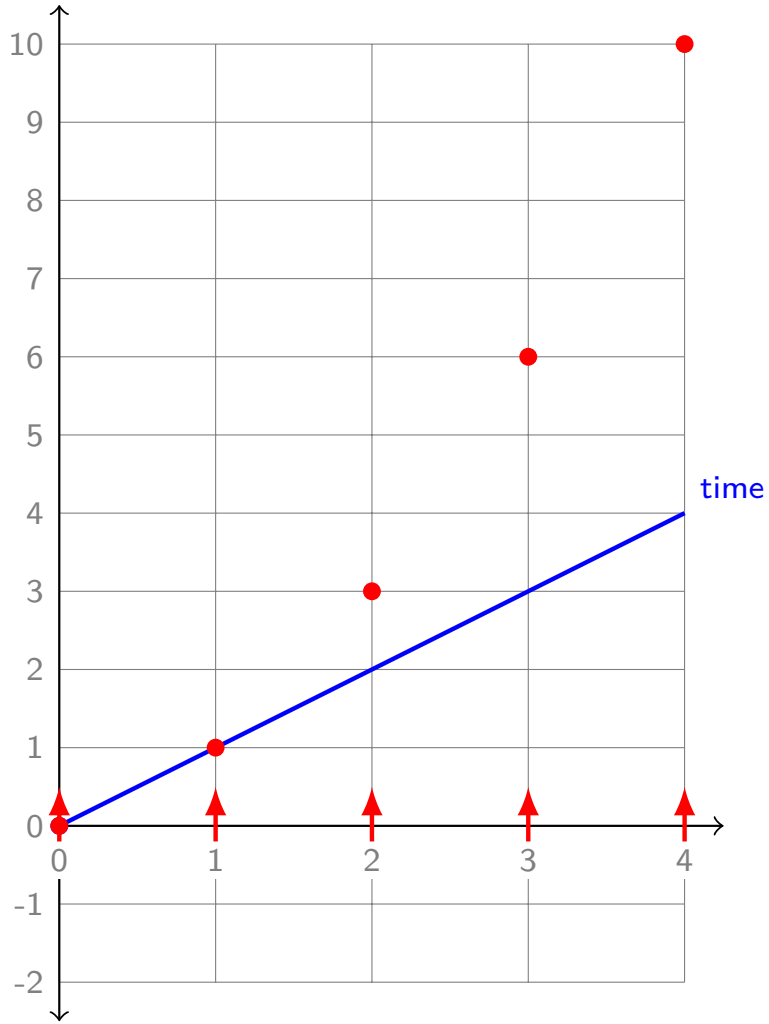
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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



Mixing discrete (logical) time and continuous time

Given:

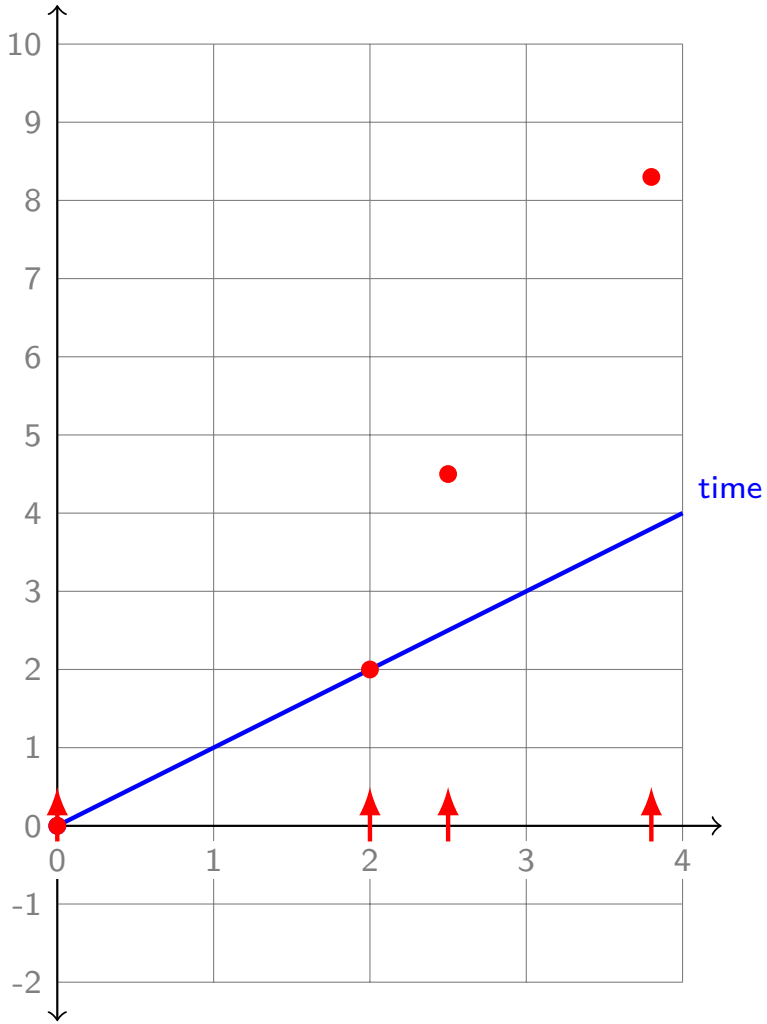
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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



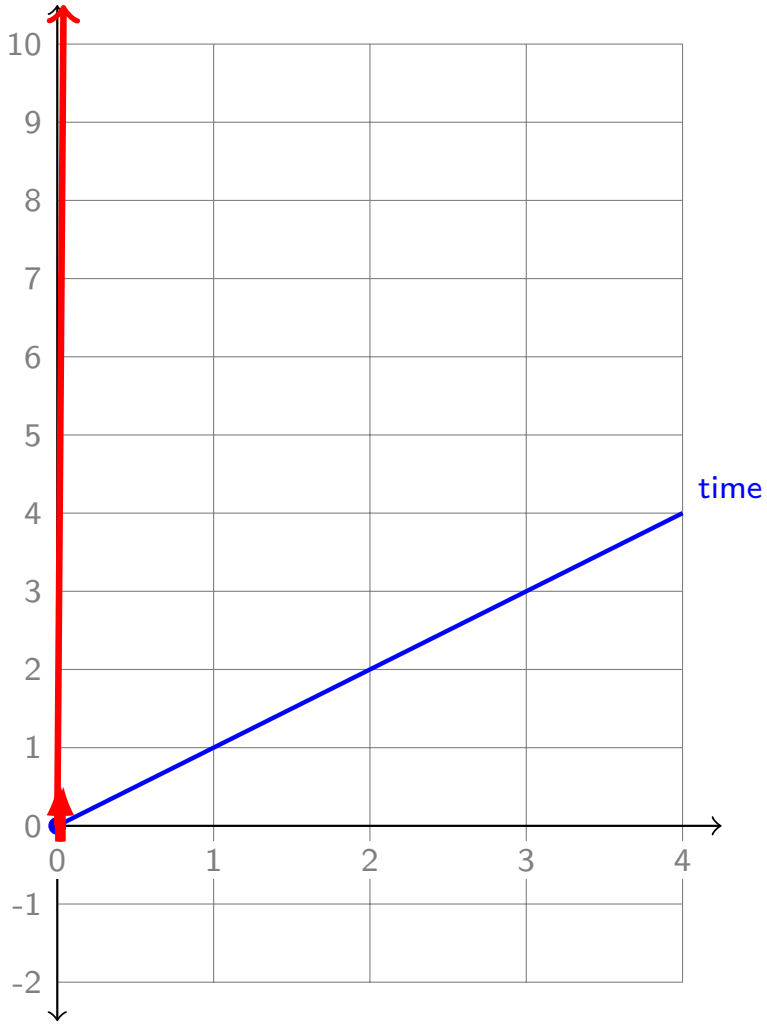
Mixing discrete (logical) time and continuous time

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

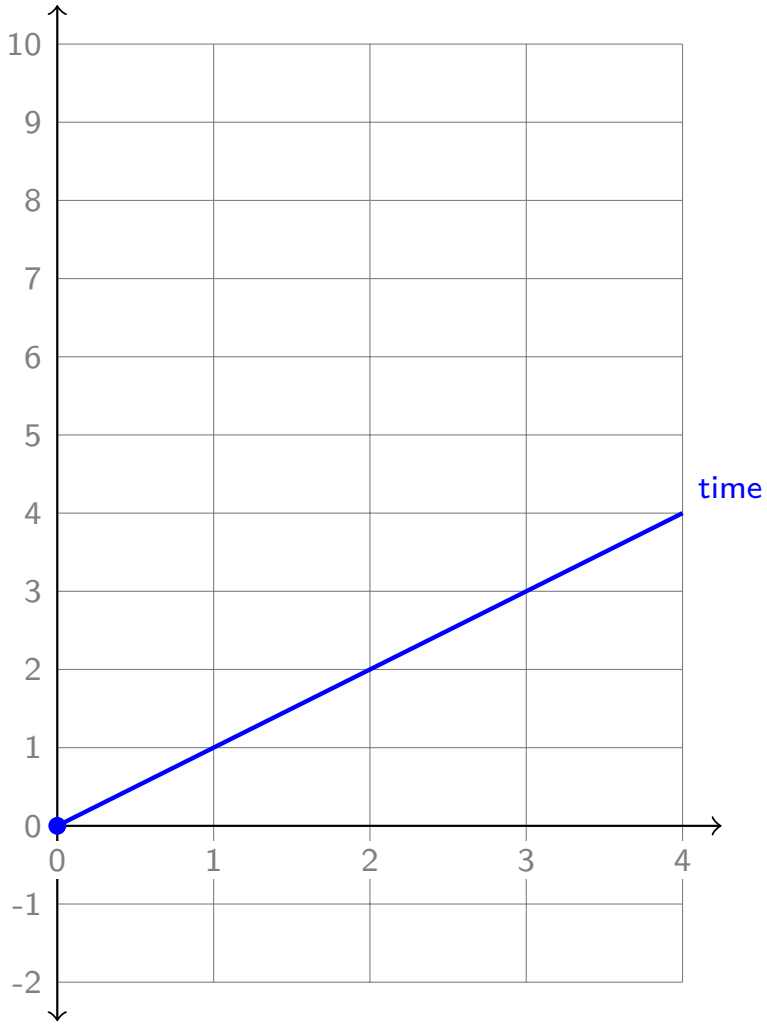
Mixing discrete (logical) time and continuous time

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

Mixing discrete (logical) time and continuous time

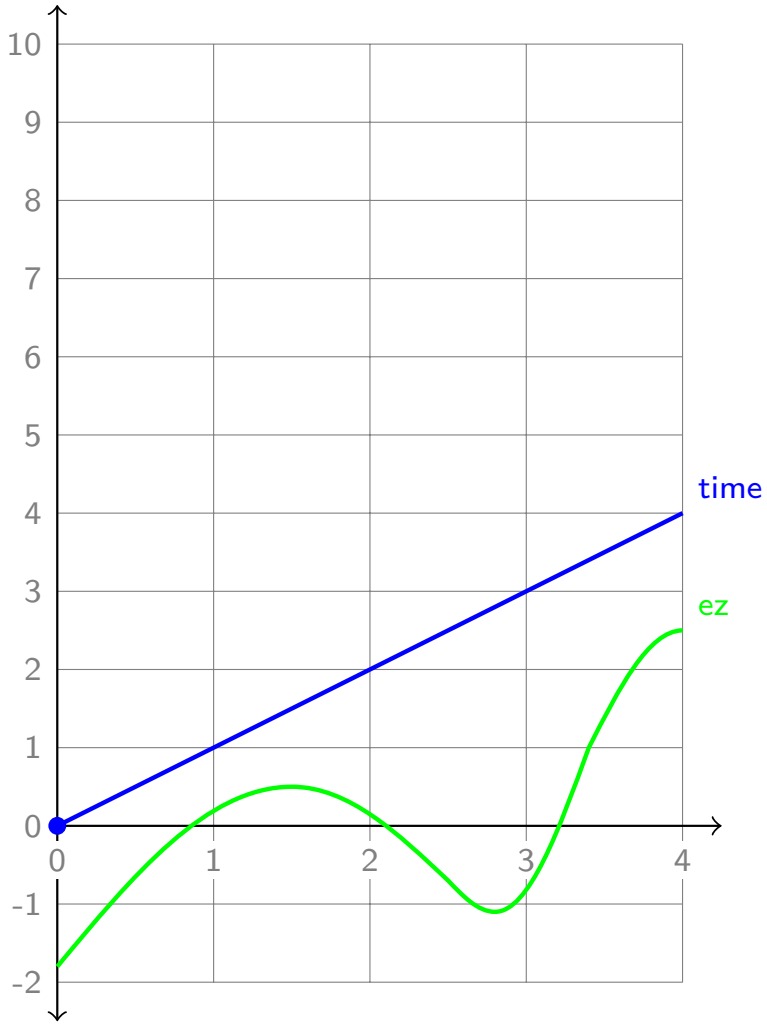
Given:

```
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



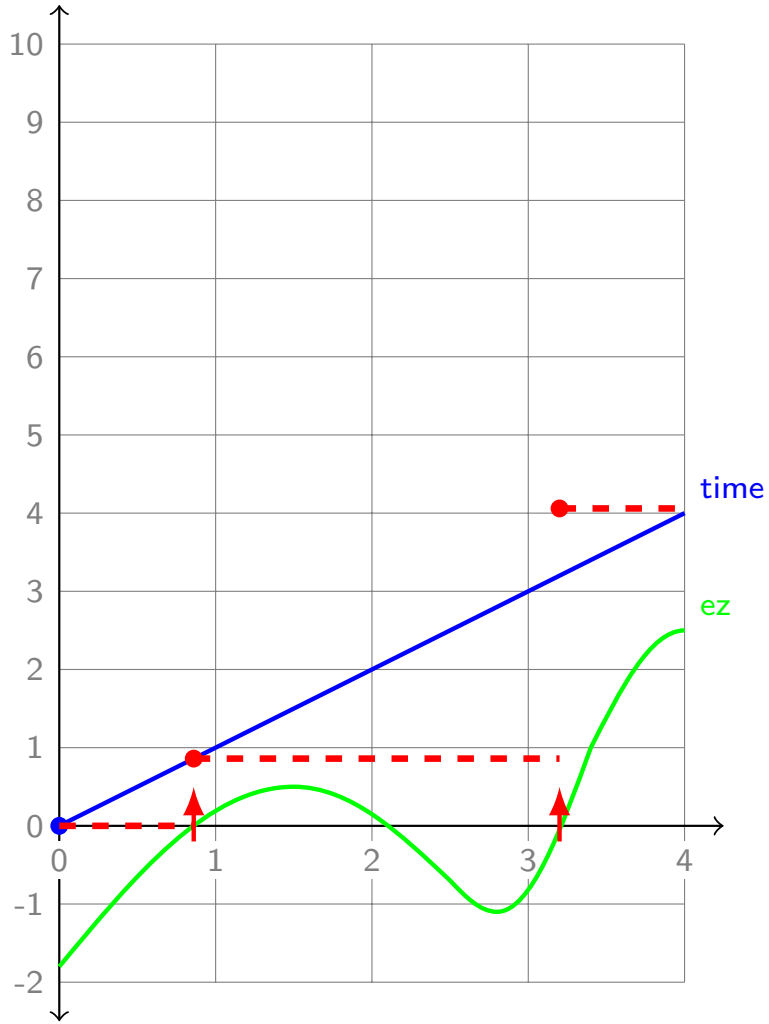
Mixing discrete (logical) time and continuous time

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)

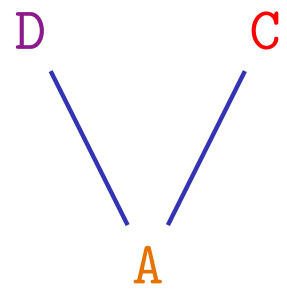
Try to minimize the effects of solver parameters and choices

Basic typing [LCTES'11]

A simple ML type-and-effect system.

The type language

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



Initial conditions

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

What about continuous automata? [EMSOFT'11]

Stateflow User's Guide The Mathworks, pages 16-26 to 16-29, 2011.

Design Considerations for Continuous-Time Modeling in Stateflow Charts

In this section...

Rationale for Design Considerations

To guarantee the integrity — or *smoothness* — of the results in continuous-time modeling, you must constrain your charts to a restricted subset of Stateflow chart semantics. The restricted semantics ensure that inputs do not depend on unpredictable factors — or *side effects* — such as:

- Simulink solver's guess for number of minor intervals in a major time step
- Number of iterations required to stabilize the integration loop or zero crossings loop

By minimizing side effects, a Stateflow chart can maintain its state at minor time steps and, therefore, update state only during major time steps when mode changes occur. Using this heuristic, a Stateflow chart can always compute outputs based on a constant state for continuous-time.

A Stateflow chart generates informative errors to help you correct semantic violations.

Summary of Rules for Continuous-Time Modeling

Here are the rules for modeling continuous-time Stateflow charts:

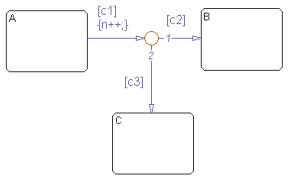
Update local data *only* in transition, `entry`, and `exit` actions

To maintain precision in continuous-time simulation, you should update local data (continuous or discrete) only during physical events at major time steps.

In Stateflow charts, physical events cause state transitions. Therefore, write to local data only in actions that execute during transitions, as follows:

- State `exit` actions, which execute before leaving the state at the beginning of the transition
- Transition actions, which execute during the transition
- State `entry` actions, which execute after entering the new state at the end of the transition
- Condition actions on a transition, but only if the transition directly reaches a state

Consider the following chart.



In this example, the action `{n++}` executes even when conditions `c2` and `c3` are false. In this case, `n` gets updated in a minor time step because there is no state transition.

Do not write to local continuous data in `during` actions because these actions execute in minor time steps.

Do not call Simulink functions in state `during` actions or transition conditions

This rule applies to continuous-time charts because you cannot call functions during minor time steps. You can call Simulink functions in state `entry` or `exit` actions and transition actions. However, if you try to call Simulink

functions in state `during` actions or transition conditions, an error message appears when you simulate your model.

For more information, see Chapter 24, "Using Simulink Functions in Stateflow Charts".

Compute derivatives only in `during` actions

A Simulink model reads continuous-time derivatives during minor time steps. The only part of a Stateflow chart that executes during minor time steps is the `during` action. Therefore, you should compute derivatives in `during` actions to give your Simulink model the most current calculation.

Do not read outputs and derivatives in states or transitions

This restriction ensures smooth outputs in a major time step because it prevents a Stateflow chart from using values that may no longer be valid in the current minor time step. Instead, a Stateflow chart always computes outputs from local discrete data, local continuous data, and chart inputs.

Use discrete variables to govern conditions in `during` actions

This restriction prevents mode changes from occurring between major time steps. When placed in `during` actions, conditions that affect control flow should be governed by discrete variables because they do not change between major time steps.

Do not use input events in continuous-time charts

The presence of input events makes a chart behave like a triggered subsystem and therefore unable to simulate in continuous-time. For example, the following model generates an error if the chart uses a continuous update method.

• • •

- 'Restricted subset of Stateflow chart semantics'
  - restricts side-effects to major time steps
  - supported by warnings and errors in tool **(mostly)**

What about continuous automata? [EMSOFT'11]

Stateflow User's Guide The Mathworks, pages 16-26 to 16-29, 2011.

Design Considerations for Continuous-Time Modeling in Stateflow Charts

In this section...

Rationale for Design Considerations

To guarantee the integrity — or *smoothness* — of the results in continuous-time modeling, you must constrain your charts to a restricted subset of Stateflow chart semantics. The restricted semantics ensure that inputs do not depend on unpredictable factors — or *side effects* — such as:

- Simulink solver's guess for number of minor intervals in a major time step
- Number of iterations required to stabilize the integration loop or zero crossings loop

By minimizing side effects, a Stateflow chart can maintain its state at minor time steps and, therefore, update state only during major time steps when mode changes occur. Using this heuristic, a Stateflow chart can always compute outputs based on a constant state for continuous-time.

A Stateflow chart generates informative errors to help you correct semantic violations.

Summary of Rules for Continuous-Time Modeling

Here are the rules for modeling continuous-time Stateflow charts:

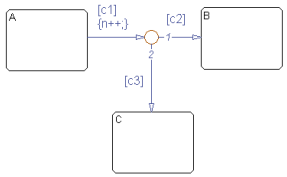
Update local data *only* in transition, `entry`, and `exit` actions

To maintain precision in continuous-time simulation, you should update local data (continuous or discrete) only during physical events at major time steps.

In Stateflow charts, physical events cause state transitions. Therefore, write to local data only in actions that execute during transitions, as follows:

- State `exit` actions, which execute before leaving the state at the beginning of the transition
- Transition actions, which execute during the transition
- State `entry` actions, which execute after entering the new state at the
- [illegible] transition ends at a state

Consider the following chart.



In this example, the action `{n++}` executes even when conditions `c2` and `c3` are false. In this case, `n` gets updated in a minor time step because there is no state transition.

Do not write to local continuous data in `during` actions because these actions execute in minor time steps.

Do not call Simulink functions in state `during` actions or transition conditions

This rule applies to continuous-time charts because you cannot call functions during minor time steps. You can call Simulink functions in state `entry` or `exit` actions and transition actions. However, if you try to call Simulink

functions in state `during` actions or transition conditions, an error message appears when you simulate your model.

For more information, see Chapter 24, "Using Simulink Functions in Stateflow Charts".

Compute derivatives only in `during` actions

A Simulink model reads continuous-time derivatives during minor time steps. The only part of a Stateflow chart that executes during minor time steps is the `during` action. Therefore, you should compute derivatives in `during` actions to give your Simulink model the most current calculation.

Do not read outputs and derivatives in states or transitions

This restriction ensures smooth outputs in a major time step because it prevents a Stateflow chart from using values that may no longer be valid in the current minor time step. Instead, a Stateflow chart always computes outputs from local discrete data, local continuous data, and chart inputs.

Use discrete variables to govern conditions in `during` actions

This restriction prevents mode changes from occurring between major time steps. When placed in `during` actions, conditions that affect control flow should be governed by discrete variables because they do not change between major time steps.

Do not use input events in continuous-time charts

The presence of input events makes a chart behave like a triggered subsystem and therefore unable to simulate in continuous-time. For example, the following model generates an error if the chart uses a continuous update method.

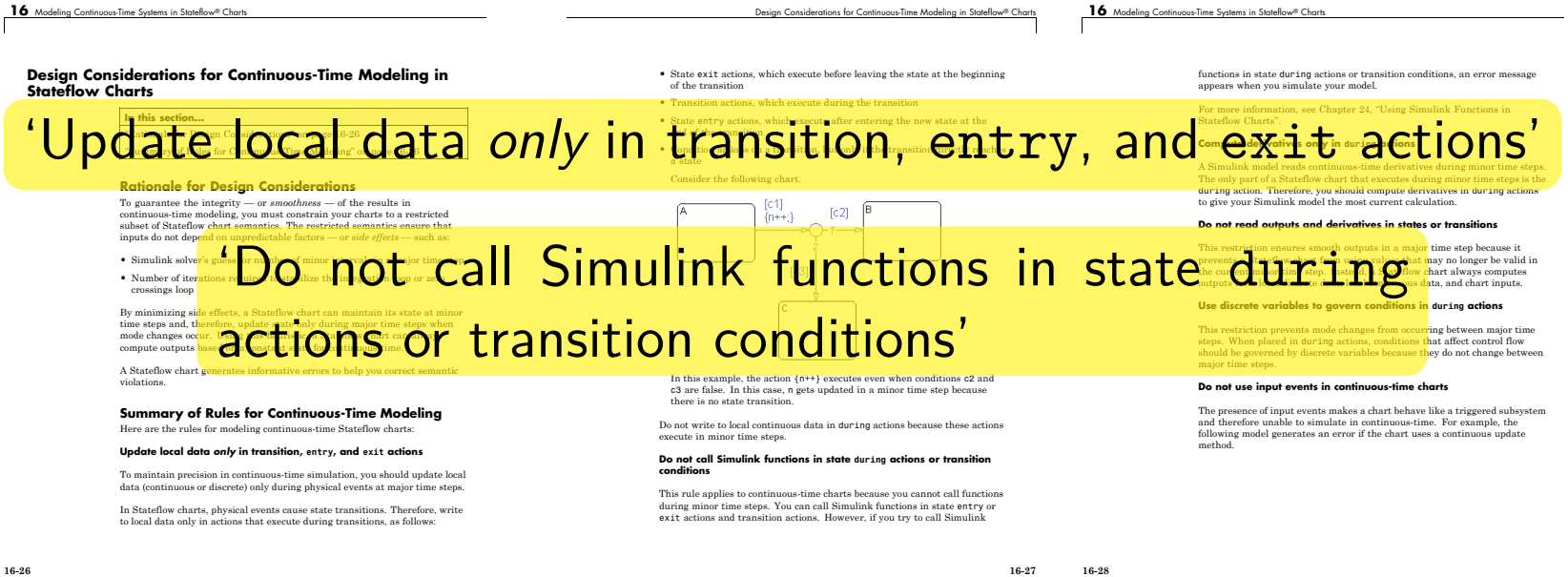
'Update local data *only* in transition, `entry`, and `exit` actions'

. . .

- 'Restricted subset of Stateflow chart semantics'
  - restricts side-effects to major time steps
  - supported by warnings and errors in tool **(mostly)**

What about continuous automata? [EMSOFT'11]

Stateflow User's Guide The Mathworks, pages 16-26 to 16-29, 2011.



16 Modeling Continuous-Time Systems in Stateflow® Charts

Design Considerations for Continuous-Time Modeling in Stateflow Charts

In this section...

Rationale for Design Considerations

To guarantee the integrity — or *smoothness* — of the results in continuous-time modeling, you must constrain your charts to a restricted subset of Stateflow chart semantics. The restricted semantics ensure that inputs do not depend on unpredictable factors — or *side effects* — such as:

- Simulink solver's guess for number of minor time steps in major time step
- Number of iterations required to localize the ... crossings loop

By minimizing side effects, a Stateflow chart can maintain its state at minor time steps and, therefore, update state only during major time steps when mode changes occur. ... compute outputs based ... major time step.

A Stateflow chart generates informative errors to help you correct semantic violations.

Summary of Rules for Continuous-Time Modeling

Here are the rules for modeling continuous-time Stateflow charts:

Update local data *only* in transition, `entry`, and `exit` actions

To maintain precision in continuous-time simulation, you should update local data (continuous or discrete) only during physical events at major time steps.

In Stateflow charts, physical events cause state transitions. Therefore, write to local data only in actions that execute during transitions, as follows:

16-26

Design Considerations for Continuous-Time Modeling in Stateflow® Charts

- State `exit` actions, which execute before leaving the state at the beginning of the transition
- Transition actions, which execute during the transition
- State `entry` actions, which execute after entering the new state at the ... a state

Consider the following chart.

In this example, the action `{n++;}` executes even when conditions `c2` and `c3` are false. In this case, `n` gets updated in a minor time step because there is no state transition.

Do not write to local continuous data in `during` actions because these actions execute in minor time steps.

Do not call Simulink functions in state `during` actions or transition conditions

This rule applies to continuous-time charts because you cannot call functions during minor time steps. You can call Simulink functions in state `entry` or `exit` actions and transition actions. However, if you try to call Simulink

16-27

16 Modeling Continuous-Time Systems in Stateflow® Charts

functions in state `during` actions or transition conditions, an error message appears when you simulate your model.

For more information, see Chapter 24, "Using Simulink Functions in Stateflow Charts".

Compute derivatives only in `during` actions

A Simulink model reads continuous-time derivatives during minor time steps. The only part of a Stateflow chart that executes during minor time steps is the `during` action. Therefore, you should compute derivatives in `during` actions to give your Simulink model the most current calculation.

Do not read outputs and derivatives in states or transitions

This restriction ensures smooth outputs in a major time step because it prevents ... that may no longer be valid in the current ... Stateflow chart always computes outputs ... data, and chart inputs.

Use discrete variables to govern conditions in `during` actions

This restriction prevents mode changes from occurring between major time steps. When placed in `during` actions, conditions that affect control flow should be governed by discrete variables because they do not change between major time steps.

Do not use input events in continuous-time charts

The presence of input events makes a chart behave like a triggered subsystem and therefore unable to simulate in continuous-time. For example, the following model generates an error if the chart uses a continuous update method.

16-28

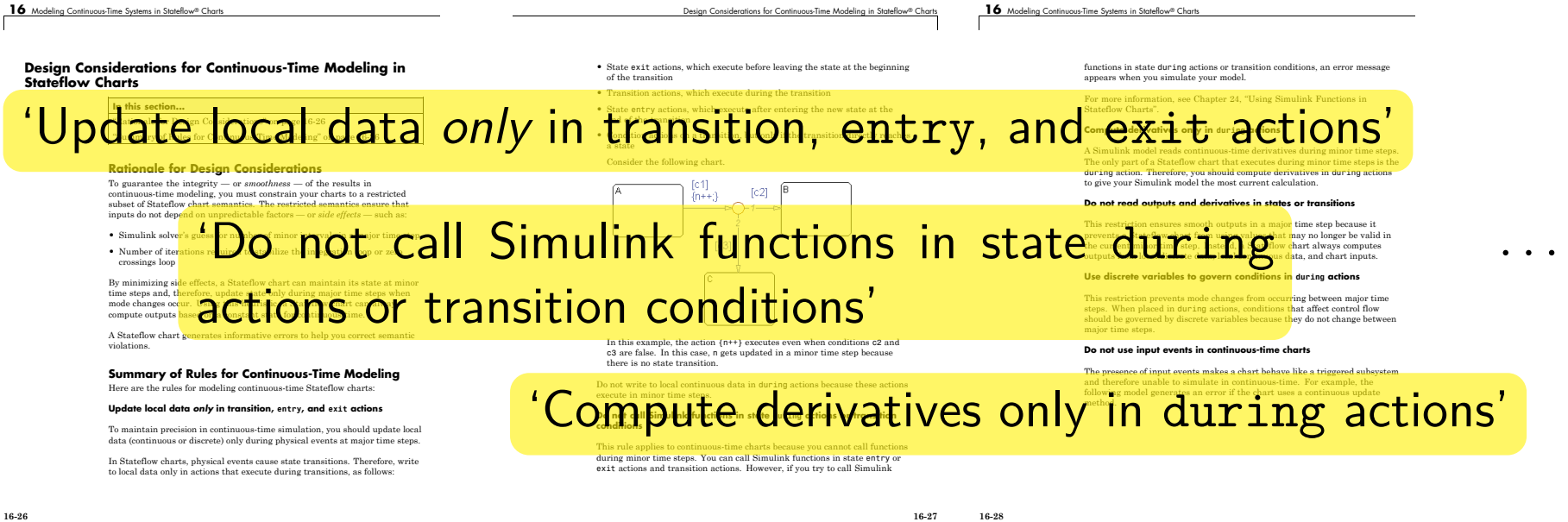
'Update local data *only* in transition, `entry`, and `exit` actions'

'Do not call Simulink functions in state during actions or transition conditions' ...

- 'Restricted subset of Stateflow chart semantics'
  - restricts side-effects to major time steps
  - supported by warnings and errors in tool **(mostly)**

What about continuous automata? [EMSOFT'11]

Stateflow User's Guide The Mathworks, pages 16-26 to 16-29, 2011.



16 Modeling Continuous-Time Systems in Stateflow® Charts

Design Considerations for Continuous-Time Modeling in Stateflow Charts

In this section...

Rationale for Design Considerations

To guarantee the integrity — or *smoothness* — of the results in continuous-time modeling, you must constrain your charts to a restricted subset of Stateflow chart semantics. The restricted semantics ensure that inputs do not depend on unpredictable factors — or *side effects* — such as:

- Simulink solver's guess for number of minor time steps in major time step or zero crossings loop

By minimizing side effects, a Stateflow chart can maintain its state at minor time steps and, therefore, update state only during major time steps when mode changes occur. The chart can compute outputs based on the current time.

A Stateflow chart generates informative errors to help you correct semantic violations.

Summary of Rules for Continuous-Time Modeling

Here are the rules for modeling continuous-time Stateflow charts:

Update local data *only* in transition, `entry`, and `exit` actions

To maintain precision in continuous-time simulation, you should update local data (continuous or discrete) only during physical events at major time steps.

In Stateflow charts, physical events cause state transitions. Therefore, write to local data only in actions that execute during transitions, as follows:

16-26

Design Considerations for Continuous-Time Modeling in Stateflow® Charts

- State `exit` actions, which execute before leaving the state at the beginning of the transition
- Transition actions, which execute during the transition
- State `entry` actions, which execute after entering the new state at the destination state

Consider the following chart.

In this example, the action `{n++;}` executes even when conditions `c2` and `c3` are false. In this case, `n` gets updated in a minor time step because there is no state transition.

Do not write to local continuous data in `during` actions because these actions execute in minor time steps.

Do not call Simulink functions in state `during` actions or transition conditions

This rule applies to continuous-time charts because you cannot call functions during minor time steps. You can call Simulink functions in state `entry` or `exit` actions and transition actions. However, if you try to call Simulink

16-27

16 Modeling Continuous-Time Systems in Stateflow® Charts

functions in state `during` actions or transition conditions, an error message appears when you simulate your model.

For more information, see Chapter 24, "Using Simulink Functions in Stateflow Charts".

Compute derivatives only in `during` actions

A Simulink model reads continuous-time derivatives during minor time steps. The only part of a Stateflow chart that executes during minor time steps is the `during` action. Therefore, you should compute derivatives in `during` actions to give your Simulink model the most current calculation.

Do not read outputs and derivatives in states or transitions

This restriction ensures smooth outputs in a major time step because it prevents [illegible] that may no longer be valid in the current [illegible]. Instead, a Stateflow chart always computes outputs [illegible] data, and chart inputs.

Use discrete variables to govern conditions in `during` actions

This restriction prevents mode changes from occurring between major time steps. When placed in `during` actions, conditions that affect control flow should be governed by discrete variables because they do not change between major time steps.

Do not use input events in continuous-time charts

The presence of input events makes a chart behave like a triggered subsystem and therefore unable to simulate in continuous-time. For example, the following model generates an error if the chart uses a continuous update method.

16-28

'Update local data *only* in transition, `entry`, and `exit` actions'

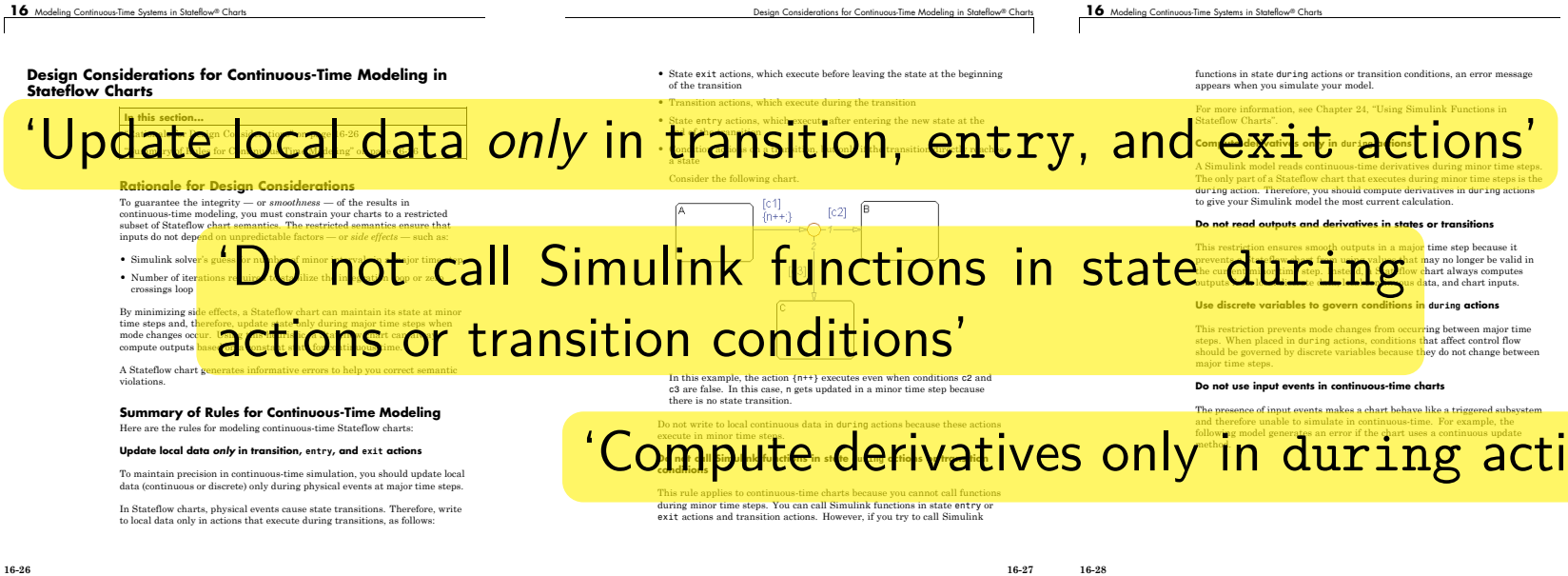
'Do not call Simulink functions in state `during` actions or transition conditions' . . .

'Compute derivatives only in `during` actions'

- 'Restricted subset of Stateflow chart semantics'
  - restricts side-effects to major time steps
  - supported by warnings and errors in tool **(mostly)**

What about continuous automata? [EMSOFT'11]

Stateflow User's Guide The Mathworks, pages 16-26 to 16-29, 2011.



16 Modeling Continuous-Time Systems in Stateflow® Charts

Design Considerations for Continuous-Time Modeling in Stateflow Charts

In this section...

Rationale for Design Considerations

To guarantee the integrity — or *smoothness* — of the results in continuous-time modeling, you must constrain your charts to a restricted subset of Stateflow chart semantics. The restricted semantics ensure that inputs do not depend on unpredictable factors — or *side effects* — such as:

- Simulink solver's guess for the number of minor time steps per major time step or zero crossings loop

By minimizing side effects, a Stateflow chart can maintain its state at minor time steps and, therefore, update state only during major time steps when mode changes occur. A Stateflow chart can compute outputs based on the current time.

A Stateflow chart generates informative errors to help you correct semantic violations.

Summary of Rules for Continuous-Time Modeling

Here are the rules for modeling continuous-time Stateflow charts:

Update local data *only* in transition, `entry`, and `exit` actions

To maintain precision in continuous-time simulation, you should update local data (continuous or discrete) only during physical events at major time steps.

In Stateflow charts, physical events cause state transitions. Therefore, write to local data only in actions that execute during transitions, as follows:

16-26

Design Considerations for Continuous-Time Modeling in Stateflow® Charts

- State `exit` actions, which execute before leaving the state at the beginning of the transition
- Transition actions, which execute during the transition
- State `entry` actions, which execute after entering the new state at the

a state

Consider the following chart.

In this example, the action `{n++;}` executes even when conditions `c2` and `c3` are false. In this case, `n` gets updated in a minor time step because there is no state transition.

Do not write to local continuous data in `during` actions because these actions execute in minor time steps.

Conditions

This rule applies to continuous-time charts because you cannot call functions during minor time steps. You can call Simulink functions in state `entry` or `exit` actions and transition actions. However, if you try to call Simulink

16-27

16 Modeling Continuous-Time Systems in Stateflow® Charts

functions in state `during` actions or transition conditions, an error message appears when you simulate your model.

For more information, see Chapter 24, "Using Simulink Functions in Stateflow Charts".

Compute derivatives only in `during` actions

A Simulink model reads continuous-time derivatives during minor time steps. The only part of a Stateflow chart that executes during minor time steps is the `during` action. Therefore, you should compute derivatives in `during` actions to give your Simulink model the most current calculation.

Do not read outputs and derivatives in states or transitions

This restriction ensures smooth outputs in a major time step because it prevents [illegible] that may no longer be valid in the current time step. Stateflow chart always computes outputs [illegible] data, and chart inputs.

Use discrete variables to govern conditions in `during` actions

This restriction prevents mode changes from occurring between major time steps. When placed in `during` actions, conditions that affect control flow should be governed by discrete variables because they do not change between major time steps.

Do not use input events in continuous-time charts

The presence of input events makes a chart behave like a triggered subsystem and therefore unable to simulate in continuous-time. For example, the following model generates an error if the chart uses a continuous update [illegible]

16-28

'Update local data *only* in transition, `entry`, and `exit` actions'

'Do not call Simulink functions in state during actions or transition conditions' ...

'Compute derivatives only in `during` actions'

- 'Restricted subset of Stateflow chart semantics'
  - restricts side-effects to major time steps
  - supported by warnings and errors in tool **(mostly)**
- Our `D`/`C`/`A`/`zero` system extends naturally for the same effect.
- For both discrete (synchronous) and continuous (hybrid) contexts.

Causality issues (feedback loops)

Which programs should we accept?

- OK to reject (no solution).

  rec x = x +. 1.0

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

  rec x = 1.0 −. x

- But NOK if sequential code generation is targeted (algebraic loop).
- OK in constructive logic (Esterel)

  rec z1 = if c then z2 else y
  and z2 = if c then x else z1

- But it calls for an expensive boolean analysis.

Can we find a simple and uniform justification for a program mixing continuous-time and discrete-time signals to be causaly correct?

Causality [HSCC'14]

We follow the `Lustre`/`Lucid Synchrone` condition for causality: every feedback loop must cross a delay.
Intuition: associate a time stamp to every expression and ensure that the relation between those time stamps is a partial order.

The type language

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

Precedence relation:

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

$<$ must be a strict partial order. $C \vdash ct_1 < ct_2$ means that ct_1 precedes ct_2 according to C.

Precedence relation

Build the transitive closure of $<$ and lift it to pairs and environments.

$$\frac{}{C + \alpha_1 < \alpha_2 \vdash \alpha_1 < \alpha_2}\ (\textsc{taut})$$

$$\frac{C \vdash ct_1 < ct' \qquad C \vdash ct' < ct_2}{C \vdash ct_1 < ct_2}\ (\textsc{trans})$$

$$\frac{C \vdash ct_1 < ct_1' \qquad C \vdash ct_2 < ct_2'}{C \vdash ct_1 \times ct_2 < ct_1' \times ct_2'}\ (\textsc{pair})$$

$$\frac{\forall i \in \{1, .., n\}, C \vdash ct_i < ct_i'}{C \vdash [x_1 : ct_1; ...; x_n : ct_n] < [x_1 : ct_1'; ...; x_n : ct_n']}\ (\textsc{env})$$

The Type System

Type Judgments

$$\frac{}{C \mid G, H \vdash_k e : ct}\ (\textsc{typ-exp}) \qquad \frac{}{C \mid G, H \vdash_k E : H'}\ (\textsc{typ-env})$$

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

Initial Conditions

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

The Typing Rules

$$\frac{C, ct_1 \xrightarrow{k} ct_2 \in \mathit{Inst}(G(f)) \qquad C \mid G, H \vdash_k e : ct_1}{C \mid G, H \vdash_k f(e) : ct_2} \;(\textsc{app})$$

$$C \mid G, H + x : ct \vdash_k x : ct \;(\textsc{var})$$

$$\frac{C \vdash ct_2 < ct_1}{C \mid G, H + x : ct_1 \vdash_{\mathtt{D}} \mathsf{last}(x) : ct_2} \;(\textsc{last})$$

$$\frac{C \mid G, H \vdash_k p : ct \qquad C \mid G, H \vdash_k e : ct}{C \mid G, H \vdash_k p = e : [ct/p]} \;(\textsc{eq})$$

$$\frac{C \mid G, H \vdash_{\mathtt{C}} e : ct_1 \qquad C \vdash ct_2 < ct_1}{C \mid G, H \vdash_{\mathtt{C}} \mathtt{der}\, x \;=\; e : [ct_2/x]} \;(\textsc{der})$$

$$\frac{C \mid G, H \vdash_k e : ct \qquad C \vdash ct < ct'}{C \mid G, H \vdash_k e : ct'} \;(\textsc{sub})$$

Examples

Discrete case

```
let node integr(xi, x') = x where
  rec x = xi → pre x + (pre x' * step)

let node heat(temp0, gain) = temp where
  rec temp = integr(temp0, gain − temp))

let cycle() = (x, y) where rec y = x + 1 and x = y + 2
```

Indeed, taken $x : \alpha_x$ and $y : \alpha_y$, the first equation is correct if both $C \vdash \alpha_x < \alpha_y$ and $C \vdash \alpha_y < \alpha_x$. This means that C must contain $\{\alpha_x < \alpha_y, \alpha_y < \alpha_x\}$ which is cyclic.

Continuous case

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

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

The causality of last

Yet, last x must appear in a discrete context only. In NS semantics:

$$\mathsf{last}(x)(t) = x(^{\bullet}t)$$

last(x) does not necessarily break causality cycles. E.g.:

rec x = last x + 1.0

A more expressive analysis

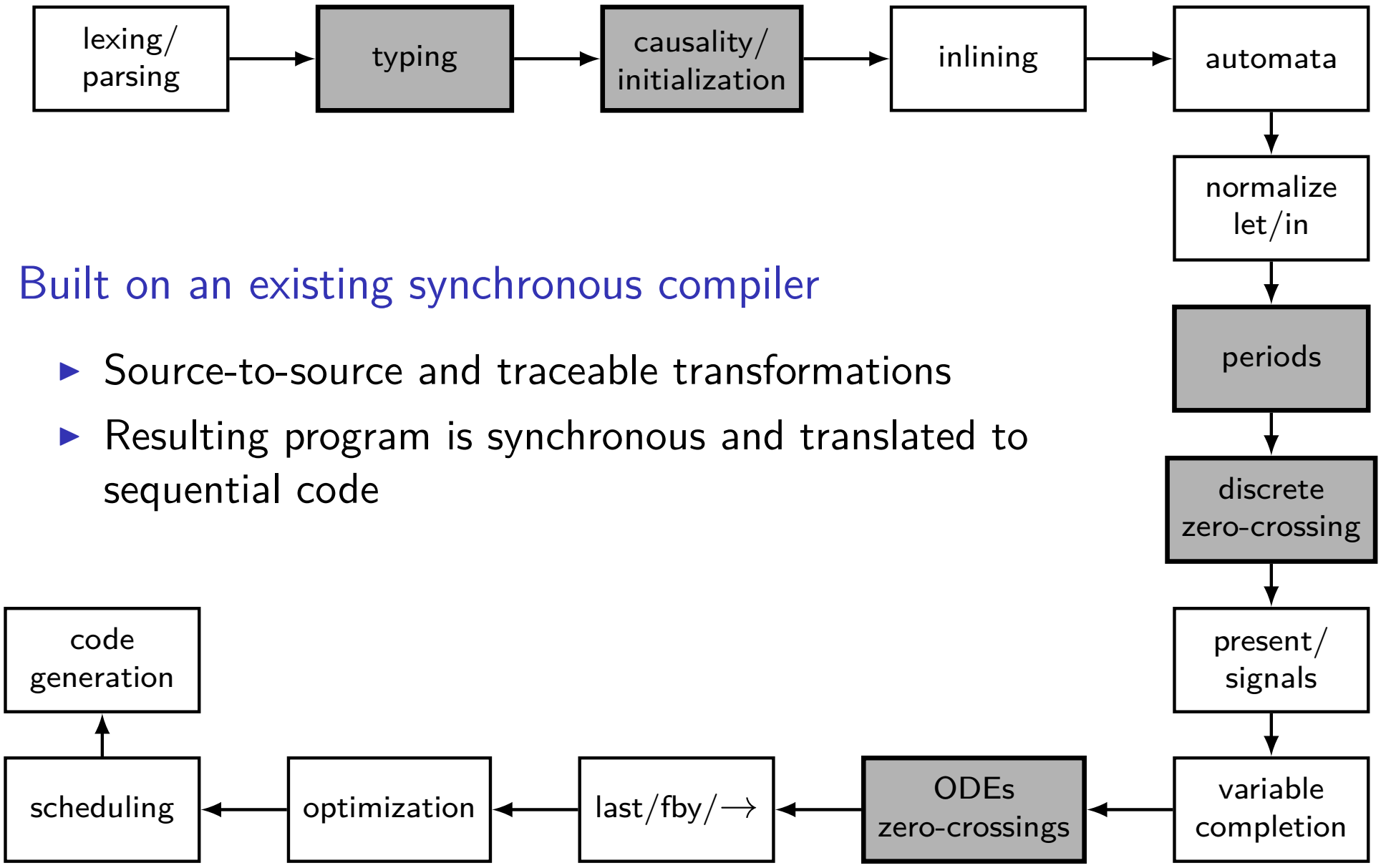
$$\mathsf{last}(x) = \texttt{if}\ d\ \texttt{then}\ \texttt{pre}(x)\ \texttt{else}\ x$$

Add a type $ct_1 + ct_2$ such that $C \mid G, H \vdash_k e : ct_1 + ct_2$ means that:

- During a discrete step, e depends on ct_1;
- During a continuous step e depends on ct_2.

$$\text{(LAST)}\ \frac{C \vdash ct_1' < ct_1}{C, H + [x : ct_1 + ct_2] \vdash \mathsf{last}(x) : ct_1' + ct_2}$$

Compiler architecture



Built on an existing synchronous compiler

- Source-to-source and traceable transformations
- Resulting program is synchronous and translated to sequential code

Comparison with existing tools

Simulink/Stateflow (Mathworks)

- Integrated treatment of automata *vs* two distinct languages
- More rigid separation of discrete and continuous behaviors

Modelica

- Do not handle DAEs
- Our proposal for automata has been integrated into version 3.3

Ptolemy

- More restrictive: A unique computational model (synchronous)
- Everything is compiled to sequential code

Perspectives

Semantics

- Corretness property: well-typed programs do not have discontinuities outside of zero-crossing events.
- This does not ensure the absence of zeno behaviors.
- The synchronous non-standard semantics is useful to prove the correctness theorem.
- Can we do it with the same precision and consision with super-dense time?

DAEs?

- Only ODEs for the moment. DAEs raise several issues (index reduction, etc.)
- Techniques by Acary and Brogliato to model *Non smooth dynamical systems* (e.g., billard balls)