Languages for Programming Hybrid Discrete/Continuous-Time Systems

Marc Pouzet

ENS Paris/UPMC/INRIA

Collège de France, March 26, 2014

In collaboration with Albert Benveniste, **Timothy Bourke**, Benoit Caillaud and Bruno Pagano.

Current Practice

- Hybrid Systems Modelers
- Some issues

Interlude: interacting with a numerical solver

Key elements of our approach

- Non Standard Synchronous Semantics
- Typing
- Causality
- Compilation

Trends for building safe and complex embedded systems

Write executable mathematical specifications in a high-level language so that a model is:

A reference semantics independent of any implementation.

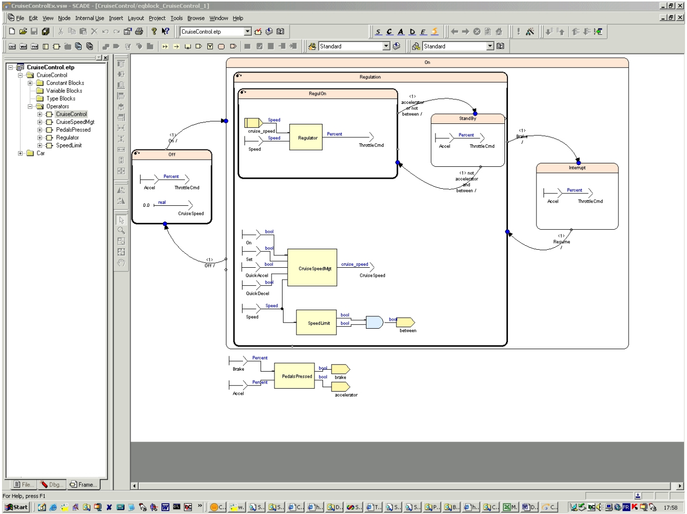
A base for simulation, testing, formal verification.

Then compiled into executable code, sequential or parallel.

A way to achieve correct-by-construction software.

Synchronous Block Diagram Languages[1]

E.g., The Cruise control in SCADE 6 (Esterel-Technologies/ANSYS).



---

[1] Cf. previous courses by Gérard Berry.

A good match for programming discrete-time controllers

Their semantics is simple and mathematically precise:

- Difference equations; hierarchical automata; parallel composition.
- Simulate/test/verify throughout the development process.

Then compiler ensures strong safety properties.

- The program is deterministic.
- The generated code runs in bounded time and memory.

Efficient and fully traceable code generation.

- The code is correct w.r.t the source model.
- Meets the highest quality level of civil avionics (DO178C, level A).[2]

SCADE 6 is used for programming various critical control software.[3]

---

[2]Cf. Seminar by Bruno Pagano, in Spring 2013.

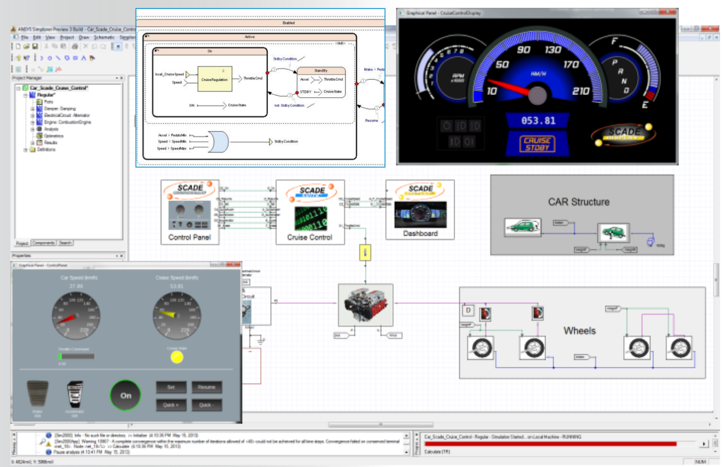
[3]Cd. Seminar by Emmanuel Ledinot, in Spring 2013.

But modern systems need more. . .

The Current Practice of Hybrid Systems Modeling

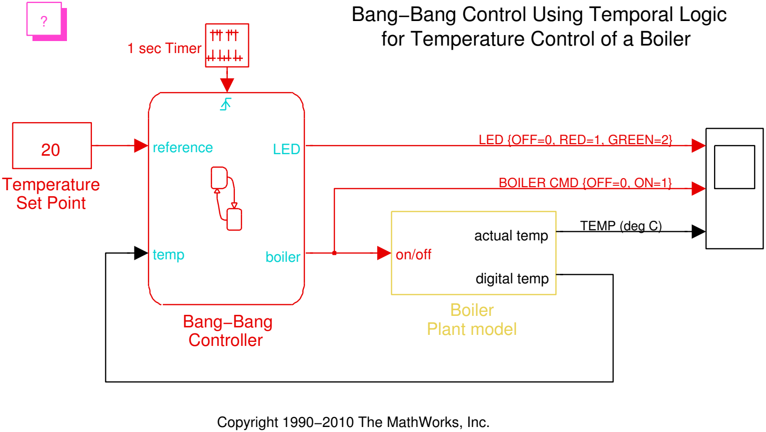
Embedded software interacts with physical devices.

The whole system has to be modeled: the controller and the plant.[4]



[4] Image by Esterel-Technologies/ANSYS.

Example: a Bang-bang controller [demo].



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.

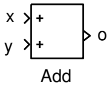
Co-simulation/combination of tools:

Agree on a common format/protocol: FMI/FMU, S-functions, etc.

Convert the model of one tool into another.

Underlying Mathematical Models

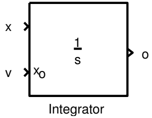
Synchronous parallelism, sequence equations:[5]



Time is discrete and logical (indices in $\mathbb{N}$)

Equation $o = x + y$ means $\forall n \in \mathbb{N}. o(n) = x(n) + y(n)$

Ordinary Differential Equations (ODEs):[6]



Time is continuous (indices in $\mathbb{R}$)

Equation $o = \frac{1}{s}(x)\ \mathit{init}\ v$ means $\forall t \in \mathbb{R}. o(t) = v(0) + \int_0^t x(\tau)\, d\tau$

[5] Cf. Course by Gerard Berry, Spring 2013.

[6] Cf. Seminar by Juliette Leblond, Feb. 2014.

Is there anything left to do?

We know how to build tools for discrete-time models.

We know how to build tools for continuous-time models.

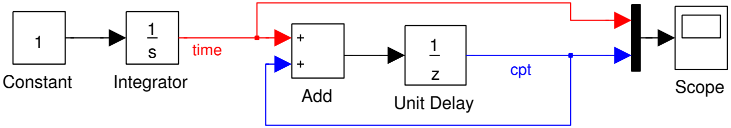
But what if the two are mixed together?

Is it enough to connect one to the other?

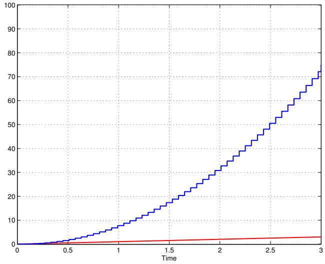
Can you trust code automatically generated from such tools?

Strange beasts. . .

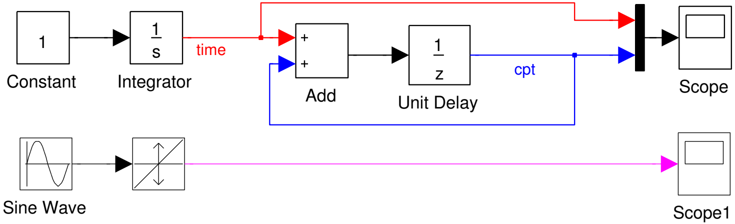
Typing issue 1: Mixing continuous & discrete components



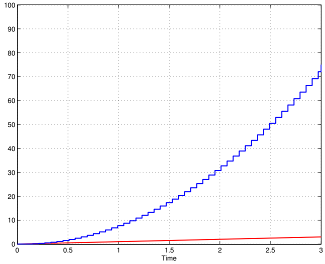
Basic model



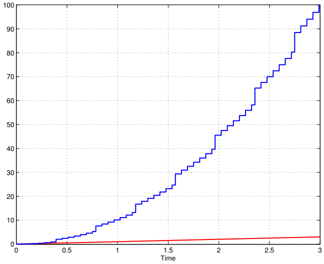
Typing issue 1: Mixing continuous & discrete components



Basic model

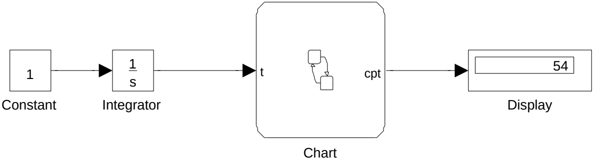


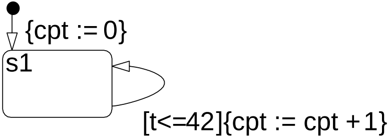
with Sine Wave



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

Typing issue 2: Boolean guards in continuous automata



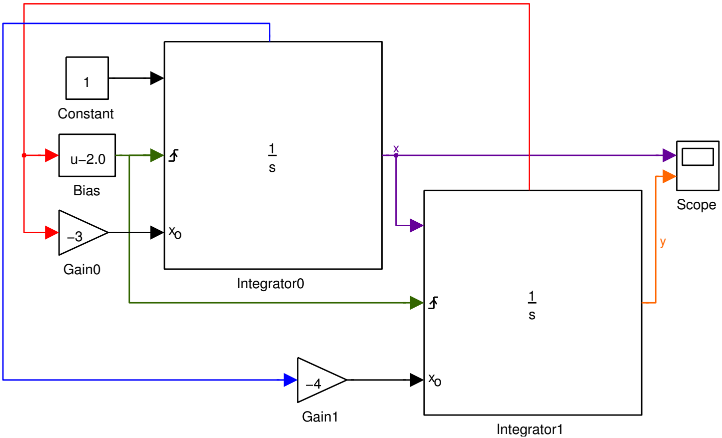


How long is a discrete step?

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

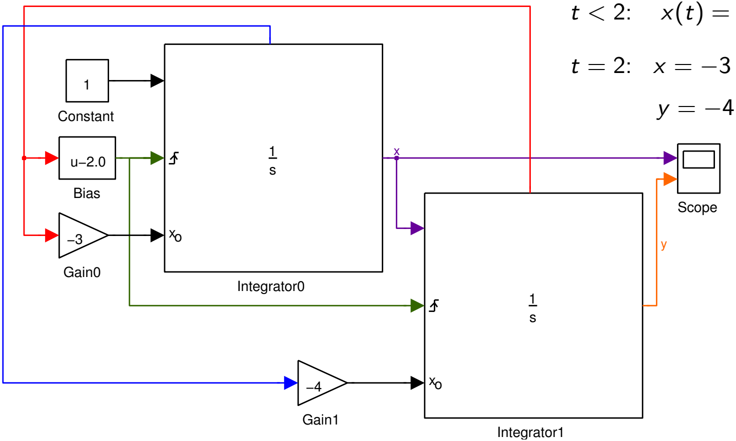
Discrete time is not logical: it is that of the simulation engine.

Causality issue: the Simulink state port



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

Causality issue: the Simulink state port



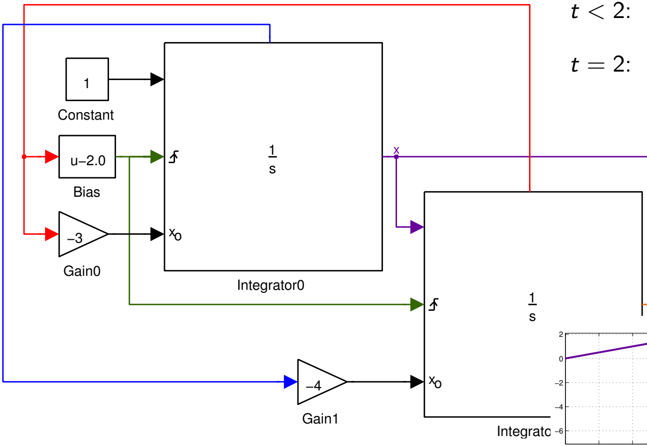
$$t < 2: \quad x(t) = t,\ y(t) = \frac{t^2}{2}$$

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

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

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

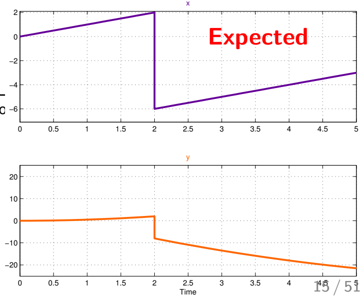
Causality issue: the Simulink state port



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

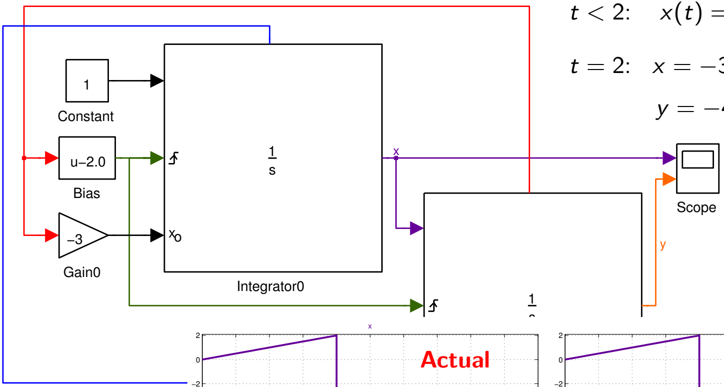
$$t = 2: \quad x = -3 \cdot \text{last}\, y = -6,$$

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



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

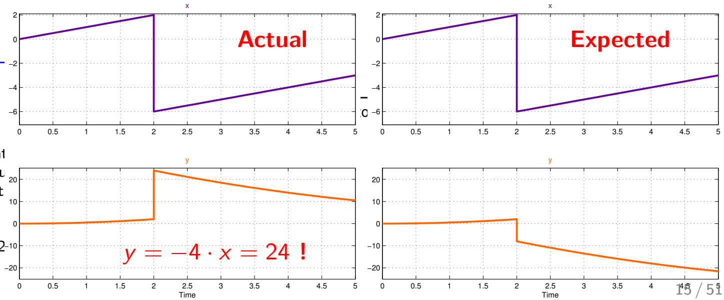
Causality issue: the Simulink state port



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

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

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



The output of the 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.

Current Practice: conclusion

What is the semantics of these tools?

When the manual and implementions diverge, which is right?

There are side effects, global variables, backtracking.

Hard to judge whether the generated code is correct.

What more could we want?

An cleaner integration of discrete and continous time.

Static rejection of bizarre programs.

Current Practice

- Hybrid Systems Modelers
- Some issues

Interlude: interacting with a numerical solver

Key elements of our approach

- Non Standard Synchronous Semantics
- Typing
- Causality
- Compilation

Interacting with a numerical solver

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

All major tools thus calculate approximate solutions numerically.

Numerical solvers (e.g., LLNL Sundials CVODE)

Designed by experts!

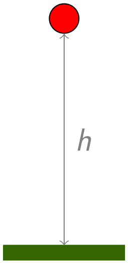
Compute a discrete-time approximations of continuous-time signals.

Subtle: variable step, change order dynamically, explicit/implicit.

Define compilation schemes with solver's internals kept abstract.

Bouncing ball

model



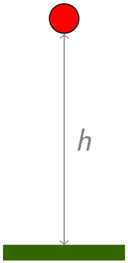
$$F = m \cdot a$$

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

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

Bouncing ball

model



$$F = m \cdot a$$

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

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

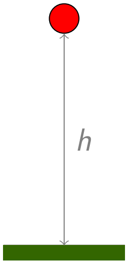
$$\dot{v} = -g \qquad v(0) = v_0$$

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

First-order ODE

Bouncing ball

model



$$F = m \cdot a$$

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

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

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

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

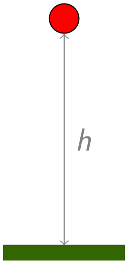
First-order ODE

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

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

Bouncing ball

model



$$F = m \cdot a$$

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

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

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

Solver

approximation

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

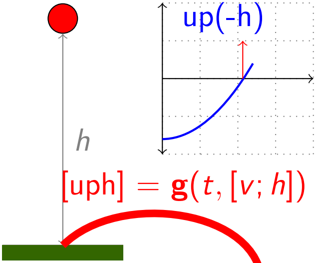
$$\boxed{\begin{matrix}\dot{v}\\ \dot{h}\end{matrix}}\begin{matrix} = -g \\ = v\end{matrix} \qquad \boxed{\begin{matrix}v(0)\\ h(0)\end{matrix}}\begin{matrix} = v_0 \\ = h_0\end{matrix}$$

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

First-order ODE

Bouncing ball

model



$$F = m \cdot a$$

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

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

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

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

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

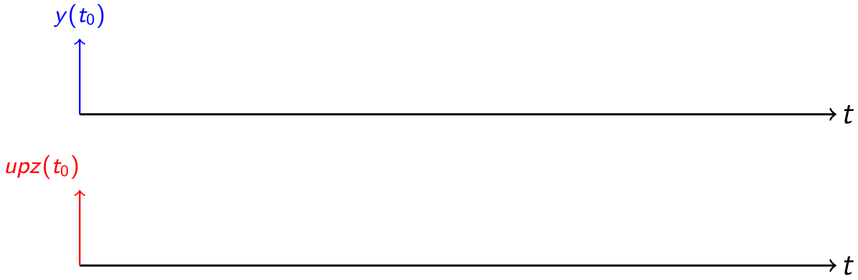
Solver execution (e.g., LLNL Sundials CVODE)

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 or g

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

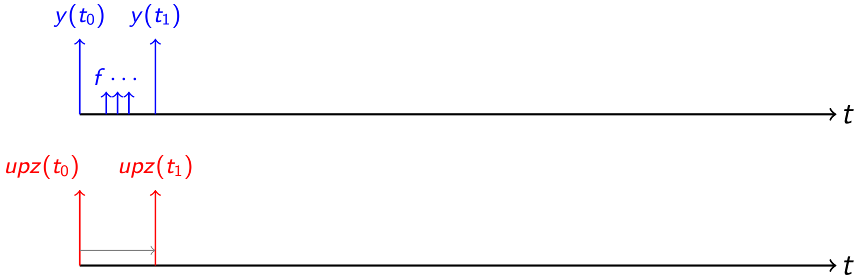
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

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

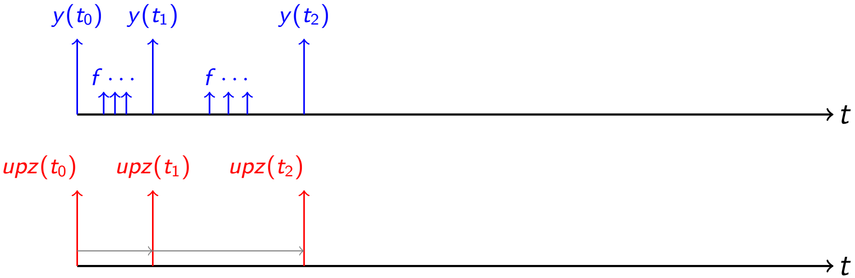
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

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

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

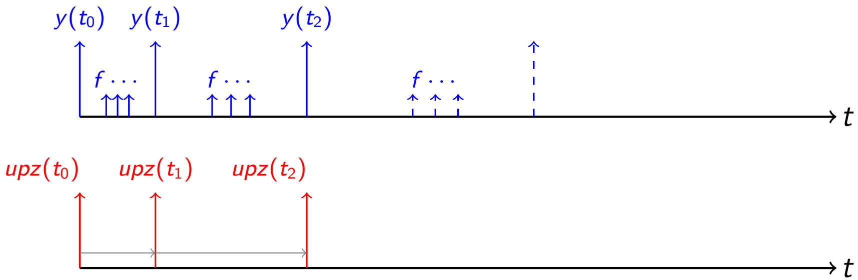


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
    - No side-effects within f or g

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

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

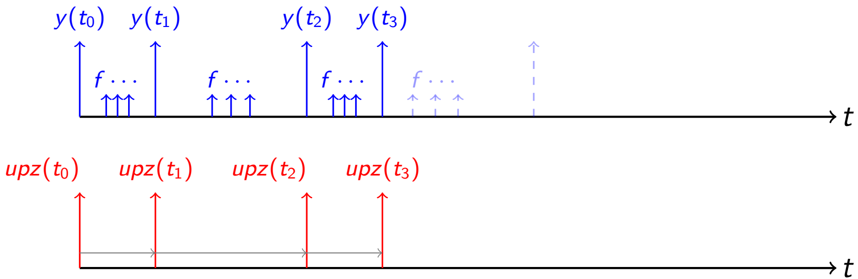
approximation error too large



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

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

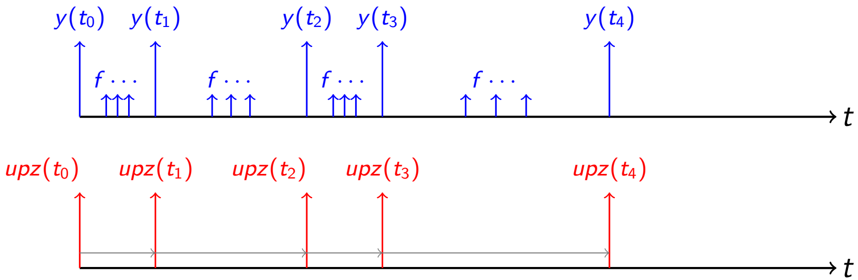
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

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

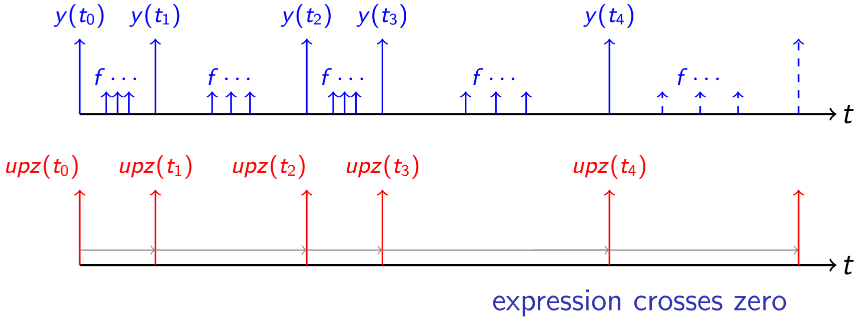
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

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

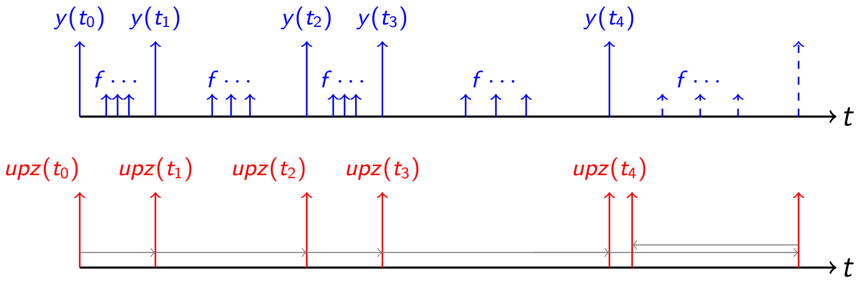
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

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

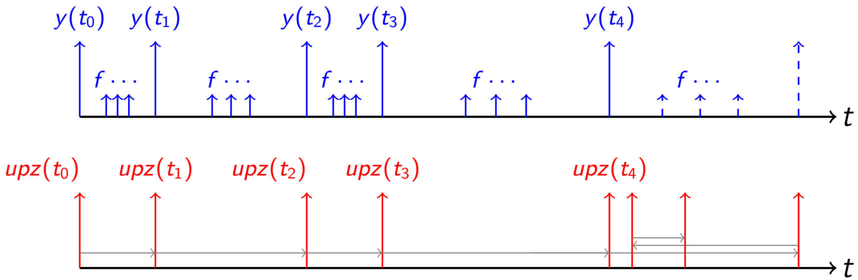
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

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

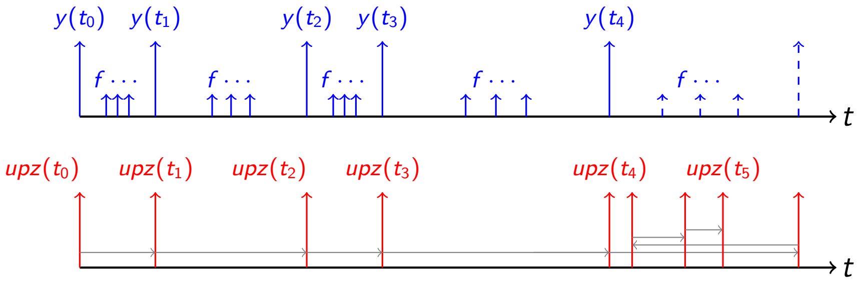
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

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

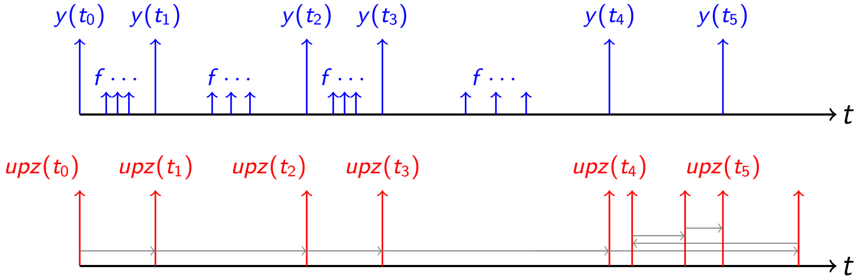
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

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

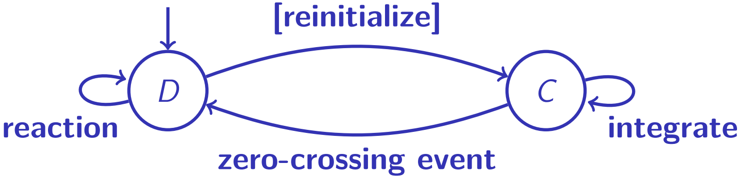
Give solver two functions: $\dot{y} = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

The Simulation Engine of Hybrid Systems

Alternate discrete steps and integration steps

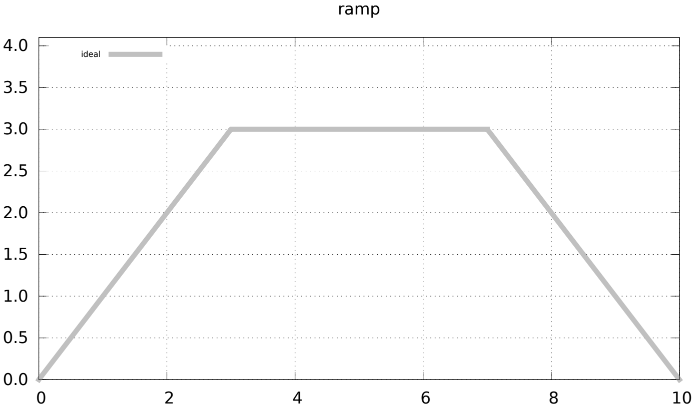


$$\sigma', y' = d_\sigma(t, y) \qquad upz = g_\sigma(t, y) \qquad \dot{y} = f_\sigma(t, y)$$

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.

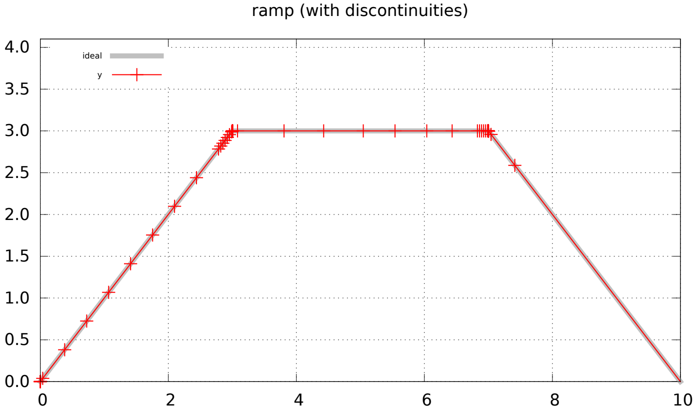
Numerical Integration (Sundials CVODE)



$$\dot{y}(t) = \begin{cases} 1 & \text{if } t < 3 \\ 0 & \text{if } 3 \leq t \leq 7 \\ -1 & \text{if } 7 < t \end{cases}$$

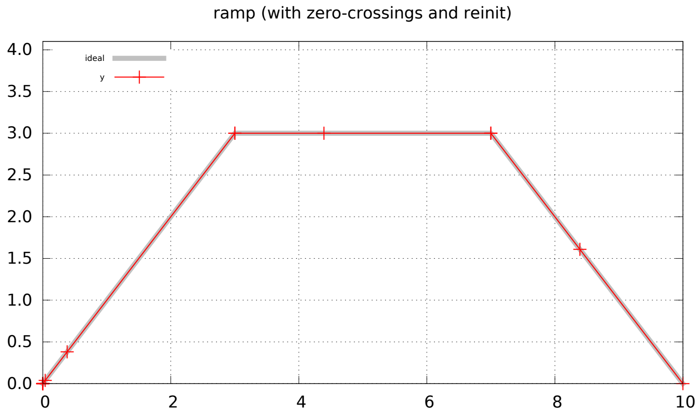
$$y(0) = 0$$

Numerical Integration: a derivative with a discontinuity




```
let f (t, y) =
  if t < 3.0 then 1.0
  else if t <= 7.0 then 0.0
  else −1.0
```

Numerical Integration: discrete state with three modes



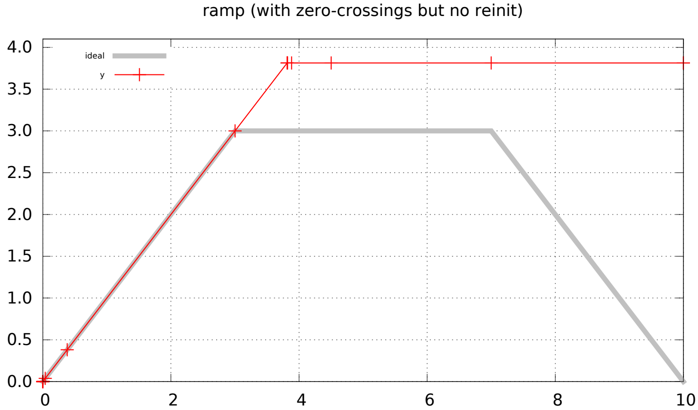

```
let f (discrete_state) (t, y) =
  match discrete_state with
  | RampingUp → 1.0
  | Flat → 0.0
  | RampingDown → −1.0

let g (discrete_state) (t, y) =
  match discrete_state with
  | RampingUp → t −. 3.0
  | Flat → t −. 7.0
  | _ → t
```

```
let discrete_state_i = RampingUp

let d (discrete_state) z (t, y) =
  match discrete_state with
  | RampingUp when z → Flat
  | Flat when z → RampingDown
  | s → s
```

Numerical Integration: with no reinit of the solver



Current Practice

- Hybrid Systems Modelers
- Some issues

Interlude: interacting with a numerical solver

Key elements of our approach

- Non Standard Synchronous Semantics
- Typing
- Causality
- Compilation

Key elements of our approach

Build a hybrid modeler on top of a synchronous language.

Use synchronous constructs for arbitray mix of discrete and continuous.

Divide and Recycle

Recycle existing synchronous languages techniques.

Semantics, static checking, code-generation.

Divide from the code what is for the solver.

Simulate with off-the-shelf numerical solvers.

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

These elements are experimented within the language `Zélus`.

Zélus

zelus.di.ens.fr

Combinatorial and sequential functions

A signal is a sequence of values. 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 fed into the compiler, we get:

val add : int $\times$ int $\xrightarrow{\mathtt{A}}$ int
val min_max : $\alpha \times \alpha \xrightarrow{\mathtt{D}} \alpha \times \alpha$
val after : int $\times$ bool $\xrightarrow{\mathtt{D}}$ bool

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

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:

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

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

Examples

Up to syntactic details, these are `Scade 6` or `Lucid Synchrone` programs. E.g., a simple heat controller.[7]

```
(* 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)
```

[7] 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 [demo]

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

Its type signature is:

val bouncing : float $\times$ float $\times$ float $\times$ float $\xrightarrow{\mathtt{C}}$ float $\times$ float

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

Summary of Programming Constructs

- Synchronous constructs: data-flow equations/automata.
- An ODE with initial condition: der x = e init e0
- last x is the left limit of x.
- Detect a zero-crossing (from negative to positive): up(x).
- This defines a discrete instant, that is, an event.
- All discrete changes must occur on an event. E.g.,:

```
let hybrid f(x, y) = (v, z1, z2) where
  rec v = present z1 -> 1 | z2 -> 2 init 0
  and z1 = up(x)
  and z2 = up(y)
```

val f : float $\times$ float $\xrightarrow{C}$ int $\times$ zero $\times$ zero

- If x = up(e), all handlers using x are governed by the same event.

Three difficulties

Semantics

- An ideal semantics to say which program make sense;
- useful to prove that compilation is correct.

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 bizarre behaviors seen previously.

Ensure that fix-points exist and code can be scheduled.

- Algebraic loops must be statically detected.
- Restrict the use of last x so that signals are proved to be continuous during integration.

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

We proposed 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 $t \in \mathbb{R}^+$:

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

What would be the value of y if it were 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.

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

Non standard semantics [JCSS'12]

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: $\mathtt{der}\ x = e\ \mathtt{init}\ e_0\ \mathtt{reset}\ z \longrightarrow e_1$

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

$\mathtt{last}\ x$ if x is defined on clock T

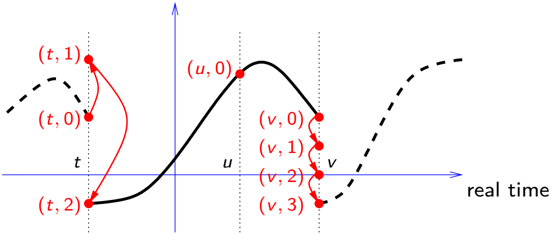
$$
{}^\star\mathtt{last}\ x(t) = {}^\star x({}^\bullet T(t))
$$

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

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

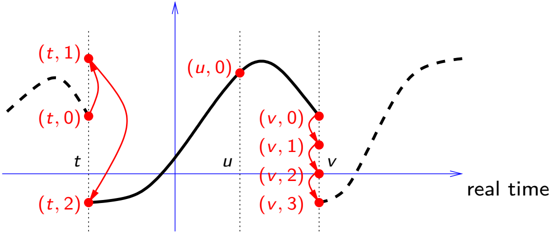
Non-standard time vs. Super-dense time

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

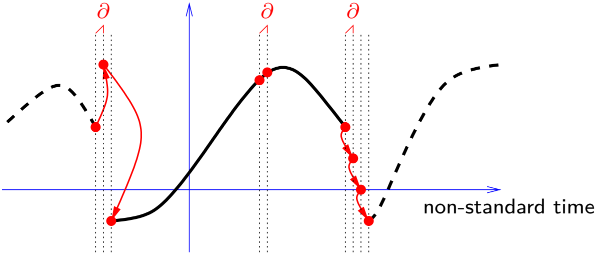


Non-standard time vs. Super-dense time

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



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



Typing: mixing discrete (logical) time and continuous time

The following two parallel composition make sense.

Discrete time: the clock should be discrete

```
let node sum(x) = cpt where
  rec cpt = 0 → pcpt
  and pcpt = pre(cpt) + x
```

Continuous time: the clock should be continuous

```
let hybrid bouncing(y0, y'0) = o where
  rec der y = y' init y0
  and der y' = −.g init y'0
  and o = y +. 10.0
```

The following do not make sense

At what clock should we compute cpt?

```
rec der t = 1.0 init 0.0
and cpt = 0.0 → pre(cpt) + t
```

Intuition

Distinguish functions with three kinds A/D/C.

- Combinatorial function get kind A (for “any”).
- Discrete-time (synchronous) functions get kind D (for “discrete”).
- Continuous-time (hybrid) functions get kind C (for “continuous”).

Explicitly relate simulation and logical time

All discontinuities and side effects must be aligned with a zero-crossing instant.

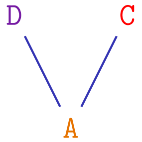
```
let hybrid correct (z) = (time, y) where
  rec der time = 1.0 init 0.0
  and y = present up(z) → sum(time) init 0.0
```

Basic typing [LCTES'11]

A simple ML type system with effects.

The type language

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



Initial conditions

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

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" on page 16-26
"Summary of Rules for Continuous-Time Modeling" on page 16-26

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:

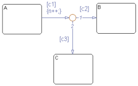
Update local data *only* in transition, `entry`, and `exit` actions

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:

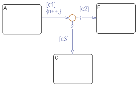
Update local data *only* in transition, `entry`, and `exit` actions

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 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.

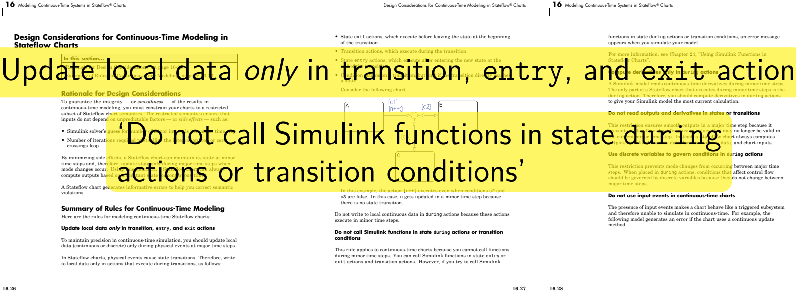
. . .

'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 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.

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

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".

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

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'

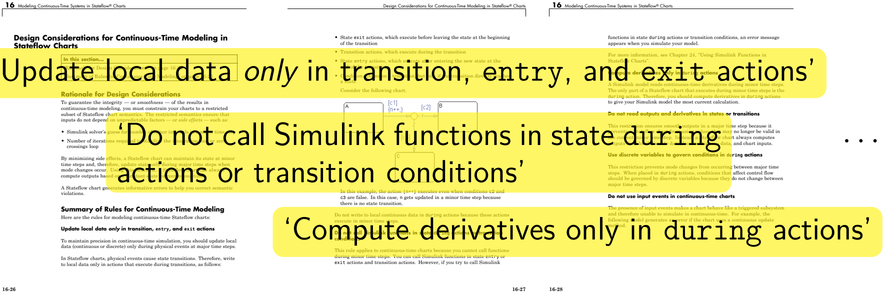
'Do not call Simulink functions in state `during` actions or transition conditions'

...

- '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.



'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.

'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 loops

Yet, some program 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 if sequential code generation is targeted.
- `last x` does not necessarily break causality loops!

  ```
  rec x = last x + 1
  ```

- OK

  ```
  rec der x = 1.0 init 0.0 reset z → t
  and y = x +. 1.0
  and t = last y
  ```

Can we find a simple and uniform justification?

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:

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

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 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.

```
der y = 1.0 init 0.0 reset up(last y -. 1.0) → 0.0
```

An explanation of the bug

The source program

```
rec der x = 1.0 init 0.0 reset z → −3.0 *. last y
and der y = x init 0.0 reset z → −4.0 *. last x
and z = up(last x −. 2.0)
```

Its non-standard interpretation

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

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.

Causality Analysis [HSCC'14]

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.

Associate a type that express input/output dependences. E.g.,

```
let node plus(x, y) = x + 0 → pre y
```

We get: $f : \forall\alpha_1, \alpha_2.\alpha_1 \times \alpha_2 \xrightarrow{\mathtt{D}} \alpha_1$

- der x breaks a loop: der temp = c −. temp init 20.0 is correct.
- last(x) breaks a loop in a discrete context.

The following is rejected; the next is accepted.

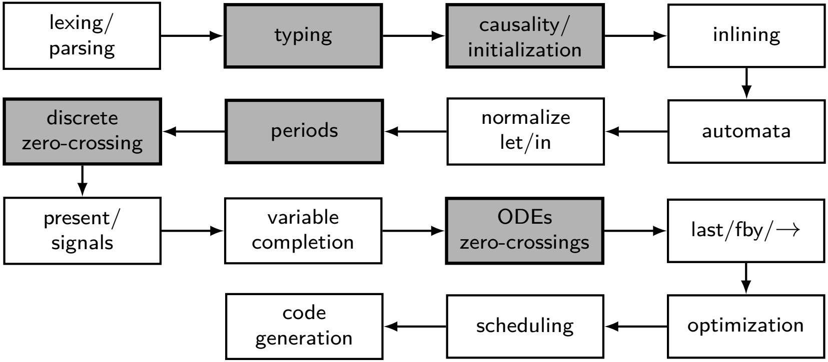
```
rec der y' = −. g init 0.0 reset up(−.y) → −0.9 *. y'
and der y = y' init y0

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

Major theorem: [HSCC 14] Well typed programs define continuous signals during integration.

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

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 (E.A. Lee et al., Berkeley)

- A unique computational model: synchronous
- Everything is compiled to sequential code (not interpreted)