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

\[
\begin{align*}
\sigma' &= d_\sigma(t, y) \\
upz &= g_\sigma(t, y) \\
\dot{y} &= f_\sigma(t, y)
\end{align*}
\]
Strange beasts...
Typing issue: Mixing continuous and discrete components

- Warning with ‘Unit Delay’ but not with ‘Memory’.
- The shape of \( \text{cpt} \) 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 \( \text{cpt} \) 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)
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

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)

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

\[ t = 2: \quad x = -3 \cdot \text{last} \ y = -6, \quad y = -4 \cdot \text{last} \ x = -8 \]

But: \( y = -4 \cdot x = 24 ! \)
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.

```ocaml
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:

```ocaml
val add : int × int → int
val mix_max : α × α → α × α
val after : int × int → 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:

```markdown
val blink : int \(\times\) int \(\times\) int \(\rightarrow\) bool
val blink_reset : int \(\times\) int \(\times\) int \(\times\) int \(\rightarrow\) 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.\footnote{This is the hybrid version of one of Nicolas Halbwachs’ examples with which he presented Lustre at the Collège de France, in January 2010.}

\[
\begin{align*}
\text{(*an hysteresis controller for a heater *)} \\
\text{let hybrid heater(active) = temp where} \\
& \quad \text{rec der temp = if active then } c -. \text{ temp else } -. \text{ temp init temp0} \\
\text{let hybrid hysteresis_controller(temp) = active where} \\
& \quad \text{rec automaton} \\
& \quad \mid \text{Idle } \to \text{ do active = false until (up(t_min -. \text{ temp})) then Active} \\
& \quad \mid \text{Active } \to \text{ do active = true until (up(temp -. t_max)) then Idle} \\
\text{let hybrid main() = temp where} \\
& \quad \text{rec active = hysteresis_controller(temp)} \\
& \quad \text{and temp = heater(active)}
\end{align*}
\]
The Bouncing ball

let hybrid bouncing\((x_0,y_0,x'0,y'0) = (x,y)\) where
\[
\begin{align*}
der(x) &= x' \text{ init } x_0 \\
der(x') &= 0.0 \text{ init } x'0 \\
der(y) &= y' \text{ init } y_0 \\
der(y') &= \text{-. } g \text{ init } y'0 \text{ reset up}(\text{-. } y) \rightarrow \text{-. } 0.9 \ast \text{ last } y'
\end{align*}
\]

Its type signature is:

\[
\text{float } \times \text{ float } \times \text{ float } \rightarrow \text{ float } \times \text{ float}
\]

- When \text{-. } y \text{ crosses zero, re-initialize the speed } y' \text{ with } \text{-. } 0.9 \ast \text{ last } y'\).
- \text{last } y' \text{ stands for the previous value of } y'.
- As } y' \text{ is immediately reset, writing } \text{last } y' \text{ is mandatory} —otherwise, } y' \text{ 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)

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.

\[
\text{let hybrid sawtooth}() = t \text{ where}
\begin{align*}
\text{rec } \text{der } t &= 1.0 \text{ init } -1.0 \text{ reset } \text{up(last t } - . \text{ 1.0) } \rightarrow -1.0
\end{align*}
\]

\[
\text{let hybrid fm}() = t \text{ where}
\begin{align*}
\text{rec } \text{init } t &= 0.0 \\
\text{and automaton}
\begin{align*}
| \text{Up } \rightarrow & \text{ do der } t = 1.0 \text{ until } (\text{up(t } - . \text{ 10.0)}) \text{ then Down} \\
| \text{Down } \rightarrow & \text{ do der } t = -1.0 \text{ until }
\end{align*}
\]

\[
\text{let hybrid fm'}() = t \text{ where}
\begin{align*}
\text{rec } \text{init } t &= 0.0 \\
\text{and automaton}
\begin{align*}
| \text{Up } \rightarrow & \text{ do der } t = 1.0 \\
& \text{ until (up(t } - . \text{ 10.0)) then do t = last t } - . \text{ 10.0 in Down}
| \text{Down } \rightarrow & \text{ do der } t = -1.0 \text{ until (up(-10.0 } - . \text{ t)) then Up}
\end{align*}
\]
Zero-crossings and Valued Signals

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

```plaintext
let hybrid f(x, y) = (v, z1, z2) where
  rec v = present z1 \rightarrow 1 \mid z2 \rightarrow 2 init 0
  and z1 = up(x)
  and z2 = up(y)

val f : float \times float \rightarrow 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 → 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 → int

The left limit

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

- During integration, last(x) ≈ x (same standard part).
- During a discrete step, last(x) is the previous value of x.
Mixing discrete (logical) time and continuous time

Given:

\[
\text{let node } \text{sum}(x) = \text{cpt where}
\]

\[
\text{rec cpt} = (0.0 \ fby \ \text{cpt}) +. \ x
\]
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:

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

Define:

```plaintext
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:

\[
\text{let node } \text{sum}(x) = \text{cpt where}
\]
\[
\text{rec } \text{cpt} = (0.0 \text{ fby } \text{cpt}) +. x
\]

Define:

\[
\text{let } \begin{align*}
\text{wrong}() &= () \\
\end{align*}
\]
\[
\text{where rec}
\]
\[
\begin{align*}
\text{der } \text{time} &= 1.0 \text{ init } 0.0 \\
\text{and } y &= \text{sum}(\text{time})
\end{align*}
\]

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:

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

Define:

```plaintext
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:

\[
\text{let node sum}(x) = \text{cpt where} \\
\text{rec cpt} = (0.0 \text{ fby cpt}) +. x
\]

Define:

\[
\text{let wrong}() = () \\
\text{where rec} \\
\text{der time} = 1.0 \text{ init} 0.0 \\
\text{and} y = \text{sum}(\text{time})
\]

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:

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

Define:

```plaintext
let hybrid correct () = ()
  where rec
    der time = 1.0 init 0.0
    and y = present up(ez) \rightarrow 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:

\[
\text{let node sum}(x) = \text{cpt where} \\
\text{rec cpt} = (0.0 \ fby \ \text{cpt}) +. \ x
\]

Define:

\[
\text{let hybrid correct} () = () \\
\text{where rec} \\
\text{der time} = 1.0 \ \text{init} \ 0.0 \\
\text{and y} = \text{present up}(\text{ez}) \rightarrow \text{sum (time)} \ \\
\text{init} \ 0.0
\]

- **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

\[
\begin{align*}
\text{bt} & ::= \text{float} | \text{int} | \text{bool} | \text{zero} \\
\text{t} & ::= \text{bt} | \text{t} \times \text{t} | \beta \\
\sigma & ::= \forall \beta_1, \ldots, \beta_n. \text{t} \xrightarrow{k} \text{t} \\
k & ::= D | C | A
\end{align*}
\]

Initial conditions

\[
\begin{align*}
(+) & : \text{int} \times \text{int} \xrightarrow{A} \text{int} \\
\text{if} & : \forall \beta. \text{bool} \times \beta \times \beta \xrightarrow{A} \beta \\
(=) & : \forall \beta. \beta \times \beta \xrightarrow{D} \text{bool} \\
\text{pre}() & : \forall \beta. \beta \xrightarrow{D} \beta \\
\cdot \text{fby} \cdot & : \forall \beta. \beta \times \beta \xrightarrow{D} \beta \\
\text{up}() & : \text{float} \xrightarrow{C} \text{zero} \\
\cdot \text{on} \cdot & : \text{zero} \times \text{bool} \xrightarrow{A} \text{zero}
\end{align*}
\]
What about continuous automata? [EMSOFT’11]

Stateflow User’s Guide

The Mathworks, pages 16-26 to 16-29, 2011.

Modeling Continuous-Time Systems in Stateflow Charts

Design Considerations for Continuous-Time Modeling 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 16-30

Rationale for Design Considerations

To guarantee the integrity — or correctness — 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 auto-crossing 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:

```
A: [k1] {n++}
B: [k2]
```

In this example, the action `{n++}` executes even when conditions `c1` and `c2` 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 Simulink functions during major 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 the model.

For more information, see Chapter 26, "Using Simulink Functions in Stateflow Charts."

Compute derivatives only 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 the during action 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 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

Summary of Rules for Continuous-Time Modeling

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

Consider the following chart.

![Chart Diagram]

In this example, the action \( n++ \) executes even when conditions \( c1 \) and \( c2 \) are false. In this case, \( n \) is updated in a minor time step because there is no state transition.

Do not write to local continuous data in minor actions because these actions execute during 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 Simulink during minor time steps. You can call Simulink in state entry or exit actions, and transition actions. However, if you try to call Simulink 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.”

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.

Do not use continuous-time charts.

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

Use discrete variables to govern conditions in discrete actions.

This restriction prevents changes from occurring between major time steps. A chart with a continuous update method generates errors if it changes states in the minor time steps and, therefore, update state only during major time steps when mode changes occur. Using that heuristic, a Stateflow chart can always compute outputs based on a consistent state for continuous-time.

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 that heuristic, a Stateflow chart can always compute outputs based on a consistent state for continuous-time.

‘Update local data only in transition, entry, and exit actions’

Rationale for Design Considerations

To guarantee the integrity — or correctness — 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
- Re-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 that heuristic, a Stateflow chart can always compute outputs based on a consistent state for continuous-time.

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

‘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

‘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.

Design Considerations for Continuous-Time Modeling in Stateflow Charts

'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]


Modeling Continuous-Time Systems in Stateflow Charts

‘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).
  
  \[
  \text{rec } x = x + 1.0
  \]

- OK as an algebraic constraint (e.g., Simulink and Modelica)
  
  \[
  \text{rec } x = 1.0 - x
  \]

- OK in constructive logic (Esterel)
  
  \[
  \text{rec } z1 = \text{if } c \text{ then } z2 \text{ else } y \\
  \text{and } z2 = \text{if } c \text{ then } x \text{ else } z1
  \]

- Modularity:
  
  \[
  \text{let } \text{node } \text{gonthier}(x, y) = (x, y) \\
  \text{let } \text{node } \text{feedback}(x) = y \text{ where} \\
  \text{rec } (z, y) = \text{gonthier}(x, z)
  \]

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.,

\[
\text{let node } \text{plus}(x, y) = x + 0 \to \text{pre } y
\]

We get: \( f : \forall \alpha_1, \alpha_2. \alpha_1 \times \alpha_2 \to \alpha_1 \)

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

The following is rejected; the next is accepted.

\[
\text{rec } \text{der } y' = -. g \text{ init } 0.0 \ \text{reset } \text{up}(\cdot y) \to -0.9 \cdot y'
\]
and \( \text{der } y = y' \text{ init } y0 \)

\[
\text{rec } \text{der } y' = -. g \text{ init } 0.0 \ \text{reset } \text{up}(\cdot y) \to -0.9 \cdot \text{last } y'
\]
and \( \text{der } y = y' \text{ init } y0 \)
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.