Languages for Programming Hybrid Discrete/Continuous-Time Systems

Marc Pouzet

ENS Paris/UPMC/INRIA

Collège de France, March 26, 2014

In collaboration with Albert Benveniste, Timothy Bourke, Benoit Caillaud and Bruno Pagano.
Current Practice
Hybrid Systems Modelers
Some issues

Interlude: interacting with a numerical solver

Key elements of our approach
Non Standard Synchronous Semantics
Typing
Causality
Compilation
Trends for building **safe and complex** embedded systems

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

A **reference semantics** independent of any implementation.

A base for **simulation, testing, formal verification**.

Then **compiled** into executable code, sequential or parallel.

A way to achieve **correct-by-construction** software.
Synchronous Block Diagram Languages\(^1\)

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

\(^1\)Cf. previous courses by Gérard Berry.
A good match for programming discrete-time controllers
Their semantics is simple and mathematically precise:

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

Then compiler ensures strong safety properties.
The program is deterministic.
The generated code runs in bounded time and memory.

Efficient and fully traceable code generation.
The code is correct w.r.t the source model.
Meets the highest quality level of civil avionics (DO178C, level A).²

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

But modern systems need more...
The Current Practice of Hybrid Systems Modeling

Embedded software interacts with physical devices.

The whole system has to be modeled: the controller and the plant.\textsuperscript{4}

\textsuperscript{4}Image by Esterel-Technologies/ANSYS.
Example: a Bang-bang controller [demo].
A Wide Range of Hybrid Systems Modelers Exist

Ordinary Differential Equations + discrete time:
  Simulink/Stateflow ($\geq 10^6$ licences), LabView, Ptolemy II, etc.

Differential Algebraic Equations + discrete time:
  Modelica, VHDL-AMS, VERILOG-AMS, etc.

Dedicated tools for multi-physics:
  Mechanics, electro-magnetics, fluid, etc.

Co-simulation/combination of tools:
  Agree on a common format/protocol: FMI/FMU, S-functions, etc.
  Convert the model of one tool into another.
Underlying Mathematical Models

Synchronous parallelism, sequence equations:\(^5\)

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

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

Ordinary Differential Equations (ODEs):\(^6\)

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

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

---

\(^5\) Cf. Course by Gerard Berry, Spring 2013.

Is there anything left to do?

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

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

But what if the two are mixed together?

Is it enough to connect one to the other?

Can you trust code automatically generated from such tools?
Strange beasts...
Typing issue 1: Mixing continuous & discrete components

![Diagram](image)

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

Basic model

```
0 0.5 1 1.5 2 2.5 3
0 10 20 30 40 50 60 70 80 90 100
Time
```
Typing issue 1: Mixing continuous & discrete components

- The shape of \( \text{cpt} \) depends on the steps chosen by the solver.
- Putting another component in parallel can change the result.
Typing issue 2: Boolean guards in continuous automata

How long is a discrete step?

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

Discrete time is not logical: it is that of the simulation engine.
Causality issue: the Simulink state port

The output of the state port is the same as the output of the block’s standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block’s standard output if the block had not been reset.

–Simulink Reference (2-685)
Causality issue: the Simulink state port

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)

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

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

\[ y = -4 \cdot \text{last} \quad x = -8 \]
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

The output of the state port is the same as the output of the block's standard output 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)

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

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

\[ y = (-4 \cdot x = 24) ! \]
Excerpt of C code produced by RTW (release R2009)

```c
static void mdlOutputs(SimStruct * S, int_T tid)
{
  _rtX = (ssGetContStates(S));
  ...
  _rtB = (_ssGetBlockIO(S));
  _rtB->B_0_0_0 = _rtX->Integrator1_CSTATE + _rtP->P_0;
  _rtB->B_0_1_0 = _rtP->P_1 * _rtX->Integrator1_CSTATE;
  if (ssIsMajorTimeStep (S))
  {
    if (zcEvent || ...)
    {
      (ssGetContStates (S))->Integrator0_CSTATE =
      _ssGetBlockIO (S))->B_0_1_0;
    }
  }
  ...
  (_ssGetBlockIO (S))->B_0_2_0 =
  (ssGetContStates (S))->Integrator0_CSTATE;
  _rtB->B_0_3_0 = _rtP->P_2 * _rtX->Integrator0_CSTATE;
  if (ssIsMajorTimeStep (S))
  {
    if (zcEvent || ...)
    {
      (ssGetContStates (S))-> Integrator1_CSTATE =
      (ssGetBlockIO (S))->B_0_3_0;
    }
  }
  ...
}
```

Before assignment: integrator state contains ‘last’ value

\[ x = -3 \cdot \text{last } y \]

After assignment: integrator state contains the new value

\[ y = -4 \cdot x \]

So, \( y \) is updated with the new value of \( x \)

There is a problem in the treatment of causality.
Current Practice: conclusion

What is the semantics of these tools?
When the manual and implementations diverge, which is right?
There are side effects, global variables, backtracking.
Hard to judge whether the generated code is correct.

What more could we want?
An cleaner integration of discrete and continous time.
Static rejection of bizarre programs.
Current Practice
  Hybrid Systems Modelers
  Some issues

Interlude: interacting with a numerical solver

Key elements of our approach
  Non Standard Synchronous Semantics
  Typing
  Causality
  Compilation
Interacting with a numerical solver

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

All major tools thus calculate approximate solutions numerically.

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

Designed by experts!

Compute a discrete-time approximations of continuous-time signals. **Subtle:** variable step, change order dynamically, explicit/implicit.

Define compilation schemes with solver’s internals kept abstract.
Bouncing ball model

\[ F = m \cdot a \]

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

\[ \frac{d^2 h(t)}{dt^2} = -g \]
Bouncing ball model

\[ F = m \cdot a \]

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

\[ \frac{d^2 h(t)}{d t^2} = -g \]

\[ \dot{v} = -g \quad v(0) = v_0 \]

\[ \dot{h} = v \quad h(0) = h_0 \]

First-order ODE
Bouncing ball model

\[ F = m \cdot a \]
\[ m \cdot -g = m \cdot \frac{d^2 h(t)}{dt^2} \]
\[ \frac{d^2 h(t)}{dt^2} = -g \]

\[ \dot{v} = -g \quad v(0) = v_0 \]
\[ \dot{h} = v \quad h(0) = h_0 \]

First-order ODE

\[ v(t) = v_0 + \int_0^t -g \cdot d\tau \]
\[ h(t) = h_0 + \int_0^t v(\tau) \cdot d\tau \]
Bouncing ball model

\[ F = m \cdot a \]
\[ m \cdot -g = m \cdot \frac{d^2 h(t)}{dt^2} \]
\[ \frac{d^2 h(t)}{dt^2} = -g \]

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

Solver

First-order ODE

\[ \dot{v} = -g \]
\[ \dot{h} = v \]
\[ v(0) = v_0 \]
\[ h(0) = h_0 \]

\[ y_i = [v_0; h_0] \]

approximation

\[ v(t) = v_0 + \int_0^t -g \cdot d\tau \]
\[ h(t) = h_0 + \int_0^t v(\tau) \cdot d\tau \]
Bouncing ball model

\[ F = m \cdot a \]
\[ m \cdot -g = m \cdot \frac{d^2 h(t)}{dt^2} \]
\[ \frac{d^2 h(t)}{dt^2} = -g \]

First-order ODE

\[ \dot{v} = -g \]
\[ \dot{h} = v \]
\[ v(0) = v_0 \]
\[ h(0) = h_0 \]

Solver

\[ y_i = [v_0; h_0] \]

Approximation

\[ v(t) = v_0 + \int_0^t -g \cdot d\tau \]
\[ h(t) = h_0 + \int_0^t v(\tau) \cdot d\tau \]

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

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

- Bigger and bigger steps (bound by \( h_{\text{min}} \) and \( h_{\text{max}} \))
- \( t \) does not necessarily advance monotonically
  - No side-effects within \( f \) or \( g \)
Solver execution (e.g., LLNL Sundials CVODE)

Give solver two functions: 
\[ \dot{y} = f_\sigma(t, y), \quad \text{upz} = g_\sigma(t, y) \]

- Bigger and bigger steps (bound by \( h_{\text{min}} \) and \( h_{\text{max}} \))
- \( t \) does not necessarily advance monotonically
  - No side-effects within \( f \) or \( g \)
Solver execution (e.g., LLNL Sundials CVODE)

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

- Bigger and bigger steps (bound by \( h_{\text{min}} \) and \( h_{\text{max}} \))
- \( t \) does not necessarily advance monotonically
  - No side-effects within \( f \) or \( g \)
Solver execution (e.g., LLNL Sundials CVODE)

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

- Bigger and bigger steps (bound by \( h_{\text{min}} \) and \( h_{\text{max}} \))
- \( t \) does not necessarily advance monotonically
  - No side-effects within \( f \) or \( g \)
Solver execution (e.g., LLNL Sundials CVODE)

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

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - No side-effects within \( f \) or \( g \)
Solver execution (e.g., LLNL Sundials CVODE)

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

- Bigger and bigger steps (bound by \( h_{\text{min}} \) and \( h_{\text{max}} \))
- \( t \) does not necessarily advance monotonically
  - No side-effects within \( f \) or \( g \)
Solver execution (e.g., LLNL Sundials CVODE)

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

- Bigger and bigger steps (bound by \(h_{\text{min}}\) and \(h_{\text{max}}\))
- \(t\) does not necessarily advance monotonically
  - No side-effects within \(f\) or \(g\)
**Solver execution (e.g., LLNL Sundials CVODE)**

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

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - No side-effects within \( f \) or \( g \)
 Solver execution (e.g., LLNL Sundials CVODE)

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

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - No side-effects within \( f \) or \( g \)
Solver execution (e.g., LLNL Sundials CVODE)

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

- Bigger and bigger steps (bound by \( h_{\text{min}} \) and \( h_{\text{max}} \))
- \( t \) does not necessarily advance monotonically
  - No side-effects within \( f \) or \( g \)
Solver execution (e.g., LLNL Sundials CVODE)

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

- Bigger and bigger steps (bound by \( h_{\text{min}} \) and \( h_{\text{max}} \))
- \( t \) does not necessarily advance monotonically
  - No side-effects within \( f \) or \( g \)
Solver execution (e.g., LLNL Sundials CVODE)
Give solver two functions: \( \dot{y} = f_\sigma(t, y) \), \( upz = g_\sigma(t, y) \)

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - No side-effects within \( f \) or \( g \)
The Simulation Engine of Hybrid Systems

Alternate discrete steps and integration steps

\[

d\sigma(t, y), \quad y' = d_{\sigma}(t, y) \quad \text{upz} = 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 free of side effects and, better, continuous.
Numerical Integration (Sundials CVODE)

\[
\begin{align*}
\dot{y}(t) &= \begin{cases} 
1 & \text{if } t < 3 \\
0 & \text{if } 3 \leq t \leq 7 \\
-1 & \text{if } 7 < t
\end{cases} \\
y(0) &= 0
\end{align*}
\]
Numerical Integration: a derivative with a discontinuity

let \( f(t, y) = \)

\[
\begin{align*}
\text{if } t < 3.0 & \text{ then } 1.0 \\
\text{else if } t \leq 7.0 & \text{ then } 0.0 \\
\text{else} & \text{ else } -1.0
\end{align*}
\]
Numerical Integration: discrete state with three modes

\[ f(\text{discrete\_state})(t, y) = \begin{cases} 1.0 & \text{RampingUp} \\ 0.0 & \text{Flat} \\ -1.0 & \text{RampingDown} \end{cases} \]

\[ g(\text{discrete\_state})(t, y) = \begin{cases} t - 0.3 & \text{RampingUp} \\ t - 0.7 & \text{Flat} \\ t & \text{else} \end{cases} \]

\[ d(\text{discrete\_state}) z(t, y) = \begin{cases} \text{RampingUp when } z \rightarrow \text{Flat} \\ \text{Flat when } z \rightarrow \text{RampingDown} \\ s \rightarrow s \end{cases} \]
Numerical Integration: with no reinit of the solver

ramp (with zero-crossings but no reinit)
Current Practice

Hybrid Systems Modelers

Some issues

Interlude: interacting with a numerical solver

Key elements of our approach

Non Standard Synchronous Semantics
Typing
Causality
Compilation
Key elements of our approach

Build a hybrid modeler on top of a synchronous language.

Use synchronous constructs for arbitrary mix of discrete and continuous.

Divide and Recycle

Recycle existing synchronous languages techniques.

Semantics, static checking, code-generation.

Divide from the code what is for the solver.

Simulate with off-the-shelf numerical solvers.

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

These elements are experimented within the language Zélus.
Zélus
zelus.di.ens.fr
A signal is a sequence of values. Nothing is said about the actual time to go from one instant to another.

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

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

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

When fed into the compiler, we get:

```ml
val add : int × int → int
val min_max: α × α → α × α
val after : int × bool → bool
```

\( x, y, \) etc. are infinite sequences of values.
The counter can be instantiated twice in a two state automaton,

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

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

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

The type signatures inferred by the compiler are:

val blink : int × int × bool → bool
val blink_reset : bool × int × int × bool → bool
Examples

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

(* an hysteretic 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)

³This is the hybrid version of one of Nicolas Halbwachs’ examples with which he presented Lustre at the Collège de France, in January 2010.
The Bouncing ball [demo]

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

Its type signature is:

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

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

- Synchronous constructs: data-flow equations/automata.
- An ODE with initial condition: \( \text{der } x = e \text{ init } e_0 \)
- \( \text{last } x \) is the left limit of \( x \).
- Detect a zero-crossing (from negative to positive): \( \text{up}(x) \).
- This defines a discrete instant, that is, an event.
- All discrete changes must occur on an event. E.g.:

  \[
  \text{let hybrid } f(x, y) = (v, z1, z2) \text{ where } \\
  \quad \text{rec } v = \text{present } z1 \rightarrow 1 \mid z2 \rightarrow 2 \text{ init } 0 \\
  \quad \text{and } z1 = \text{up}(x) \\
  \quad \text{and } z2 = \text{up}(y) \\
  \]

  \[
  \text{val } f : \text{float } \times \text{float} \rightarrow \text{int } \times \text{zero } \times \text{zero} \\
  \]

  - If \( x = \text{up}(e) \), all handlers using \( x \) are governed by the same event.
Three difficulties

Semantics

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

Ensure that continuous and discrete time signals interfere correctly.

- Discrete time should stay logical and independent on when the solver decides to stop.
- Otherwise, we get the bizarre behaviors seen previously.

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

- Algebraic loops must be statically detected.
- Restrict the use of \texttt{last x} so that signals are proved to be continuous during integration.
A Non-standard Semantics for Hybrid Modelers [JCSS’12]

We proposed to build the semantics on non-standard analysis.

\[ \text{der } y = z \text{ init } 4.0 \text{ and } z = 10.0 - 0.1 \cdot y \text{ and } k = y + 1.0 \]
defines signals \( y \), \( z \) and \( k \), where for all \( t \in \mathbb{R}^+ \):

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

What would be the value of \( y \) if it were computed by an ideal solver taking an infinitesimal step of duration \( \partial \)?

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

\[
*y(0) = 4 \\
*y(n + 1) = *y(n) + *z(n) \cdot \partial \\
*z(n) = 10 - 0.1 \cdot *y(n) \\
*k(n) = *y(n) + 1
\]
Non standard semantics [JCSS’12]

Let $\star \mathbb{R}$ and $\star \mathbb{N}$ be the non-standard extensions of $\mathbb{R}$ and $\mathbb{N}$.

Let $\partial \in \star \mathbb{R}$ be an infinitesimal, i.e., $\partial > 0, \partial \approx 0$.

Let the global time base or base clock be the infinite set of instants:

$$T_\partial = \{ t_n = n\partial \mid n \in \star \mathbb{N} \}$$

$T_\partial$ inherits its total order from $\star \mathbb{N}$. A sub-clock $T \subset T_\partial$.

What is a discrete clock?

A clock $T$ is termed discrete if it is the result of a zero-crossing or a sub-sampling of a discrete clock. Otherwise, it is termed continuous.

If $T \subseteq \mathbb{T}$, we write $\bullet T(t)$ for the immediate predecessor of $t$ in $T$ and $T^\bullet(t)$ for the immediate successor of $t$ in $T$.

A signal is a partial function from $\mathbb{T}$ to a set of values.
Semantics of basic operations

Replay the classical semantics of a synchronous language.

An ODE with reset on clock $T$: $\text{der } x = e \text{ init } e_0 \text{ reset } z \rightarrow e_1$

$\star x(t_0) = \star e_0(0)$ if $t_0 = \min T$

$\star x(t) = \text{if } \star z(t) \text{ then } \star e_1(t) \text{ else } \star x(\bullet T(t)) + \partial \cdot \star e(\bullet T(t))$ if $t \in T$

last $x$ if $x$ is defined on clock $T$

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

Zero-crossing up($x$) on clock $T$

$\star \text{up}(x)(t_0) = \text{false}$ if $t_0 = \min T$

$\star \text{up}(x)(t) = (\star x(\bullet T(t) \leq 0) \land (\star x(t) > 0))$ if $t \in T$
Non-standard time vs. Super-dense time

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

\[
\begin{align*}
(t, 0) &\quad (t, 1) &\quad (t, 2) \\
(u, 0) &\quad (v, 0) &\quad (v, 1) &\quad (v, 2) &\quad (v, 3)
\end{align*}
\]
Non-standard time vs. Super-dense time

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

- non-standard time modeling $\mathbb{T}_\partial = \{n\partial \mid n \in \ast \mathbb{N}\}$
Typing: mixing discrete (logical) time and continuous time

The following two parallel composition make sense.

Discrete time: the clock should be discrete

\[ \text{let node } \text{sum}(x) = \text{cpt} \text{ where} \]
\[ \text{rec } \text{cpt} = 0 \to \text{pcpt} \]
\[ \text{and } \text{pcpt} = \text{pre}(\text{cpt}) + x \]

Continuous time: the clock should be continuous

\[ \text{let hybrid } \text{bouncing}(y0, y'0) = o \text{ where} \]
\[ \text{rec } \text{der } y = y' \text{ init } y0 \]
\[ \text{and } \text{der } y' = -g \text{ init } y'0 \]
\[ \text{and } o = y + 10.0 \]

The following do not make sense

At what clock should we compute \( \text{cpt} \)?

\[ \text{rec } \text{der } t = 1.0 \text{ init } 0.0 \]
\[ \text{and } \text{cpt} = 0.0 \to \text{pre}(\text{cpt}) + t \]
Intuition

Distinguish functions with three kinds $A/D/C$.

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

Explicitly relate simulation and logical time

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

```plaintext
let hybrid correct (z) = (time, y) where
  rec der time = 1.0 init 0.0
  and y = present up(z) \rightarrow sum(time) init 0.0
```
Basic typing [LCTES’11]

A simple ML type system with effects.

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

\[
(+) \ : \ \text{int} \times \text{int} \xrightarrow{A} \text{int} \\
\text{if} \ : \ \forall \beta. \text{bool} \times \beta \times \beta \xrightarrow{A} \beta \\
(=) \ : \ \forall \beta. \beta \times \beta \xrightarrow{D} \text{bool} \\
\text{pre}(\cdot) \ : \ \forall \beta. \beta \xrightarrow{D} \beta \\
\cdot \text{fby} \cdot \ : \ \forall \beta. \beta \times \beta \xrightarrow{D} \beta \\
\text{up}(\cdot) \ : \ \text{float} \xrightarrow{C} \text{zero}
\]
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...
* “Summary of Rules for Continuous-Time Modeling” on page 16-26
* “Rationale for Design Considerations” on page 16-26

Rationale for Design Considerations

To generate 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 model’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 (variables 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

<table>
<thead>
<tr>
<th>[ n ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P ]</td>
</tr>
<tr>
<td>[ Q ]</td>
</tr>
<tr>
<td>[ 3 ]</td>
</tr>
</tbody>
</table>

Compute derivatives only in discrete actions

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

Do not call Simulink functions in stateentry or exit actions

This restriction prevents mode changes from occurring between major time steps. When placed in discrete 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

‘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)

Stateflow User’s Guide

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

- 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 before entering the new state at the end of the transition.
- Condition actions on a transition, but only if the transition directly reaches another state.

This chart of Stateflow charts.

**Rationale for Design Considerations**

To guarantee the integrity — or correctness — of the results in continuous-time modeling, you must maintain your charts to a restricted subset of Stateflow chart semantics. The restricted semantics mean 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:

  - Do not write to local continuous data in states or transitions because these actions execute during minor time steps. However, if you try to call Simulink functions in state actions or transition conditions, an error message appears when you simulate your model.
  - Do not call Simulink functions in state during actions or transition conditions, an error message appears when you simulate your model.

- 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.
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

Rationale for Design Considerations
To guarantee the integrity — or soundness — of the model in continuous-time modeling, you must constrain your charts to a restricted subset of Stateflow chart semantics. The restricted semantics ensures that no output variable is updated during a state transition. The following list summarizes the design considerations:

- Simulink engine's general restrictions to Stateflow charts
- Number of iterations required to stabilize the integration loop
- By minimizing side effects, a Stateflow chart can maintain its state at minor time steps and, therefore, update outputs during major time steps.
- 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.
- Simulation continues without errors even if there is no state transition.
- Do not call Simulink functions in state during actions or transition conditions.
- Do not write to local continuous data in actions because these actions execute during minor time steps.
- Do not read outputs and derivatives in states or transitions.
- Do not use input events in continuous-time charts.

- 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

Rationale for Design Considerations

To guarantee the integrity — or semantics — of the model in continuous-time modeling, you must constrain your charts to a restricted subset of Stateflow chart semantics. The restricted semantics ensures that:

- Simulink's only computes outputs at major time steps.
- Number of iterations required to stabilize the integration loop or zero.

By minimizing side effects, a Stateflow chart can maintain its state at major time steps and, therefore, execute state transitions without requiring major time steps to make changes occur. Use discrete variables and local state to compute outputs based on major time steps.

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.
- Do not call Simulink functions in state during actions or transition conditions.
- Compute derivatives only in during actions.

- Restrict local data accesses to major time steps.
- Supported by warnings and errors in tool (mostly)

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

To maintain integrity 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.

A Simulink model reads continuous-time derivatives during minor time steps. When placed in state during actions or transition conditions, an error message appears when you simulate your model.

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

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

The Mathworks recommends that you restrict the use of Simulink functions to 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.

Compute derivatives only in during actions.

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.

Compute derivatives only in during actions because these actions give your Simulink model the most current calculation.

Do not read outputs and derivatives in state during actions.

The Mathworks recommends that you restrict the use of Simulink functions to 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.

Use discrete variables to govern conditions 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 forces updates to take place at each transition. For example, the following chart generates errors if the chart is a continuous update chart.

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.

An error message appears when you simulate this model because it violates this rule.

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

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

The Mathworks recommends that you restrict the use of Simulink functions to 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.

Compute derivatives only in during actions because these actions give your Simulink model the most current calculation.

Do not read outputs and derivatives in state during actions.

The Mathworks recommends that you restrict the use of Simulink functions to 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.

Use discrete variables to govern conditions 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 forces updates to take place at each transition. For example, the following chart generates errors if the chart is a continuous update chart.

Consider the following chart.

An error message appears when you simulate this model because it violates this rule.

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

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

The Mathworks recommends that you restrict the use of Simulink functions to 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.

Compute derivatives only in during actions because these actions give your Simulink model the most current calculation.

Do not read outputs and derivatives in state during actions.

The Mathworks recommends that you restrict the use of Simulink functions to 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.

Use discrete variables to govern conditions 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 forces updates to take place at each transition. For example, the following chart generates errors if the chart is a continuous update chart.

Consider the following chart.

An error message appears when you simulate this model because it violates this rule.

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

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

The Mathworks recommends that you restrict the use of Simulink functions to 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.

Compute derivatives only in during actions because these actions give your Simulink model the most current calculation.

Do not read outputs and derivatives in state during actions.

The Mathworks recommends that you restrict the use of Simulink functions to 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.

Use discrete variables to govern conditions 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 forces updates to take place at each transition. For example, the following chart generates errors if the chart is a continuous update chart.

Consider the following chart.

An error message appears when you simulate this model because it violates this rule.

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

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

The Mathworks recommends that you restrict the use of Simulink functions to 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.

Compute derivatives only in during actions because these actions give your Simulink model the most current calculation.

Do not read outputs and derivatives in state during actions.

The Mathworks recommends that you restrict the use of Simulink functions to 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.

Use discrete variables to govern conditions 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 forces updates to take place at each transition. For example, the following chart generates errors if the chart is a continuous update chart.
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)

Our $D/C/A$/zero system extends naturally for the same effect.

For both discrete (synchronous) and continuous (hybrid) contexts.
Causality loops

Yet, some programs are well typed but have algebraic loops. Which programs should we accept?

- OK to reject (no solution).
  \[ \text{rec } x = x + 1 \]

- OK as an algebraic constraint (e.g., Simulink and Modelica).
  \[ \text{rec } x = 1 - x \]

- But NOK if sequential code generation is targeted.

- \text{last } x \text{ does not necessarily break causality loops!}
  \[ \text{rec } x = \text{last } x + 1 \]

- OK
  \[
  \begin{align*}
  \text{rec } \text{der } x &= 1.0 \ \text{init } 0.0 \ \text{reset } z \rightarrow t \\
  \text{and } y &= x +. 1.0 \\
  \text{and } t &= \text{last } y
  \end{align*}
  \]

Can we find a simple and uniform justification?
ODEs with reset

Consider the sawtooth signal \( y : \mathbb{R}^+ \to \mathbb{R}^+ \) such that:

\[
\frac{dy}{dt}(t) = 1 \quad y(t) = 0 \text{ if } t \in \mathbb{N}
\]

written:

\[
\text{der } y = 1.0 \ \text{init} \ 0.0 \ \text{reset} \ \text{up}(y - 1.0) \to 0.0
\]

The ideal non-standard semantics is:

\[
\begin{align*}
^*y(0) &= 0 \\
^*y(n) &= \text{if } ^*z(n) \text{ then } 0.0 \text{ else } ^*y(n) \\
^*ly(n) &= ^*y(n - 1) + \partial \\
^*z(0) &= \text{false} \\
^*z(n) &= ^*c(n) \land \neg ^*c(n - 1)
\end{align*}
\]

\[
^*c(n) = (^*y(n) - 1) \geq 0
\]

This set of equation is not causal: \(^*y(n)\) depends on itself.
Accessing the “left limit” of a signal

There are two ways to break this cycle:

- consider that the effect of the zero-crossing is delayed by one cycle, that is, the test is made on \( *z(n-1) \) 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 \( *ly(n) \).

Testing a zero-crossing of \( ly \) (instead of \( y \)),

\[
*c(n) = (*ly(n) - 1) \geq 0,
\]

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

\[
der y = 1.0 \text{ init } 0.0 \text{ reset up(last y } -. 1.0) \rightarrow 0.0
\]
An explanation of the bug

The source program

\[ \text{rec der } x = 1.0 \text{ init } 0.0 \text{ reset } z \rightarrow -3.0 \ast. \text{ last } y \]
and \[ \text{der } y = x \text{ init } 0.0 \text{ reset } z \rightarrow -4.0 \ast. \text{ last } x \]
and \[ z = \text{up(last } x - . \text{ 2.0)} \]

Its non-standard interpretation

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

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.
Causality Analysis [HSCC’14]

Every feedback loop must cross a delay.

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

The type language

\[ \sigma ::= \forall \alpha_1, \ldots, \alpha_n : C . \ ct \xrightarrow{k} ct \]
\[ ct ::= ct \times ct \ | \ \alpha \]
\[ k ::= D \ | \ C \ | \ A \]

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 \( C \).
Associate a type that express input/output dependences. E.g.,

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

We get: \( f : \forall \alpha_1, \alpha_2. \alpha_1 \times \alpha_2 \xrightarrow{D} \alpha_1 \)

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

The following is rejected; the next is accepted.

\[
\text{rec der } y' = -g \text{ init 0.0 reset up}(-.y) \to -0.9 \ast y'
\]
and \( \text{der } y = y' \text{ init } y_0 \)

\[
\text{rec der } y' = -g \text{ init 0.0 reset up}(-.y) \to -0.9 \ast \text{last } y'
\]
and \( \text{der } y = y' \text{ init } y_0 \)

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

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

Built on an existing synchronous compiler

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

Simulink/Stateflow (Mathworks)
- Integrated treatment of automata vs two distinct languages
- More rigid separation of discrete and continuous behaviors

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

Ptolemy (E.A. Lee et al., Berkeley)
- A unique computational model: synchronous
- Everything is compiled to sequential code (not interpreted)