Zélus, a Synchronous Language with ODEs

Marc Pouzet[2,3,1] Albert Benveniste[1]

Timothy Bourke[1,2] Benoît Caillaud[1]

1. INRIA
2. École normale supérieure (DI)
3. Univ. Pierre et Marie Curie

Action d'envergure SYNCHRONICS, Final review, 18 June 2013, Paris

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:

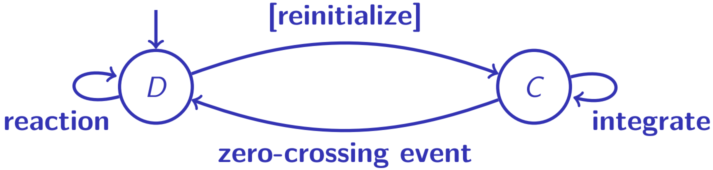
Any synchronous program must be compiled, optimized, and executed as per usual

Type systems to separate continuous from discrete

What is a discrete step?

- Reject unreasonable parallel compositions
- Ensure by static typing that discrete changes occur on zero-crossings
- Signals are continuous during integration
- Statically detect causality loops, initialization issues

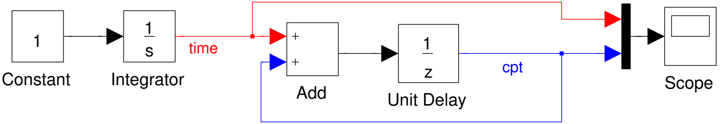
Simulation engine

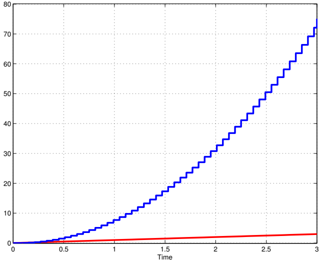


$$\sigma' = d_\sigma(t, y) \qquad upz = g_\sigma(t, y) \qquad \dot{y} = f_\sigma(t, y)$$

Strange beasts. . .

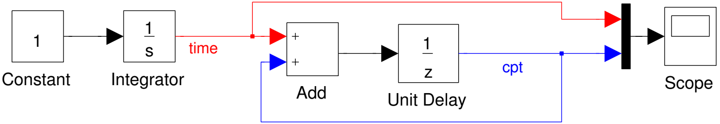
Typing issue: Mixing continuous and discrete components

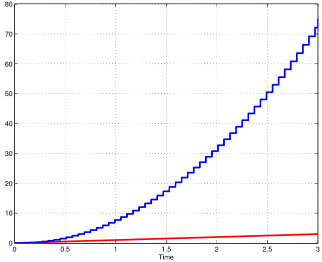




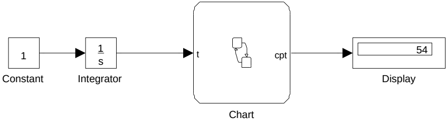
- Warning with ‘Unit Delay’ but not with ‘Memory’.
- The shape of cpt depends on the steps chosen by the solver.
- Putting another component in parallel can change the result.

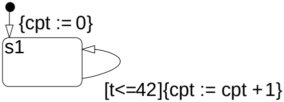
Typing issue: Mixing continuous and discrete components



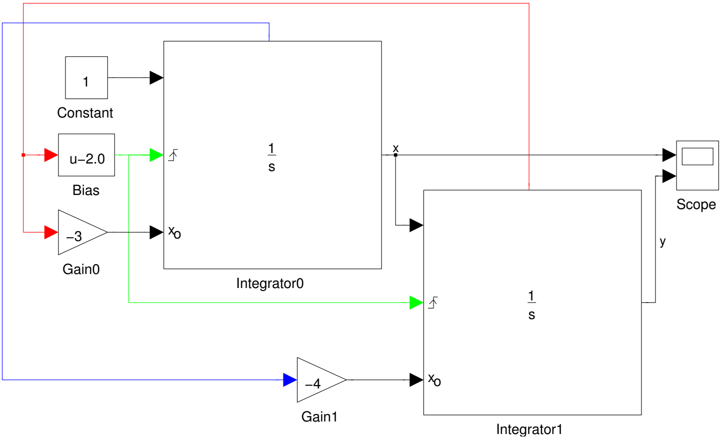


- Warning with 'Unit Delay' but not with 'Memory'.
- The shape of cpt depends on the steps chosen by the solver.
- Putting another component in parallel can change the result.
- Similar issues with Stateflow.



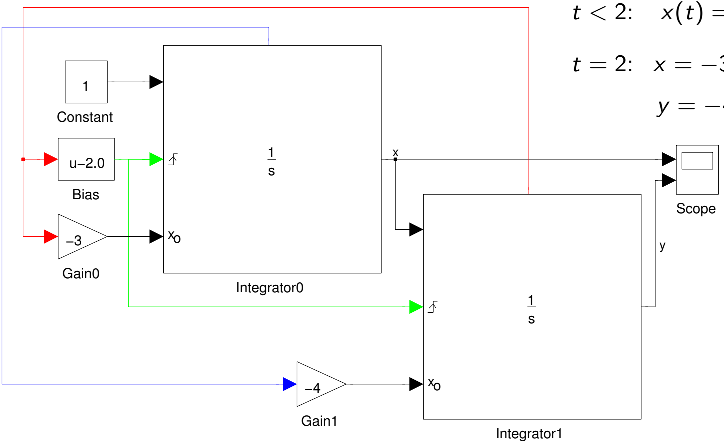


Causality issue: the Simulink state port



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



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

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

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

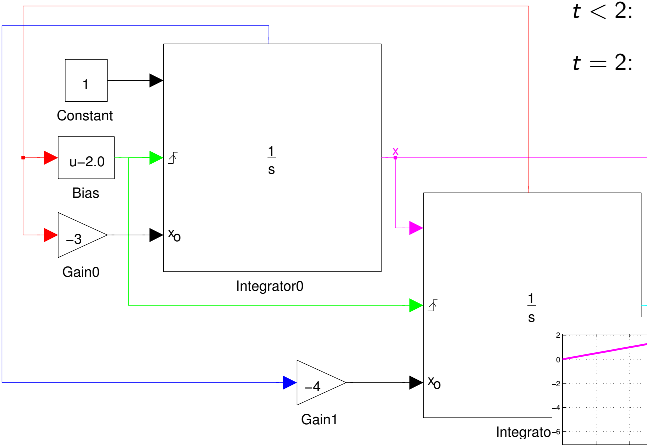
The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.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

Causality issue: the Simulink state port

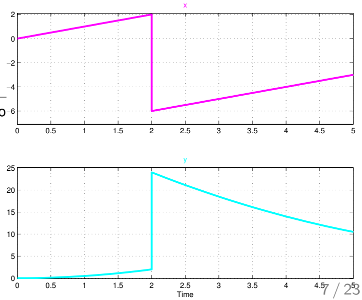
$$t < 2: \quad x(t) = t,\ y(t) = \frac{t^2}{2}$$

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

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



But: $y = -4 \cdot x = 24$!



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)

Zélus

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 $\times$ int $\xrightarrow{\mathtt{A}}$ int

val mix_max : $\alpha \times \alpha \xrightarrow{\mathtt{D}} \alpha \times \alpha$

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

Here x, y, etc. are sequences.

The counter can be instantiated as 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:

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

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

Examples

Up to syntactic details, these programs could have been written *as is* in `Scade 6` or `Lucid Synchrone`. Now, a simple heat controller with ODEs.[1]

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

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

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

[1] 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'
```

Its type signature is:

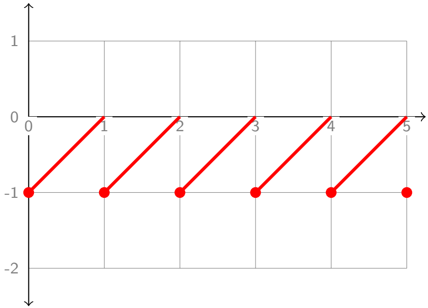
$$\text{float} \times \text{float} \times \text{float} \xrightarrow{\texttt{C}} \text{float} \times \text{float}$$

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

ODEs and Zero-crossings

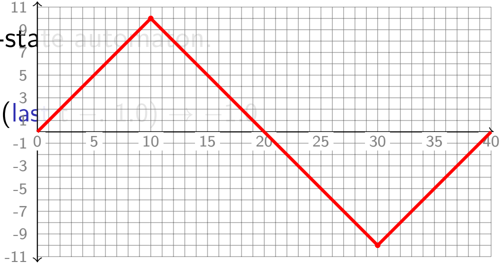
E.g., the sawtooth signal, the two-state automaton.

```
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
```

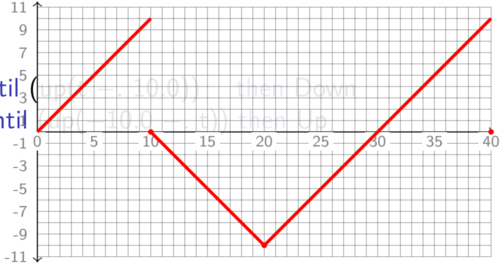
```
let hybrid fm'() = t where
  rec init t = 0.0
  and automaton
      | Up → do der t = 1.0
               until (up(t −. 10.0)) then do t = last t −. 10.0 in Down
      | Down → do der t = −1.0 until (up(−10.0 −. t)) then Up
```

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
```

Zero-crossings and Valued Signals

- $\mathtt{up}(e)$ tests the zero-crossing of expression e from strictly negative to strictly positive.
- Performed by the solver during integration.
- If $x = \mathtt{up}(e)$, all handlers using x are governed by the same zero-crossing.
- Handlers have priorities.

```
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 $\xrightarrow{\mathtt{C}}$ float $\times$ zero $\times$ zero

Valued events and left limit

Emit a value on a zero-crossing

```
let hybrid f(x, y) = o where
  rec o = present (up(x)) → 42 | (up(y) → 43

val f: float −C→ int signal
```

o is only present when either $up(x)$ or $up(y)$ and it carries an integer value.

```
let hybrid default(x, x0) = o where
  rec o = present x(p) → p init x0

val f: int signal −C→ int
```

The left limit

last(x) is the "previous" value of x. It coincides with the left-limit of x.

- During integration, last(x) $\approx x$ (same standard part).
- During a discrete step, last(x) is the previous value of x.

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:

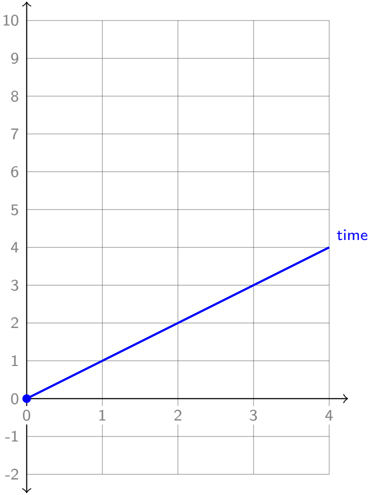
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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:

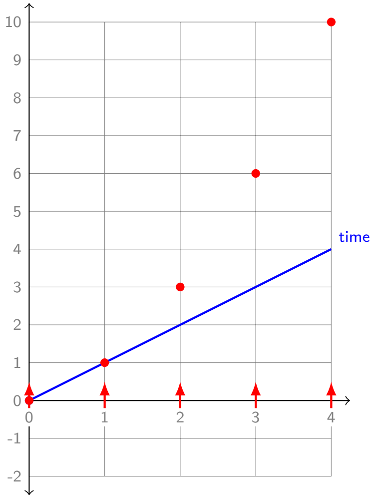
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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



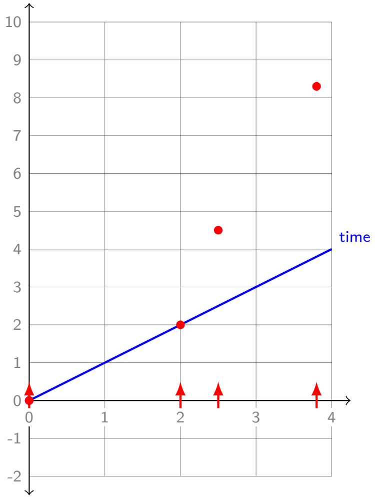
Mixing discrete (logical) time and continuous time

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

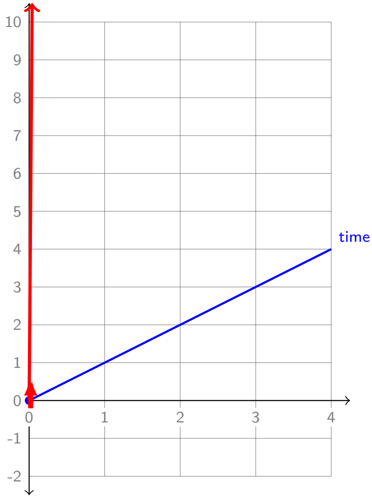
Mixing discrete (logical) time and continuous time

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:

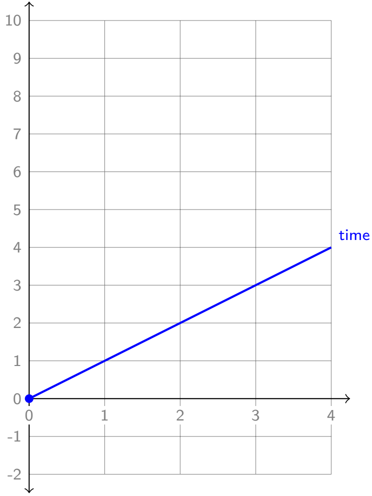
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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

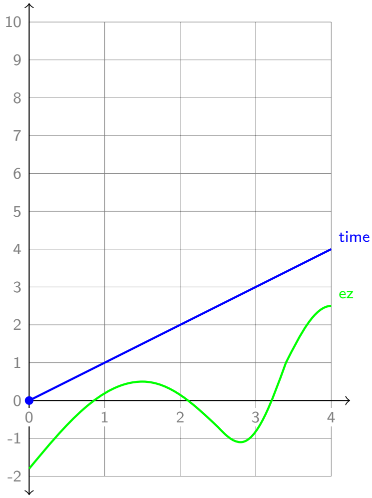
Given:

```
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



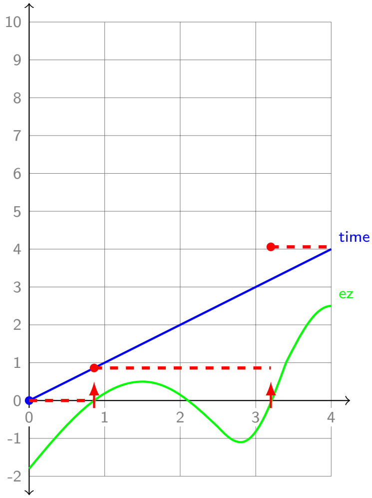
Mixing discrete (logical) time and continuous time

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 system with effects.

The type language

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

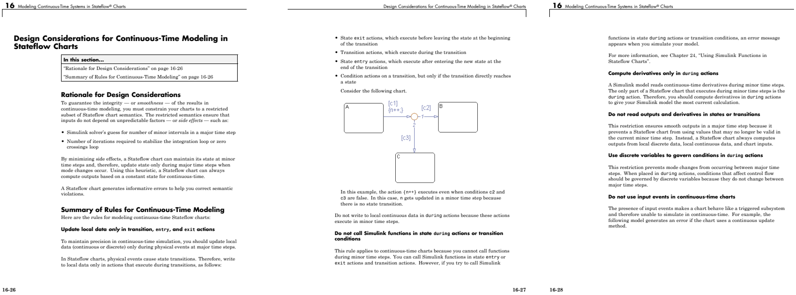
$$
\begin{tikzcd}
\texttt{D} \arrow[dr, dash] & & \texttt{C} \arrow[dl, dash] \\
& \texttt{A} &
\end{tikzcd}
$$

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.



16 Modeling Continuous-Time Systems in Stateflow® Charts

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:

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 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

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 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.

16-28

...

- '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:

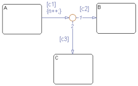
Update local data *only* in transition, `entry`, and `exit` actions

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.

. . .

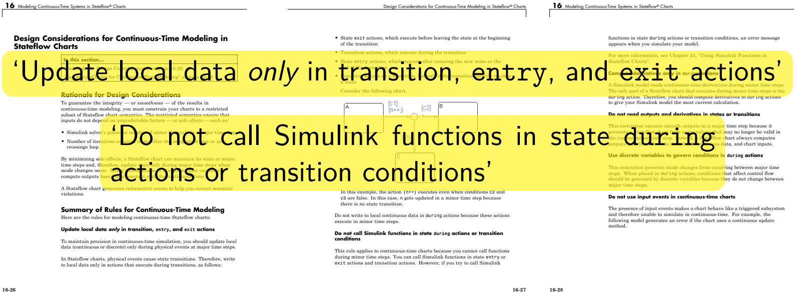
'Update local data *only* in transition, `entry`, and `exit` actions'

- '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.



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

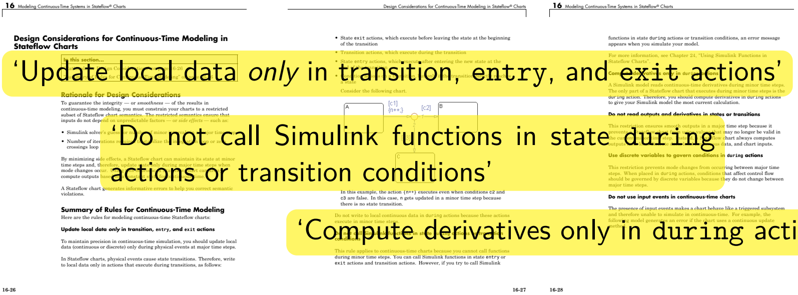
'Do not call Simulink functions in state `during` actions or transition conditions'

...

- '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'

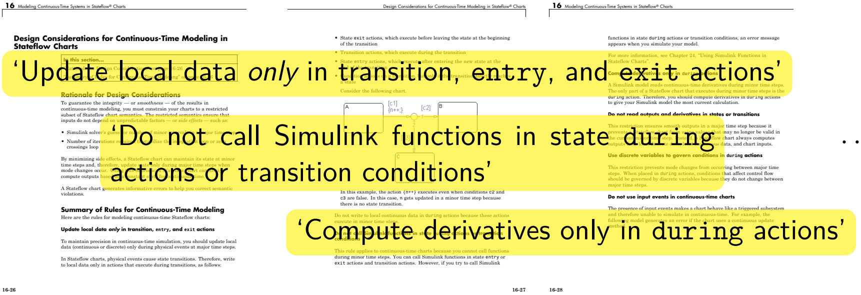
'Compute derivatives only in during actions'

...

- '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.



…

- '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
```

- OK in constructive logic (Esterel)

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

- Modularity:

```
let node gonthier(x,y) = (x, y)
let node feedback(x) = y where
  rec (z, y) = gonthier(x, z)
```

At the moment, we stick to a simple Lustre-like solution:

every feedback loop must cross a delay

Yet, what is a delay in mixed systems?

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

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

We get: $f : \forall \alpha_1, \alpha_2 . \alpha_1 \times \alpha_2 \rightarrow \alpha_1$

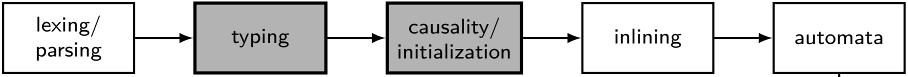
- pre(x) is a, discrete-time only, unit delay.
- der x breaks a loop: der temp = c −. temp init 20.0 is correct.
- last(x) is the left limit of a signal:
  - when x is a continuous-state variable (der x = ...), this is the `Simulink` state port.
  - writting last x in a discrete context always make sense.

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
```

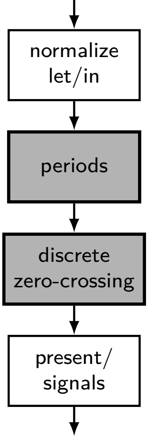
```
rec der y' = −. g init 0.0 reset up(−.y) → −0.9 *. last y'
and der y = y' init y0
```

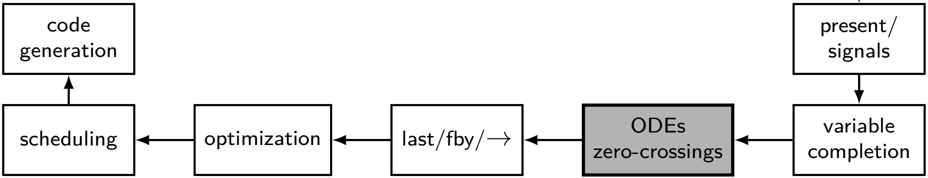
Compiler architecture



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)

What next?

Typing, Causality analysis, Optimization

- The current type system is very limited: if x and y are integers, $x = y$ is rejected in a hybrid node.
- Share states and zero-crossings, as much as possible.

DAEs

- Only ODEs for the moment.
- DAEs raise several issues: index reduction, etc.