Zélus, a Synchronous Language with ODEs

Marc Pouzet

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

Seminar, Berkeley Univ.
Feb. 11, 2014

1 Joint work with Benveniste, Bourke, Caillaud (INRIA) and Pagano (Esterel-Tech.).
Hybrid Systems Modelers

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

Many tools exist

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

Focus on programming language issues to improve safety.

Our proposal

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

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

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

A conservative extension:
Any synchronous program must be compiled, optimized, and executed as per usual
The Simulation Engine of Hybrid Systems

Alternate discrete steps and integration steps

\[ \sigma', y' = d_\sigma(t, y) \quad u \text{p}z = g_\sigma(t, y) \quad \dot{y} = f_\sigma(t, y) \]

Properties of the three functions

- \(d_\sigma\) gathers all discrete changes.
- \(g_\sigma\) defines signals for zero-crossing detection.
- \(f_\sigma\) and \(g_\sigma\) should be \textit{free of side effects} and, better, \textit{continuous}. 
Causality Issues

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

The classical solution for difference equations (Lustre)

\[
x = 0 \rightarrow \text{pre } y \quad \text{and} \quad y = \text{if } c \text{ then } x + 1 \text{ else } x
\]
defines the two sequences \((x_n)_{n \in \mathbb{N}}\) and \((y_n)_{n \in \mathbb{N}}\):

\[
x(0) = 0 \quad y(n) = \text{if } c(n) \text{ then } x(n) + 1 \text{ else } x(n)
\]

\[
x(n) = y(n - 1)
\]

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

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

Can we reproduce the very same argument when mixing ODEs and discrete transitions and generate sequential code?
The Non Standard Interpretation of Hybrid Systems

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

der \ y = z \ init 4.0 \ and \ z = 10.0 - 0.1 \ * \ y \ and \ k = y + 1.0

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

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

Consider the value that y would have if computed by an ideal solver taking an infinitesimal step of duration \ \partial.

\*y(n) stands for the values of y at instant \ n\partial, with \ n \in \mathbb{N}^\ast \ a non-standard integer.

\[
\begin{align*}
\*y(0) &= 4 \\
\*y(n + 1) &= \*y(n) + \*z(n) \cdot \partial \\
\*z(n) &= 10 - 0.1 \cdot \*y(n) \\
\*k(n) &= \*y(n) + 1
\end{align*}
\]
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:

$$\text{der } y = 1.0 \text{ init } 0.0 \text{ reset } \text{up}(y - 1.0) \rightarrow 0.0$$

The ideal non-standard semantics is:

$$\begin{align*}
*y(0) &= 0 \\
*ly(n) &= *y(n - 1) + \partial \\
*z(0) &= \text{false} \\
*c(n) &= (*y(n) - 1) \geq 0 \\
*z(n) &= *c(n) \land \neg *c(n - 1)
\end{align*}$$

$*y(n)$ depends on itself so this set of equations is not causal.
Acessing the “left limit” of a signal (last y)

Two ways to break this cycle

▶ consider that the effect of the zero-crossing is delayed by one cycle, that is, the test is made on $z(n-1)$ instead of on $z(n)$, or,

▶ distinguish the current value of $y(n)$ from the value it would have had were there no reset, namely $y(n)$.

Testing a zero-crossing of $y(n)$ (instead of $y$)

$$c(n) = (y(n) - 1) \geq 0,$$

gives a program that is causal since $y(n)$ no longer depends instantaneously on itself.

Write

$$\text{der } y = 1.0 \ \text{init } 0.0 \ \text{reset up(last } y - 1.0) \rightarrow 0.0$$
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.
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)
Excerpt of C code produced by RTW (release R2009)

```c
// P_0 = -2.0 P_1 = -3.0 P_2 = -4.0 P_3 = 1.0
static void mdlOutputs(SimStruct * S, int_T tid)
{  _rtX = (ssGetContStates(S));
...
    _rtB = (_ssGetBlockIO(S));
    _rtB->B_0_0_0 = _rtX->Integrator1_CSTATE + _rtP->P_0;
    _rtB->B_0_1_0 = _rtP->P_1 * _rtX->Integrator1_CSTATE;
    if (ssIsMajorTimeStep (S))
    { ...
        if (zcEvent || ...)
        { (ssGetContStates (S))->Integrator0_CSTATE =
            _ssGetBlockIO (S))->B_0_1_0;
        }
    }
...
    (_ssGetBlockIO (S))->B_0_2_0 =
    (ssGetContStates (S))->Integrator0_CSTATE;
    _rtB->B_0_3_0 = _rtP->P_2 * _rtX->Integrator0_CSTATE;
    if (ssIsMajorTimeStep (S))
    { ...
        if (zcEvent || ...)
        { (ssGetContStates (S))-> Integrator1_CSTATE =
            (ssGetBlockIO (S))->B_0_3_0;
        }
    }
...
} ...
```
An explanation of the bug

The source program

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

Its non-standard interpretation

\[
\begin{align*}
^*x(n) &= \text{ if } ^*z(n) \text{ then } -3 \cdot ^*y(n - 1) \text{ else } ^*x(n - 1) + \partial \\
^*y(n) &= \text{ if } ^*z(n) \text{ then } -4 \cdot ^*x(n - 1) \text{ else } ^*y(n - 1) + \partial \cdot ^*x(n - 1)
\end{align*}
\]

Explanation

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

zelus.di.ens.fr
Combinatorial and sequential functions

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

```plaintext
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:

```plaintext
val add : int × int → int
val mix_max : α × α → α × α
val after : int × int → bool
```

x, y, etc. are infinite sequences of values.
A signal is a sequence of values or *stream*.

A system is a function from streams to streams.

Operations apply pointwise to their arguments.

All streams progress *synchronously*.
The counter can be instantiated twice in a two state automaton,

\[
\text{let node blink } \ (n, m, t) = x \text{ where}
\]

\[
\text{automaton}
\]

| 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 \text{true}\ for \(n\) occurrences of \(t\) and \text{false}\ for \(m\) occurrences of \(t\).

\[
\text{let node blink\_reset } \ (r, n, m, t) = x \text{ where}
\]

\[
\text{reset}
\]

\[
\text{automaton}
\]

| On → do x = true until (after(n, t)) then Off
| Off → do x = false until (after(m, t)) then On

\text{every } r

The type signatures inferred by the compiler are:

\[
\text{val blink : int } \times \text{ int } \times \text{ int } ^{D} \to \text{ bool}
\]

\[
\text{val blink\_reset : int } \times \text{ int } \times \text{ int } \times \text{ int } ^{D} \to \text{ bool}
\]
Examples

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

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

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

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

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

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

The type signature is:

val bouncing : float × float × float → float × 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)

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

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

Ensure that continuous and discrete time signals interfere correctly

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

Ensure that fix-point exist and code can be scheduled

- Algebraic loops must be statically detected.
- Introduce the operator \( \text{last}(x) \) as the “left limit” of a signal.
- Ensure that signals are left-continuous during integration.
- Rely on a dedicated type system.
Mixing discrete (logical) time and continuous time

Given:

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

Explicitly relate simulation and logical time (using zero-crossings)

Try to minimize the effects of solver parameters and choices
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)
```
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:

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

Interpretation:

▶ Option 1: \( \mathbb{N} \subseteq \mathbb{R} \)
▶ Option 2: depends on solver
▶ Option 3: infinitesimal steps
▶ Option 4: type and reject

Explicitly relate simulation and logical time (using zero-crossings)

Try to minimize the effects of solver parameters and choices
Mixing discrete (logical) time and continuous time

Given:

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

Define:

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

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:

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

Define:

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

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:

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

Define:

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

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:

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

Define:

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

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 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:

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

Define:

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

- 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-and-effect system.

The type language

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

Initial conditions

\[
(+): \text{int} \times \text{int} \xrightarrow{A} \text{int} \\
\text{if}: \forall \beta. \text{bool} \times \beta \times \beta \xrightarrow{A} \beta \\
(=): \forall \beta. \beta \times \beta \xrightarrow{D} \text{bool} \\
\text{pre}(\cdot): \forall \beta. \beta \xrightarrow{D} \beta \\
\cdot \text{fby} \cdot: \forall \beta. \beta \times \beta \xrightarrow{D} \beta \\
\text{up}(\cdot): \text{float} \xrightarrow{C} \text{zero} \\
\cdot \text{on} \cdot: \text{zero} \times \text{bool} \xrightarrow{A} \text{zero}
\]
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 16-18
“Summary of Rules for Continuous-Time Modeling” on page 16-20

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 anti-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 state 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 continuously 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 (++) executes even when conditions c1 and c2 are false. In this case, a state 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 needs 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 correct 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.

Modeling Continuous-Time Systems in Stateflow Charts

Design Considerations for Continuous-Time Modeling in Stateflow Charts

‘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 outputs do not depend on unpredictable factors — or side effects — such as:

- 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 when 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, a port updated in a minor time step because there is no state transition.

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

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

In this section...

- Rationale for Design Considerations
- Summary of Rules for Continuous-Time Modeling
- Design Considerations for Continuous-Time Modeling in Stateflow Charts

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
- Do not write to local continuous data in draining actions
- Do not call Simulink functions in state during actions or transition conditions

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

A Simulink model reads continuous-time derivatives during minor time steps. When placed in a Stateflow chart, Simulink functions cause side effects that may occur in the minor time step, whether or not there is a state transition. Therefore, you should compute derivatives or calling actions to give your Simulink model the most current calculation.

Do not write outputs and derivatives in state 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 based on a constant state for continuous-time.

Use discrete variables to govern conditions in draining actions

This restriction prevents changes from occurring between major time steps. When placed in draining actions, conditions that affect control flow should be governed by discrete variables because they do not change between minor 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 was a continuous-time model.

▶ ‘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

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

‘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

- 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

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

Consider the following chart.

In this example, the action \( n++ \) updates 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 read outputs and derivatives in states or transitions

This restriction ensures correct outputs in a minor time step because it prevents a Stateflow chart from using values that may no longer be valid in that state's major time step.

Do not use input events in continuous-time charts

The presence of input events makes a chart behave like a triggered subsystem, which is not allowed in continuous-time context. The following model generates an error if the chart uses a continuous update method.

Do not write to local continuous data in state entry or exit actions or transition conditions

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

- Update local data only in transition, entry, and exit actions
- Compute derivatives only in during actions
- Do not call Simulink functions in state during actions or transition conditions
- Do not read outputs and derivatives in states or transitions
- Use discrete variables to govern conditions in actions

Rationale for Design Considerations

To ensure 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 means that inputs do not depend on unmeasurable factors — or side-effects — such as:

- Side-effects of Simulink functions
- Number of minor time steps
- Crossings loop
- A Stateflow chart always computes outputs based on a constant state for continuous-time.

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
- Do not call Simulink functions during actions or transition conditions
- Compute derivatives only in during actions
- Do not read outputs and derivatives in states or transitions
- Use discrete variables to govern conditions in during actions

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

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.

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

- But NOK if sequential code generation is targeted (algebraic loop).

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

- But it calls for an expensive boolean analysis.

Can we find a simple and uniform justification for a program mixing continuous-time and discrete-time signals to be causaly correct?
Causality [HSCC’14]

We follow the Lustre/Lucid Synchrone condition for causality: every feedback loop must cross a delay.

Intuition: associate a time stamp to every expression and ensure that the relation between those time stamps is a partial order.

The type language

\[
\begin{align*}
\sigma &::= \forall \alpha_1, \ldots, \alpha_n : C. \; ct \xrightarrow{k} ct \\
ct &::= ct \times ct \mid \alpha \\
k &::= D \mid C \mid A
\end{align*}
\]

Precedence relation:

\[
C ::= \{ \alpha_1 < \alpha'_1, \ldots, \alpha_n < \alpha'_n \}
\]

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

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

\[(\text{TAUT})\]
\[C + \alpha_1 < \alpha_2 \vdash \alpha_1 < \alpha_2\]

\[(\text{TRANS})\]
\[
\frac{C \vdash ct_1 < ct' \quad C \vdash ct' < ct_2}{C \vdash ct_1 < ct_2}
\]

\[(\text{PAIR})\]
\[
\frac{C \vdash ct_1 < ct'_1 \quad C \vdash ct_2 < ct'_2}{C \vdash ct_1 \times ct_2 < ct'_1 \times ct'_2}
\]

\[(\text{ENV})\]
\[
\forall i \in \{1, \ldots, n\}, C \vdash ct_i < ct'_i
\]
\[
C \vdash [x_1 : ct_1; \ldots; x_n : ct_n] < [x_1 : ct'_1; \ldots; x_n : ct'_n]
\]
The Type System

Type Judgments

\[(\text{TYP-EXP})\]
\[C \mid G, H \vdash_k e : ct\]

\[(\text{TYP-ENV})\]
\[C \mid G, H \vdash_k E : H'\]

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

Initial Conditions

\[(+), (-), (*), (/) : \forall \alpha. \alpha \times \alpha \xrightarrow{A} \alpha\]
\[\text{pre}(\cdot) : \forall \alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\}. \alpha_1 \xrightarrow{D} \alpha_2\]
\[\cdot \text{fby} \cdot : \forall \alpha_1, \alpha_2 : \{\alpha_1 < \alpha_2\}. \alpha_1 \times \alpha_2 \xrightarrow{D} \alpha_1\]
\[\text{up}(\cdot) : \forall \alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\}. \alpha_1 \xrightarrow{C} \alpha_2\]
The Typing Rules

\[(\text{APP})\]
\[
C, ct_1 \xrightarrow{k} ct_2 \in \text{Inst}(G(f)) \quad C \mid G, H \vdash_k e : ct_1
\]
\[
C \mid G, H \vdash_k f(e) : ct_2
\]

\[(\text{VAR})\]
\[
C \mid G, H + x : ct \vdash_k x : ct
\]

\[(\text{LAST})\]
\[
C \vdash ct_2 < ct_1
\]
\[
C \mid G, H + x : ct_1 \vdash_D \text{last}(x) : ct_2
\]

\[(\text{EQ})\]
\[
C \mid G, H \vdash_k p : ct \quad C \mid G, H \vdash_k e : ct
\]
\[
C \mid G, H \vdash_k p = e : [ct/p]
\]

\[(\text{DER})\]
\[
C \mid G, H \vdash_c e : ct_1 \quad C \vdash ct_2 < ct_1
\]
\[
C \mid G, H \vdash_c \text{der} \times = e : [ct_2/x]
\]

\[(\text{SUB})\]
\[
C \mid G, H \vdash_k e : ct \quad C \vdash ct < ct'
\]
\[
C \mid G, H \vdash_k e : ct'
\]
Examples

Discrete case

let node integr(xi, x’) = x where
rec x = xi \rightarrow pre x + (pre x’ \ast step)

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

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

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

Continuous case

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

let hybrid loop(x) = y where rec y = f(y) + x
The causality of last

Yet, \( \text{last} \times \) must appear in a discrete context only. In NS semantics:

\[
\text{last}(x)(t) = x(\cdot t)
\]

\( \text{last}(x) \) does not necessarily break causality cycles. E.g.:

\[
\text{rec } x = \text{last } x + 1.0
\]

A more expressive analysis

\[
\text{last}(x) = \begin{cases} 
\text{if } d \text{ then } \text{pre}(x) \text{ else } x 
\end{cases}
\]

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

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

\[
\text{(LAST)}
\]

\[
C \vdash ct'_1 < ct_1
\]

\[
C, H + [x : ct_1 + ct_2] \vdash \text{last}(x) : ct'_1 + ct_2
\]
Compiler architecture

Built on an existing synchronous compiler

- Source-to-source and traceable transformations
- Resulting program is synchronous and translated to sequential code
Comparison with existing tools

Simulink/Stateflow (Mathworks)

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

Modelica

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

Ptolemy

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

Semantics

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

DAEs?

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