A Hybrid Synchronous Language with Hierarchical Automata

Static Typing and Translation to Synchronous Code

Albert Benveniste\textsuperscript{1}  Benoît Caillaud\textsuperscript{1}
Timothy Bourke\textsuperscript{1,2}  Marc Pouzet\textsuperscript{2,1}

1. INRIA

2. ENS, Paris

18th Workshop SYNCHRON, Nov. 28 – Dec. 2, 2011, Damarie-les-Lys
Aim

Programming languages perspective:

purely discrete data-flow  well understood  (Lustre, SCADE 6)
purely continuous  well understood  (Numerical solvers, Simulink)
hier. automata (disc.)  well understood  (Statecharts, Esterel)
data-flow + hier. auto.  well understood  (SCADE 6, Esterel v7)

Better understand the combination of discrete and continuous components

The usual questions (and techniques):

- Which programs make sense? (typing)
- How to reason about programs? (semantics, Benveniste et al. The Fundamentals of Hybrid Modelers. JCSS 2011.)
- Efficient and faithful execution? (compilation)

Our interest: a language for programming discrete systems and their physical environments
Aim

Programming languages perspective:

- purely discrete data-flow well understood (Lustre, SCADE 6)
- purely continuous well understood (Numerical solvers, Simulink)
- hier. automata (disc.) well understood (Statecharts, Esterel)
- data-flow + hier. auto. well understood (SCADE 6, Esterel v7)

Better understand the combination of discrete and continuous components

The usual questions (and techniques):

- Which programs make sense? (typing)
- How to reason about programs? (semantics, Benveniste et al. The Fundamentals of Hybrid Modelers. JCSS 2011.)
- Efficient and faithful execution? (compilation)

Our interest: a language for programming discrete systems and their physical environments
Aim

Programming languages perspective:

purely discrete data-flow    well understood    (Lustre, SCADE 6)
purely continuous           well understood    (Numerical solvers, Simulink)
hier. automata (disc.)      well understood    (Statecharts, Esterel)
data-flow + hier. auto.     well understood    (SCADE 6, Esterel v7)

Better understand the combination of discrete and continuous components

The usual questions (and techniques):

- Which programs make sense?  (typing)
- How to reason about programs?  (semantics, Benveniste et al. The Fundamentals of Hybrid Modelers. JCSS 2011.)
- Efficient and faithful execution?  (compilation)

Our interest: a language for programming discrete systems and their physical environments
Aim

Programming languages perspective:

purely discrete data-flow well understood (Lustre, SCADE 6)
purely continuous well understood (Numerical solvers, Simulink)
hier. automata (disc.) well understood (Statecharts, Esterel)
data-flow + hier. auto. well understood (SCADE 6, Esterel v7)

Better understand the combination of discrete and continuous components

The usual questions (and techniques):

- Which programs make sense? (typing)
- How to reason about programs? (semantics, Benveniste et al. The Fundamentals of Hybrid Modelers. JCSS 2011.)
- Efficient and faithful execution? (compilation)

Our interest: a language for programming discrete systems and their physical environments
Approach

► Add Ordinary Differential Equations to an existing synchronous language

► Two concrete reasons:
  ► Increase modeling power (hybrid programming)
  ► Exploit existing compiler (target for code generation)


► Conservative extension: synchronous functions are compiled, optimized, and executed as per usual.

► Extends previous work: add hierarchical automata to LCTES 2011

Understand (continuous) automata and their parallel composition from a synchronous language viewpoint (causality relations, activations (clocks), semantics)
Approach

▶ Add Ordinary Differential Equations to an existing synchronous language

▶ Two concrete reasons:
  ▶ Increase modeling power (hybrid programming)
  ▶ Exploit existing compiler (target for code generation)

▶ Simulate with an external off-the-shelf numerical solver

▶ Conservative extension: synchronous functions are compiled, optimized, and executed as per usual.

▶ Extends previous work: add hierarchical automata to LCTES 2011

Understand (continuous) automata and their parallel composition from a synchronous language viewpoint (causality relations, activations (clocks), semantics)
Approach

- Add Ordinary Differential Equations to an existing synchronous language

- Two concrete reasons:
  - Increase modeling power (hybrid programming)
  - Exploit existing compiler (target for code generation)


- Conservative extension: synchronous functions are compiled, optimized, and executed as per usual.

- Extends previous work: add hierarchical automata to LCTES 2011

Understand (continuous) automata and their parallel composition from a synchronous language viewpoint (causality relations, activations (clocks), semantics)
Approach

- Add Ordinary Differential Equations to an existing synchronous language

- Two concrete reasons:
  - Increase modeling power (hybrid programming)
  - Exploit existing compiler (target for code generation)


- Conservative extension: synchronous functions are compiled, optimized, and executed as per usual.

- Extends previous work: add hierarchical automata to LCTES 2011

Understand (continuous) automata and their parallel composition from a synchronous language viewpoint (causality relations, activations (clocks), semantics)
Lee and Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. EMSOFT’07.
Lee and Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. EMSOFT'07.

Ptolemy and HyVisual

- Programming languages perspective
- Numerical solvers as directors
- Working tool and examples
Lee and Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. EMSOFT’07.

Carloni et al. Languages and tools for hybrid systems design. 2006.

Simulink/Stateflow

- Simulation ⇝ development
- two distinct simulation engines
- semantics & consistency: non-obvious
Our approach

- Source-to-source compilation
- Automata $\leadsto$ data-flow
- Extend other languages (SCADE 6)
Which programs make sense?

Given:

\[
\text{let node } \text{sum}(x) = \text{cpt where } \\
\quad \text{rec } \text{cpt} = (0.0 \ \text{fby} \ \text{cpt}) +. x
\]
Which programs make sense?

Given:

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

Evaluate:

```plaintext
der time = 1.0 init 0.0
and
y = sum(time)
```
Which programs make sense?

Given:

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

Evaluate:

\[
\text{der time} = 1.0 \ \text{init} \ 0.0
\]
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
Which programs make sense?

Given:

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

Evaluate:

```
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
Which programs make sense?

Given:

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

Evaluate:

```
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

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

Try to minimize the effects of solver parameters and choices
Which programs make sense?

Given:

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

Evaluate:

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

Interpretation:

- Option 1: \( \mathbb{N} \subseteq \mathbb{R} \)
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject
Which programs make sense?

Given:

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

Evaluate:

\[
\text{der } \text{time} = 1.0 \ \text{init} \ 0.0
\]

and

\[
\text{y} = \text{sum}(\text{time})
\]

Interpretation:

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

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

Try to minimize the effects of solver parameters and choices
Which programs make sense?

Given:

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

Evaluate:

\[
\text{der time} = 1.0 \ \text{init} \ 0.0
\]
and
\[
y = \text{sum(time)} \ \text{every up}(ez) \ \text{init} \ 0.0
\]

Interpretation:

- Option 1: \( \mathbb{N} \subseteq \mathbb{R} \)
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject
Which programs make sense?

Given:

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

Evaluate:

```
der time = 1.0 init 0.0
and
y = sum(time) every up(ez) init 0.0
```

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
Typing
Motivation

Reject unreasonable programs: behavior depends ‘too much’ on simulation parameters (like the step size, or number of iterations).

Translation to synchronous code: ensure that the translated code has no side effect/state changes during integration.

A signal is discrete if it is activated on a discrete clock. A clock is discrete if it is a zero-crossing event, declared so or a sub-clock of discrete clock.

Type system: reject programs that do not respect the invariant:

- discrete computations in $D$ only
- continuous evolutions in $C$ only
**Typing**

**Motivation**

**Reject unreasonable programs:** behavior depends ‘too much’ on simulation parameters (like the step size, or number of iterations).

**Translation to synchronous code:** ensure that the translated code has no side effect/state changes during integration.

A signal is discrete if it is activated on a discrete clock. A clock is discrete if it is a zero-crossing event, declared so or a sub-clock of discrete clock.

**Type system:** reject programs that do not respect the invariant:

- discrete computations in $D$ only
- continuous evolutions in $C$ only
Reject unreasonable programs: behavior depends ‘too much’ on simulation parameters (like the step size, or number of iterations).

Translation to synchronous code: ensure that the translated code has no side effect/state changes during integration.

A signal is discrete if it is activated on a discrete clock. A clock is discrete if it is a zero-crossing event, declared so or a sub-clock of discrete clock.

Type system: reject programs that do not respect the invariant:

- discrete computations in $D$ only
- continuous evolutions in $C$ only
Reject unreasonable programs: behavior depends ‘too much’ on simulation parameters (like the step size, or number of iterations).

Translation to synchronous code: ensure that the translated code has no side effect/state changes during integration.

A signal is discrete if it is activated on a discrete clock. A clock is discrete if it is a zero-crossing event, declared so or a sub-clock of discrete clock.

Type system: reject programs that do not respect the invariant:

- discrete computations in $D$ only
- continuous evolutions in $C$ only
Typing

Unreasonable programs

\[
der \ y = 1.0 \ \text{init} \ 0.0 \quad \text{and} \quad x = (0.0 \rightarrow \text{pre} \ x) + y
\]

\[
x = 0.0 \rightarrow (\text{pre} \ x +. 1.0) \quad \text{and} \quad \text{der} \ y = x \ \text{init} \ 0.0
\]

- y is a variable that changes \textit{continuously}
- x is \textit{discrete} register
- The relationship between the two is ill-defined
Typing

The type language

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

Initial conditions

\[ (+) : \text{int} \times \text{int} \xrightarrow{A} \text{int} \]
\[ (=) : \forall \beta.\beta \times \beta \xrightarrow{A} \text{bool} \]
\[ \text{if} : \forall \beta.\text{bool} \times \beta \times \beta \xrightarrow{A} \beta \]
\[ \text{pre(.)} : \forall \beta.\beta \xrightarrow{D} \beta \]
\[ \text{.fby.} : \forall \beta.\beta \times \beta \xrightarrow{D} \beta \]
\[ \text{up(.)} : \text{float} \xrightarrow{C} \text{zero} \]
Typing

The type language

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

Initial conditions

\[ (+) : \text{int} \times \text{int} \xrightarrow{A} \text{int} \]
\[ (=) : \forall \beta. \beta \times \beta \xrightarrow{A} \text{bool} \]
\[ \text{if} : \forall \beta. \text{bool} \times \beta \times \beta \xrightarrow{A} \beta \]
\[ \text{pre(.)} : \forall \beta. \beta \xrightarrow{D} \beta \]
\[ \text{fby} . : \forall \beta. \beta \times \beta \xrightarrow{D} \beta \]
\[ \text{up(.)} : \text{float} \xrightarrow{C} \text{zero} \]
Typing
\[ G, H \vdash_\mathbb{C} \text{der } y = 1.0 \ \text{init } 0.0 \quad G, H \vdash_\mathbb{D} x = (0.0 \ \text{fby } (x + 1)) \]

Typing of function body gives its kind \( k \in \{ \mathbb{C}, \mathbb{D}, \mathbb{A} \} \):

\[ h : \text{float } \times \text{float} \xrightarrow{k} \text{float } \times \text{float} \]

Less expressive but simpler than ‘per-wire’ kinds, e.g. Simulink

\[ j : (\text{float}_\mathbb{D}) \times (\text{float}_\mathbb{C}) \rightarrow (\text{float}_\mathbb{D}) \times (\text{float}_\mathbb{C}) \]
Typing of function body gives its kind $k \in \{C, D, A\}$:

$$h : \text{float} \times \text{float} \xrightarrow{k} \text{float} \times \text{float}$$

Less expressive but simpler than ‘per-wire’ kinds, e.g. Simulink

$$j : (\text{float}_D) \times (\text{float}_C) \longrightarrow (\text{float}_D) \times (\text{float}_C)$$
Typing

\[ G, H \vdash_c \text{der } y = 1.0 \text{ init } 0.0 \quad G, H \vdash_D x = (0.0 \text{ fby } (x + 1)) \]

\[ G, H \vdash_? \text{der } y = \cdots \text{ and } x = \cdots \]

Typing of function body gives its kind \( k \in \{C, D, A\} \):

\[ h : \text{float} \times \text{float} \xrightarrow{k} \text{float} \times \text{float} \]

Less expressive but simpler than ‘per-wire’ kinds, e.g. Simulink

\[ j : (\text{float}_D) \times (\text{float}_C) \rightarrow (\text{float}_D) \times (\text{float}_C) \]
Typing

\[ G, H \vdash_{\text{c}} \text{der } y = 1.0 \text{ init } 0.0 \]

\[ G, H \vdash_{\text{d}} x = (0.0 \text{ fby } (x + 1)) \]

\[ G, H \vdash_{\text{c}} x' = (0.0 \text{ fby } (x + 1)) \]

\[ \text{every up(ez) init } 0.0 \]

Typing of function body gives its kind \( k \in \{\text{C, D, A}\} \):

\[ h : \text{float } \times \text{float} \rightarrow^{k} \text{float } \times \text{float} \]

Less expressive but simpler than ‘per-wire’ kinds, e.g. Simulink

\[ j : (\text{float}_D) \times (\text{float}_C) \rightarrow (\text{float}_D) \times (\text{float}_C) \]
Typing

\[ G, H \vdash_c \text{der } y = 1.0 \text{ init } 0.0 \]
\[ G, H \vdash_D x = (0.0 \text{ fby } (x + 1)) \]
\[ G, H \vdash_c x' = (0.0 \text{ fby } (x + 1)) \]
\[ \text{every up(ez) init } 0.0 \]

Typing of function body gives its kind \( k \in \{C, D, A\} \):

\[ h : \text{float} \times \text{float} \xrightarrow{k} \text{float} \times \text{float} \]

Less expressive but simpler than ‘per-wire’ kinds, e.g. Simulink

\[ j : (\text{float}_D) \times (\text{float}_C) \rightarrow (\text{float}_D) \times (\text{float}_C) \]
Typing of function body gives its kind $k \in \{C, D, A\}$:

$$h : \text{float} \times \text{float} \xrightarrow{k} \text{float} \times \text{float}$$

Less expressive but simpler than ‘per-wire’ kinds, e.g. Simulink

$$j : (\text{float}_D) \times (\text{float}_C) \rightarrow (\text{float}_D) \times (\text{float}_C)$$
Typing

\[ G, H \vdash_c \text{der } y = 1.0 \text{ init } 0.0 \]

\[ G, H \vdash_D x = (0.0 \text{ fby } (x + 1)) \]

\[ G, H \vdash_c x' = (0.0 \text{ fby } (x + 1)) \]

\[ \text{every up(ez) init } 0.0 \]

\[ G, H \vdash_? \text{der } y = \cdots \text{ and } x = \cdots \]

\[ G, H \vdash_c \text{der } y = \cdots \text{ and } x' = \cdots \]

\[ G, H \vdash_? x = \cdots \text{ and } x = \cdots \]

Typing of function body gives its kind \( k \in \{C, D, A\} \):

\[ h : \text{float} \times \text{float} \xrightarrow{k} \text{float} \times \text{float} \]

Less expressive but simpler than ‘per-wire’ kinds, e.g. Simulink

\[ j : (\text{float}_D) \times (\text{float}_C) \longrightarrow (\text{float}_D) \times (\text{float}_C) \]
Typing

\[ G, H \vdash_C \text{der } y = 1.0 \ \text{init } 0.0 \]
\[ G, H \vdash_D x = (0.0 \ \text{fby} \ (x + 1)) \]
\[ G, H \vdash_C x' = (0.0 \ \text{fby} \ (x + 1)) \]
\[ \text{every up(ez) init } 0.0 \]

Typing of function body gives its kind \( k \in \{ C, D, A \} \):

\[ h : \text{float} \times \text{float} \xrightarrow{k} \text{float} \times \text{float} \]

Less expressive but simpler than ‘per-wire’ kinds, e.g. Simulink

\[ j : (\text{float}_D) \times (\text{float}_C) \longrightarrow (\text{float}_D) \times (\text{float}_C) \]
Typing

\[ G, H \vdash_c \text{der } y = 1.0 \text{ init } 0.0 \]
\[ G, H \vdash_D x = (0.0 \text{ fby } (x + 1)) \]

\[ G, H \vdash_c x' = (0.0 \text{ fby } (x + 1)) \]
\[ \text{every up(ez) init } 0.0 \]

\[ G, H \vdash_? \text{der } y = \cdots \text{ and } x = \cdots \]
\[ G, H \vdash_c \text{der } y = \cdots \text{ and } x' = \cdots \]
\[ G, H \vdash_D x = \cdots \text{ and } x = \cdots \]

Typing of function body gives its kind \( k \in \{C, D, A\} \):

\[ h : \text{float } \times \text{float} \xrightarrow{k} \text{float } \times \text{float} \]

Less expressive but simpler than ‘per-wire’ kinds, e.g. Simulink

\[ j : (\text{float}_D) \times (\text{float}_C) \longrightarrow (\text{float}_D) \times (\text{float}_C) \]
What about continuous automata?

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...

- “Restricted subset of Stateflow chart semantics” on page 16-26
- “Summary of Rules for Continuous-Time Modeling” on page 16-26

Rationale for Design Considerations

To guarantee the integrity — or correctness — of the results in Stateflow charts, as well as to support the Simulink solver, designers must employ a restricted subset of Stateflow chart semantics. The restricted semantics ensure that inputs do not depend on unpredictable factors — or side effects — such as:

- Simulink solver’s guess for number of minor intervals in a major time step
- Number of iterations required to stabilize the integration loop or zero crossings loop

By minimizing side effects, a Stateflow chart can maintain its state at minor time steps and, therefore, update state only during major time steps when make changes occur. Using this heuristic, a Stateflow chart can always compute outputs based on a consistent state for continuous-time.

A Simulink 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:

- State exit actions, which execute before leaving the state at the beginning of the transition
- Transition actions, which execute during the transition
- State entry actions, which execute after entering the new state at the end of the transition
- Condition actions on a transition, but only if the transition directly reaches a state

Consider the following chart.

In this example, the action `{n++}` executes even when conditions `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 transition actions because these actions execute in minor time steps.

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

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

Compute derivatives only in during actions.

A Simulink model uses continuous-time derivatives during minor time steps. The only part of a Stateflow chart that executes during minor time steps is the entering action. Therefore, you should compute derivatives in during actions to drive 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 made 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.

<table>
<thead>
<tr>
<th>‘Restricted subset of Stateflow chart semantics’</th>
</tr>
</thead>
<tbody>
<tr>
<td>restricts side-effects to major time steps</td>
</tr>
<tr>
<td>supported by warnings and errors in tool (mostly)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Our D/C/A/zero system extends naturally for the same effect</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>For both discrete (synchronous) and continuous (hybrid) contexts</th>
</tr>
</thead>
</table>
let hybrid ball(y0, y'0, start) =
let
rec init y = y0
and
automaton
| Await →
do

  der y = 0.0
until start then Bounce(y'0)
done

| Bounce(v) →
local c, y' in
do

  der y' = −9.81 init v
  and der y = y'
  and c = up(−. y)

until c on (y' < eps) then Await
| c then Bounce(−0.9 *. y')
done
end

in
y
let hybrid ball(y0, y'0, start) =
let
rec init y = y0
and
automaton
| Await →
do
  | der y = 0.0
until start then Bounce(y'0)
done
| Bounce(v) →
local c, y' in
do
  | der y' = −9.81 init v
  and der y = y'
  and c = up(−. y)
until c on (y' < eps) then Await
| c then Bounce(−0.9 ∗. y')
done
end

in
y

Automata à la Lucid Synchrone/SCADE 6

- (Parameterized) modes contain definitions, incl. automata
- mode-local definitions
- until: weak preemption (test after)
- unless: strong preemption (test before)
- then: enter-with-reset
- continue: entry-by-history
Automata à la Lucid Synchrone/SCADE 6

- (Parameterized) modes
  - contain definitions, incl. automata
- mode-local definitions
  - until: weak preemption (test after)
  - unless: strong preemption (test before)
  - then: enter-with-reset
  - continue: entry-by-history

```plaintext
let hybrid ball(y0, y'0, start) =
  let
    rec init y = y0
    and

  automaton
    | Await \rightarrow
      do
        der y = 0.0
    until start then Bounce(y'0)
  done

  | Bounce(v) \rightarrow
    local c, y' in
    do
      der y' = -9.81 init v
      and der y = y'
      and c = up(-. y)
    until c on (y' < eps) then Await
    | c then Bounce(-0.9 *. y')
  done
end

in y
```

Automata
Automata

Automata à la Lucid Synchrone/SCADE 6

- (Parameterized) modes contain definitions, incl. automata
- mode-local definitions
- **until**: weak preemption (test after)
- **unless**: strong preemption (test before)
- **then**: enter-with-reset
- **continue**: entry-by-history
Automata

let hybrid ball(y0, y'0, start) =
  let
  rec init y = y0
  and
  automaton
  | Await →
  do
    der y = 0.0
  until start then Bounce(y'0)
  done

  | Bounce(v) →
  local c, y' in
  do
    der y' = -9.81 init v
    and der y = y'
    and c = up(-. y)
  until c on (y' < eps) then Await
  | c then Bounce(-0.9 *. y')
  done
end

in y

Automata à la Lucid Synchrone/SCADE 6

- (Parameterized) modes contain definitions, incl. automata
- mode-local definitions
- until: weak preemption (test after)
- unless: strong preemption (test before)
- then: enter-with-reset
- continue: entry-by-history
Automata

```plaintext
let hybrid ball(y0, y'0, start) =
  let
    rec init y = y0
    and
  automaton
    | Await →
      do
        der y = 0.0
    until start then Bounce(y'0)
  done

| Bounce(v) →
  local c, y' in
  do
    der y' = −9.81 init v
    and der y = y'
    and c = up(−. y)
    until c on (y' < eps) then Await
    | c then Bounce(−0.9 *. y')
  done
end

in
y
```

Typing rules

- mode body: same kind as outer context
- until
  - guard : zero :: C/D
  - action :: D
- unless
  - guard : zero :: A
  - action :: D
Automata

```plaintext
let hybrid ball(y0, y'0, start) =
  let rec init y = y0
  and automaton
    | Await →
    do
      der y = 0.0
    until start then Bounce(y'0)
  done

| Bounce(v) →
  local c, y' in
  do
    der y' = -9.81 init v
    and der y = y'
    and c = up(-. y)
  until c on (y' < eps) then Await
  | c then Bounce(-0.9 *. y')
  done
end
```

Typing rules

- **mode body**: same kind as outer context
  - **until**
    - **guard**: zero :: C/D
    - **action**: D
  - **unless**
    - **guard**: zero :: A
    - **action**: D
Automata

let hybrid ball(y₀, y'₀, start) =
let
rec init y = y₀
and

automaton |
| Await →
| do
|     der y = 0.0
| until start then Bounce(y'₀)
done
zero :: C

| Bounce(v) →
| local c, y' in
| do
|     der y' = -9.81 init v
|     and der y = y'
|     and c = up(-. y)
| until c on (y' < eps) then Await
|     then Bounce(-0.9 *. y')
done
end
zero :: C

Typing rules

- mode body: same kind as outer context
- until
  - guard : zero :: C/D
  - action :: D
- unless
  - guard : zero :: A
  - action :: D
Automata

let hybrid ball(y0, y'0, start) =
let rec init y = y0
and

automaton
| Await →
do
  der y = 0.0
until start then Bounce(y'0)
done

| Bounce(v) →
local c, y' in
do
  der y' = -9.81 init v
  and der y = y'
  and c = up(-. y)
until c on (y' < eps) then Await
| c then Bounce(-0.9 *. y')
done
end

zero :: C  D

Typing rules

- mode body: same kind as outer context
- until
  - guard : zero :: C/D
  - action :: D
- unless
  - guard : zero :: A
  - action :: D
Automata

```
let hybrid ball(y0, y'0, start) =
  let rec init y = y0
  and

  automaton |
    Await →
      do
        der y = 0.0
      until start then Bounce(y'0)
      done

  | Bounce(v) →
    local c, y' in
    do
      der y' = -9.81 init v
      and der y = y'
      and c = up(-. y)
      until c on (y' < eps) then Await
      | c then Bounce(-0.9 *. y')
    done

  end

in y
```

Zero-crossing events

- Detected by the solver
- Constant mode during integration
- Cannot be negated (i.e. no reaction to absence)
- Less convenient than booleans?
  - `up(if b then 1.0 else -1.0)`
  - `· on ·: zero × bool → zero`
Automata

let hybrid ball(y0, y'0, start) =
let rec init y = y0
and
automaton
| Await →
do
der y = 0.0
until start then Bounce(y'0)
done
| Bounce(v) →
local c, y' in
do
der y' = −9.81 init v
and der y = y'
and c = up(−. y)
until c on (y' < eps) then Await
| c then Bounce(−0.9 *. y')
done
end

in
y

Zero-crossing events

- Detected by the solver
- Constant mode during integration
- Cannot be negated (i.e. no reaction to absence)
- Less convenient than booleans?
  - up(if b then 1.0 else −1.0)
  - · on · : zero × bool → zero

Zero-crossing events
▶ Detected by the solver
▶ Constant mode during integration
▶ Cannot be negated (i.e. no reaction to absence)
▶ Less convenient than booleans?
  ▶ up(if b then 1.0 else −1.0)
  ▶ · on · : zero × bool → zero
Automata

```haskell
let hybrid ball(y0, y'0, start) = 
  let rec init y = y0 
  and 
  automaton 
  | Await → 
  do 
    der y = 0.0 
  until start then Bounce(y'0) 
  done 
  | Bounce(v) → 
  local c, y' in 
  do 
    der y' = -9.81 init v 
    and der y = y' 
    and c = up(-. y) 
  until c on (y' < eps) then Await 
  | c then Bounce(-0.9 *. y') 
  done 
end 

in y
```

Zero-crossing events

- Detected by the solver
- Constant mode during integration
- Cannot be negated (i.e. no reaction to absence)
- Less convenient than booleans?
  - `up(if b then 1.0 else -1.0)`
  - `· on · : zero × bool → zero`
Automata

```plaintext
let hybrid ball(y0, y'0, start) =
  let rec init y = y0
  and

  automaton
  | Await →
  do
    der y = 0.0
  until start then Bounce(y'0)
done

  | Bounce(v) →
  local c, y' in
  do
    der y' = −9.81 init v
    and der y = y'
    and c = up(−. y)
  until c on (y' < eps) then Await
  | c then Bounce(−0.9 *. y')
done

in y
```

### Zero-crossing events

- Detected by the solver
- Constant mode during integration
- Cannot be negated (i.e. no reaction to absence)
- Less convenient than booleans?
  - `up(if b then 1.0 else −1.0)`
  - `· on · : zero × bool → zero`
Strong and weak transitions

$E_1$ unless $z$ $E_2$
Strong and weak transitions

$E_1$ unless $z$ to $E_2$
Strong and weak transitions

E₁ → E₂ unless z

E₁

E₂

transition

discrete
Strong and weak transitions

$E_1 \xrightarrow{\text{unless } z} E_2$
Strong and weak transitions

transition

unless $z$

$E_1 \rightarrow E_2$

$E_1 \rightarrow E_2$

$z \rightarrow E_1$

$E_1 \rightarrow E_2$

$E_2 \rightarrow E_2$

$E_1 \rightarrow E_2$

$E_2 \rightarrow E_2$

$Z$

$E_1 \rightarrow E_2$

$E_1 \rightarrow E_2$

$E_2 \rightarrow E_2$

$E_2 \rightarrow E_2$
Strong and weak transitions

transitions from $E_1$ to $E_2$ unless $z$ transitions from $E_1$ to $E_2$.
Strong and weak transitions

- Synchronous languages ignore the gaps between reactions
- But a hybrid language cannot
- Strong preemption: ok (*state entry on discrete step*)
Strong and weak transitions

- Synchronous languages ignore the gaps between reactions
- But a hybrid language cannot
- Strong preemption: ok \((state\ entry\ on\ discrete\ step)\)
Strong and weak transitions

- Synchronous languages ignore the gaps between reactions
- But a hybrid language cannot
- **Strong preemption:** ok (*state entry on discrete step*)
Strong and weak transitions

- **transition**
  
  \[ E_1 \xrightarrow{\text{unless } z} E_2 \]

- **discrete**
  
  \[ E_1 \quad E_1 \quad E_2 \quad E_2 \quad E_1 \quad E_2 \]

- **continuous**
  
  \[ E_1 \quad E_1 \quad E_2 \quad E_2 \]

- **Weak preemption:** 
  
  - **transition**
    
    \[ E_1 \xrightarrow{\text{until } z} E_2 \]
  
  - **discrete**
    
    \[ E_1 \quad E_1 \quad E_2 \quad E_2 \quad E_1 \quad E_2 \]
  
  - **continuous**
    
    \[ E_1 \quad E_1 \quad E_2 \quad E_2 \]
Strong and weak transitions

- Transition: $E_1$ unless $z$ to $E_2$
- Discrete: $E_1 \uparrow \uparrow \uparrow \uparrow \uparrow \downarrow E_1 \uparrow \uparrow \uparrow \uparrow E_2 \uparrow \uparrow \uparrow \uparrow E_2$
- Continuous: $E_1 \rightarrow \uparrow \uparrow \uparrow \uparrow \uparrow \downarrow E_2$

- Weak preemption: \ldots
Strong and weak transitions

Transition

\[ E_1 \xrightarrow{\text{unless } z} E_2 \]

Discrete

\[ E_1 \quad E_1 \quad E_2 \quad E_2 \quad E_2 \]

Continuous

\[ E_1 \quad \rightarrow \quad E_1 \rightarrow E_2 \rightarrow \]

Weak preemption: \ldots
Strong and weak transitions

transition

unless $z$

$E_1 \rightarrow E_2$

$E_1$ until $z$

$E_1 \rightarrow E_2$

$E_1$

$E_2$

$E_2$

$E_1$

$E_1$

$E_2$

$E_2$

$Z$

$Z$

Weak preemption: ...
Strong and weak transitions

transition

\[ E_1 \quad \text{unless} \quad z \quad \text{until} \quad z \quad E_2 \]

\[ E_1 \quad E_1 \quad E_2 \quad E_2 \quad E_2 \]

\[ E_1 \quad E_1 \quad E_1 \quad \]

\[ E_1 \quad E_2 \quad E_2 \quad \]

\[ E_1 \quad E_1 \quad E_2 \quad E_2 \quad \]

\[ E_1 \quad E_2 \quad \]

Weak preemption: ...
Strong and weak transitions

- **Strong transition:**
  - *Unless* $z$
  - $E_1 \rightarrow z \rightarrow E_2$

- **Weak transition:**
  - *Until* $z$
  - $E_1 \rightarrow z \rightarrow E_2$

> Weak preemption: ...
Strong and weak transitions

transition


discrete


continuous


Weak preemption: ...
Strong and weak transitions

- **Strong transition:**
  - $E_1$ unless $z$ to $E_2$
  - $E_1 \rightarrow E_2$

- **Weak transition:**
  - $E_1$ until $z$ to $E_2$
  - $E_1 \rightarrow E_2$

- **Discrete transition:**
  - $Z$
  - $E_1 \rightarrow E_1 \rightarrow E_2 \rightarrow E_2$

- **Continuous transition:**
  - $Z$
  - $E_1 \rightarrow [E_1 \rightarrow E_2 \rightarrow]$

- **Weak preemption:** trickier
Strong and weak transitions

- **Transition**
  - $E_1$ unless $z$ to $E_2$

- **Discrete**
  - $E_1$ $E_1$ $E_2$ $E_2$ $E_2$

- **Continuous**
  - $E_1$ $E_1$ $E_2$ $E_2$ $E_2$

- Weak preemption: trickier
- State exit on discrete step
Strong and weak transitions

▶ Weak preemption: trickier
▶ state exit on discrete step
Strong and weak transitions

- **Weak preemption**: trickier
- state exit on discrete step
- need an extra discrete step for state entry
Execution (Simulation)

- Only \( d \) may have side effects and change the discrete state (\( \sigma \))
- Both \( f \), nor \( g \) must be combinatorial
- \( D' \) ensures correct initialization after weak transitions
Execution (Simulation)

- Only $d$ may have side effects and change the discrete state ($\sigma$)
- Both $f$, nor $g$ must be combinatorial
- $D'$ ensures correct initialization after weak transitions

- Cf. Simulink: major and minor time steps, time always advances
- Cf. Ptolemy: iteration in $D$ until $\sigma$ is stable (no need for $D'$)
Solver execution

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

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - Cannot change state within \( f \) or \( g \)
  - Guaranteed for well-typed programs
Solver execution

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

- Bigger and bigger steps (bound by $h_{min}$ and $h_{max}$)
- $t$ does not necessarily advance monotonically
  - Cannot change state within $f$ or $g$
  - Guaranteed for well-typed programs
Solver execution

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

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - Cannot change state within \( f \) or \( g \)
  - Guaranteed for well-typed programs
Solver execution

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

- Bigger and bigger steps (bound by \( h_{\text{min}} \) and \( h_{\text{max}} \))
- \( t \) does not necessarily advance monotonically
  - Cannot change state within \( f \) or \( g \)
  - Guaranteed for well-typed programs
Solver execution

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

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - Cannot change state within \( f \) or \( g \)
  - Guaranteed for well-typed programs
Solver execution

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

- Bigger and bigger steps (bound by $h_{\text{min}}$ and $h_{\text{max}}$)
- $t$ does not necessarily advance monotonically
  - Cannot change state within $f$ or $g$
  - Guaranteed for well-typed programs
Solver execution

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

- Bigger and bigger steps (bound by \( h_{\text{min}} \) and \( h_{\text{max}} \))
- \( t \) does not necessarily advance monotonically
  - Cannot change state within \( f \) or \( g \)
  - Guaranteed for well-typed programs
Solver execution

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

1. approximation error too large

- Bigger and bigger steps (bound by $h_{min}$ and $h_{max}$)
- $t$ does not necessarily advance monotonically
  - Cannot change state within $f$ or $g$
  - Guaranteed for well-typed programs
Solver execution

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

1. approximation error too large

- Bigger and bigger steps (bound by \(h_{\text{min}}\) and \(h_{\text{max}}\))
- \(t\) does not necessarily advance monotonically
  - Cannot change state within \(f\) or \(g\)
  - Guaranteed for well-typed programs
Sander van der Linden

Solver execution
Give solver two functions: \( dy = f_\sigma(t, y), \ upz = g_\sigma(t, y) \)

1. approximation error too large

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - Cannot change state within \( f \) or \( g \)
  - Guaranteed for well-typed programs
Solver execution

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

1. approximation error too large

- Bigger and bigger steps (bound by \( h_{\min} \) and \( h_{\max} \))
- \( t \) does not necessarily advance monotonically
- Cannot change state within \( f \) or \( g \)
- Guaranteed for well-typed programs
Solver execution

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

1. approximation error too large

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - Cannot change state within \( f \) or \( g \)
  - Guaranteed for well-typed programs
**Solver execution**

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

1. approximation error too large

Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))

- \( t \) does not necessarily advance monotonically

- Cannot change state within \( f \) or \( g \)

- Guaranteed for well-typed programs

2. expression crosses zero
### Solver execution

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

1. **approximation error too large**

2. **expression crosses zero**

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - Cannot change state within \( f \) or \( g \)
  - Guaranteed for well-typed programs
Solver execution

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

1. approximation error too large

2. expression crosses zero

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - Cannot change state within \( f \) or \( g \)
  - Guaranteed for well-typed programs
Give solver two functions: \( dy = f_\sigma(t, y) \), \( upz = g_\sigma(t, y) \)

1. approximation error too large

2. expression crosses zero

- Bigger and bigger steps (bound by \( h_{\text{min}} \) and \( h_{\text{max}} \))
- \( t \) does not necessarily advance monotonically
  - Cannot change state within \( f \) or \( g \)
  - Guaranteed for well-typed programs
Solver execution

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

1. approximation error too large

2. expression crosses zero

- Bigger and bigger steps (bound by \( h_{min} \) and \( h_{max} \))
- \( t \) does not necessarily advance monotonically
  - Cannot change state within \( f \) or \( g \)
  - Guaranteed for well-typed programs
Source-to-source transformation

lexing/parsing → typing/caus./init. → automata → ... → scheduling → code gen.

(σ)
Pro: simpler definition of ODE

Con: subtle invariant over intermediate language
Source-to-source transformation

- lexing/parsing
- typing/caus./init.
- automata
- scheduling
- code gen.

(\(f_\sigma, g_\sigma, d_\sigma\))

Data-flow + Auto. + ODE $\xrightarrow{ode}$ Data-flow + Auto.

Data-flow + ODE $\xrightarrow{auto}$ Data-flow $\xrightarrow{codegen}$ Imperative code

- Pro: intermediate result is well-typed
- Pro/Con: ODE code must include cases for automata
let hybrid ball(y0, y'0, start) =
  let
  rec init y = y0
  and automaton
    | Await →
      do
      der y = 0.0
      until start then Bounce(y'0)
    done

    | Bounce(v) →
      local c, y' in
      do
      der y' = −9.81 init v
      and der y = y'
      and c = up(−. y)
      until c on (y' < eps) then Await
      | c then Bounce(−0.9 *. y')
    done
  end

in
y
Source-to-source transformation details

\[
\begin{align*}
\text{let hybrid ball}(y_0, y'_0, \text{start}) = & \ \
\text{let rec init } y = y_0 \\
& \text{and automaton} \\
& | \text{Await } \rightarrow \\
& \hspace{1em} \text{do} \\
& \hspace{2em} \text{der } y = 0.0 \\
& \hspace{2em} \text{until start then } \text{Bounce}(y'_0) \\
& \text{done} \\
& | \text{Bounce}(v) \rightarrow \\
& \hspace{1em} \text{local } c, y' \text{ in} \\
& \hspace{2em} \text{do} \\
& \hspace{3em} \text{der } y' = -9.81 \text{ init } v \\
& \hspace{3em} \text{and der } y = y' \\
& \hspace{3em} \text{and } c = \text{up}(\text{-}y) \\
& \hspace{2em} \text{until c on } (y' < \text{eps}) \text{ then } \text{Await} \\
& \hspace{2.5em} | c \text{ then } \text{Bounce}(\text{-}0.9 * y') \\
& \text{done} \\
& \text{in } y \\
\end{align*}
\]

\[
\begin{align*}
\text{let node ball}((y_0, y'_0, \text{start}), ((ly, ly'), z)) = & \ \
\text{let rec } y = y_0 \rightarrow \text{ ly} \\
& \text{and automaton} \\
& | \text{Await } \rightarrow \\
& \hspace{1em} \text{do} \\
& \hspace{2em} \text{dy}' = 0.0 \\
& \hspace{2em} \text{and } y' = \text{ly}' \\
& \hspace{2em} \text{and } dy = 0.0 \\
& \hspace{2em} \text{and } \text{upz} = (0.0, \text{false}) \\
& \hspace{2em} \text{until start then } \text{Bounce}(y'_0) \text{ done} \\
& | \text{Bounce}(v) \rightarrow \\
& \hspace{1em} \text{local } c \text{ in} \\
& \hspace{2em} \text{do} \\
& \hspace{3em} \text{dy}' = -9.81 \\
& \hspace{3em} \text{and } y' = v \rightarrow \text{ ly}' \\
& \hspace{3em} \text{and } dy = y' \\
& \hspace{3em} \text{and } c = z \\
& \hspace{3em} \text{and upz} = (\text{-}y, \text{true}) \\
& \hspace{2em} \text{until c } & (y' < \text{eps}) \text{ then } \text{Await} \\
& \hspace{2.5em} | c \text{ then } \text{Bounce}(\text{-}0.9 * y') \\
& \text{done} \\
& \text{in } (y, ((y, y'), (dy, dy'), \text{upz}))
\end{align*}
\]

▶ Source-to-source transformation (to give \(f_\sigma, g_\sigma, d_\sigma\)
Source-to-source transformation details

let hybrid ball(y₀, y₀', start) =
  let rec init y = y₀
  and automaton
      | Await →
      do
der y = 0.0
  until start then Bounce(y₀')
  done

| Bounce(v) →
  local c, y' in
  do
der y' = −9.81 init v
  and der y = y'
  and c = up(−. y)
  until c on (y' < eps) then Await
  | c then Bounce(−0.9 * . y')
  done
end

let node ball(((y₀, y₀'), start), ((l_y, l_y'), z)) =
  let rec y = y₀ -> l_y
  and automaton
      | Await →
      do
dy' = 0.0
  and y' = l_y'
  and dy = 0.0
  and upz = (0.0, false)
  until start then Bounce(y₀')
  done

| Bounce(v) →
  local c in
  do
dy' = −9.81
  and y' = v -> l_y'
  and dy = y'
  and c = z
  and upz = (−. y, true)
  until c & (y' < eps) then Await
  | c then Bounce(−0.9 * . y')
  done
end

▶ Source-to-source transformation (to give \( f_\sigma, g_\sigma, d_\sigma \))

▶ Transform each hybrid function into a discrete one
Continuous-state definitions are ‘externalized’ via inputs and outputs
Source-to-source transformation details

Continuous-state definitions are ‘externalized’ via inputs and outputs

Initialization is a discrete action; branch entry must be restricted
Source-to-source transformation details

Continuous-state definitions are ‘externalized’ via inputs and outputs
Initialization is a discrete action; branch entry must be restricted
Source-to-source transformation details

Continuous-state definitions are ‘externalized’ via inputs and outputs
 Initialization is a discrete action; branch entry must be restricted
 Extending the scope mandates additional definitions for other modes
Source-to-source transformation details

let hybrid ball(y0, y'0, start) =
let rec init y = y0
and automaton
  | Await →
    do
      der y = 0.0
    until start then Bounce(y'0)
done

| Bounce(v) →
  local c, y' in
  do
    der y' = -9.81 init v
    and der y = y
    and c = up(- y)
  until c on (y' < eps) then Await
  | c then Bounce(-0.9 * y')
done

end

in y

| Bounce(v) →
  local c in
  do
    dy' = -9.81
    and y' = v -> ly'
    and dy = y'
    and c = z
    and upz = (- y, true)
  until c & (y' < eps) then Await
  | c then Bounce(-0.9 * y')
done

end

in (y, ((y, y'), (dy, dy'), upz))

▶ Zero-crossing operators, up(·), are also ‘externalized’
▶ Detection always occurs externally; boolean values internally
Source-to-source transformation details

```plaintext
let hybrid ball(y0, y'0, start) =
  let rec init y = y0
  and automaton
    | Await ->
      do
        der y = 0.0
      until start then Bounce(y'0)
    done

  | Bounce(v) ->
    local c, y' in
    do
      der y' = -9.81 init v
      and der y = y'
      and c = up(-. y)
      until c on (y' < eps) then Await
      | c then Bounce(-0.9 * . y')
    done
  end

let node ball(((y0, y'0, start), ((ly, ly'), z))
  let rec y = y0 -> ly
  and automaton
    | Await ->
      do
        dy' = 0.0
        and y' = ly'
        and dy = 0.0
        and upz = (0.0, false)
      until start then Bounce(y'0) done

  | Bounce(v) ->
    local c in
    do
      dy' = -9.81
      and y' = v -> ly'
      and dy = y'
      and c = z
      and upz = (-. y, true)
      until c & (y' < eps) then Await
      | c then Bounce(-0.9 * . y')
    done
  end

in
  (y, ((y, y'), (dy, dy'), upz))
```

- Zero-crossing operators, `up(·)`, are also ‘externalized’
- Detection always occurs externally; boolean values internally
- Additional definitions in inactive modes involve a slight technicality
Demonstrations

- **Bouncing ball** (standard)
- **Bang-bang temperature controller** *(Simulink/Stateflow)*
- **Sticky Masses** *(Ptolemy)*
- ...
Conclusions and Future Work

Conclusions

- Synchronous languages should and can properly treat hybrid systems
- There are three good reasons for doing so:
  1. To exploit existing compilers and techniques
  2. For programming the discrete subcomponents
  3. To clarify underlying principles and guide language design/semantics
- A prototype compiler in OCaml using Sundials CVODE solver

Future Work

- clock calculus, higher order functions
- integrate multiple solvers
- real-time simulation (compromise accuracy and execution time)