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. École normale supérieure (LIENS)

EMSOFT 2011, ES Week, October 9–14, Taipei, Taiwan
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 complex discrete systems and modeling 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 complex discrete systems and modeling 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 complex discrete systems and modeling their physical environments
### Aim

**Programming languages perspective:**

<table>
<thead>
<tr>
<th>Component</th>
<th>Understanding</th>
<th>Tools/Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>purely discrete data-flow</td>
<td>well understood</td>
<td>(Lustre, SCADE 6)</td>
</tr>
<tr>
<td>purely continuous</td>
<td>well understood</td>
<td>(Numerical solvers, Simulink)</td>
</tr>
<tr>
<td>hier. automata (disc.)</td>
<td>well understood</td>
<td>(Statecharts, Esterel)</td>
</tr>
<tr>
<td>data-flow + hier. auto.</td>
<td>well understood</td>
<td>(SCADE 6, Esterel v7)</td>
</tr>
</tbody>
</table>

**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 complex discrete systems and modeling their physical environments
Approach

- Add Ordinary Differential Equations to an existing synchronous language

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


- 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)
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:

```plaintext
let node sum(x) = cpt where
  rec cpt = x → (pre cpt +. x)
```
Which programs make sense?

Given:

```plaintext
let node sum(x) = cpt where
  rec cpt = x → (pre cpt +. x)
```

Evaluate:

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

Given:

```ml
let node sum(x) = cpt where
  rec cpt = x → (pre cpt +. x)
```

Evaluate:

```ml
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: type and reject
Which programs make sense?

Given:

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

Evaluate:

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

Interpretation:

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

Given:

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

Evaluate:

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

Interpretation:

- Option 1: \( \mathbb{N} \subseteq \mathbb{R} \)
- Option 2: depends on solver
- Option 3: 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:

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

\[
\text{rec cpt} = x \rightarrow (\text{pre cpt} +. x)
\]

Evaluate:

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

and

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

Interpretation:

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

Given:

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

Evaluate:

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

Interpretation:

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

Given:

```
let node sum(x) = cpt where
rec cpt = x \rightarrow (pre 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: type and reject

Explicitly relate simulation and logical time (using zero-crossings)
Try to minimize the effects of solver parameters and choices
Basic typing

The type language

\[ bt ::= \text{float} \mid \text{int} \mid \text{bool} \mid \text{zero} \]

\[ t ::= bt \mid t \times t \mid \beta \]

\[ \sigma ::= \forall \beta_1, \ldots, \beta_n.t \xrightarrow{k} t \]

\[ k ::= D \mid C \mid A \]

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}(\cdot) : \forall \beta.\beta \xrightarrow{D} \beta\\
\cdot \text{fby} \cdot : \forall \beta.\beta \times \beta \xrightarrow{D} \beta\\
\text{up}(\cdot) : \text{float} \xrightarrow{C} \text{zero}\\
\cdot \text{on} \cdot : \text{zero} \times \text{bool} \xrightarrow{A} \text{zero}
Design Considerations for Continuous-Time Modeling in Stateflow Charts

In this section...
- “Rationale for Design Considerations” on page 16-26
- “Summary of Rules for Continuous-Time Modeling” on page 16-26

Rationale for Design Considerations

To guarantee the stability — or correctness — of the results in continuous-time modeling, you must use a restricted subset of Stateflow chart semantics. The restricted semantics ensure that
- 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 major changes occur. Using this feature, a Stateflow chart can achieve side effects only during major time steps. A Stateflow chart generates informative errors to help you correct semantic violations.

Summary of Rules for Continuous-Time Modeling

These are the rules for modeling continuous-time Stateflow charts.

Use discrete variables to govern conditions in state transitions. Therefore, write to local data only in actions that execute during transitions, as follows:
- State exit actions, which execute before leaving the state at the beginning of the transition
- Transition actions, which execute during the transition
- State entry actions, which execute after entering the new state at the end of the transition
- Condition actions on a transition, but only if the transition directly reaches a state

Consider the following chart.

In this example, the action {n++} executes even when conditions c2 and c3 are false. In that case, c gets updated in a minor time step because there is no state transition.

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

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

To guarantee the integrity — or smooth output — of a chart, you should update local data (continuous or discrete) only during major time steps in continuous-time modeling. This rule applies to continuous-time charts because you cannot call functions during minor time steps. You can call Simulink functions in state entry or exit actions and transition actions. However, if you try to call Simulink functions in state during actions or transition conditions, an error message appears when you simulate your model.

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

Compute derivatives only in major actions.

A Simulink model reads continuous-time derivatives during minor time steps. The only part of a Stateflow chart that executes during minor time steps is the crossing a Simulink chart always computes outputs based on a constant state for continuous-time. 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. These values 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 unstable in continuous-time. For example, the following model generates an error if the chart uses a continuous update method.

We recommend restricting your Stateflow chart to the following rules.

- Do not call Simulink functions in state during actions or transition conditions.
- Do not use Simulink functions in minor time steps.
- Do not update local data in state during actions.
- Do not call Simulink functions in state during actions or transition conditions.
- Do not use input events in continuous-time charts.

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

For both discrete (synchronous) and continuous (hybrid) contexts.
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
```

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

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
```

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

```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
```

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

\[
\begin{align*}
\text{let hybrid ball}(y_0, y_0', \text{start}) &= \quad\text{let} \\
\quad\text{rec init } y &= y_0 \\
\quad\text{and} \\
\text{automaton} &\quad\mid \text{Await} \rightarrow \\
&\quad\begin{aligned}
\text{do} &\quad\text{der} y = 0.0 \\
\text{until start then} &\text{Bounce}(y_0') \\
\text{done} \\
\end{aligned} \\
\mid \text{Bounce}(v) \rightarrow \\
&\quad\begin{aligned}
\quad\text{local } c, y' \text{ in} \\
\text{do} &\quad\text{der} y' = -9.81 \text{ init } v \\
&\quad\text{and der} y = y' \\
&\quad\text{and } c = \text{up}(\rightarrow \text{. } y) \\
\text{until c on } (y' < \text{eps}) \text{ then} &\text{Await} \\
\mid c \text{ then} &\text{Bounce}(0.9 \text{ . } y') \\
\text{done} \\
\end{aligned}
\end{align*}
\]

Typing rules

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

let hybrid ball(y0, y'0, start) =
  let
    rec init y = y0
  and
  automaton
    | Await →
      do
        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
zero :: C

| Bounce(v) →
  local c, y' in
  do
    der y' = -9.81 init v
    and der y = y'
    and c = up(-. y)

  until on (y' < eps) then Await
  | then Bounce(-0.9 *. y')
done
end
zero :: C

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

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

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

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
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
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
Strong and weak transitions

transition \( E_1 \) unless \( z \) discrete \( E_2 \)
Strong and weak transitions

transition

$E_1 \xrightarrow{\text{unless } z} E_2$

discrete

$E_1$
Strong and weak transitions

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

\[ E_1 \quad E_1 \quad E_1 \]

\[ E_2 \]
Strong and weak transitions

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

\[ E_1 \quad E_1 \quad E_2 \]

\[ Z \]

\[ \text{transition} \quad \text{discrete} \]
Strong and weak transitions

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

\[ \begin{align*}
E_1 & \quad \uparrow \quad \uparrow \quad \uparrow \\
E_1 & \quad E_1 & \quad E_2 & \quad E_2 
\end{align*} \]

\[ Z \]

transition

discrete
Strong and weak transitions

transition

unless $z$

$E_1 \rightarrow E_2$

discrete

$Z$

$E_1 \uparrow E_1 \uparrow E_2 \uparrow E_2 \uparrow E_2$
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

▶ Weak preemption: ...
Strong and weak transitions

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

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

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

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

- **discrete**
  - $E_1$ to $E_1$ to $E_2$ to $E_2$ to $E_2$

- **continuous**
  - $E_1$ to $E_2$

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

Weak preemption: ...
Strong and weak transitions

Transition

Discrete

Continuous

- Weak preemption: ...
Strong and weak transitions

transition

unless $z$

continuous

weak preemption: ...
Strong and weak transitions

Transition

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

Discrete

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

Continuous

\[ E_1 \quad E_2 \]

Weak preemption: ...
Strong and weak transitions

transition

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

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

discrete

\[ \begin{array}{cccccc}
E_1 & E_1 & E_2 & E_2 & E_2 \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow
\end{array} \]

continuous

\[ \begin{array}{cccccc}
E_1 & E_1 & E_1 & E_2 & E_2 \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow
\end{array} \]

Weak preemption: trickier
Strong and weak transitions

**transition**

\[ E_1 \overset{\text{unless } z}{\rightarrow} E_2 \]

**discrete**

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

**continuous**

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

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

Transition

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

Discrete

Z

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

Continuous

Z

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

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

- Strong transition
  - unless
  - until

- 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 \( h_{\text{min}} \) 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: \( dy = f_\sigma(t, y) \), \( upz = g_\sigma(t, y) \)

- 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: \( dy = f_\sigma(t, y) \), \( upz = g_\sigma(t, y) \)

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 \( h_{\text{min}} \) 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 \( h_{\text{min}} \) 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

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 \( h_{\text{min}} \) 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

[Diagram showing approximation error]

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.
Source-to-source transformation

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

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

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

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

- lexing/parsing
- typing/caus./init.
- automata
- ... (f_σ, g_σ, d_σ)
- ODEs
- scheduling
- code gen.

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

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

- Pro: simpler definition of ODE
- Con: subtle invariant over intermediate language
Source-to-source transformation

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

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

Data-flow + ODE \(\xrightarrow{ode}\) 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
in
y
let hybrid ball\((y_0, y_0', \text{start}) = \)

let

rec init \(y = y_0\)

and automaton

| Await \(\rightarrow\)

| do

| der \(y = 0.0\)

| until start \(\rightarrow\) then Bounce\((y_0')\)

done

| Bounce\((v) \rightarrow\)

| local \(c, y'\) in

| do

| der \(y' = -9.81\) init \(v\)

| and der \(y = y'\)

| and \(c = \text{up}(\neg y)\)

| until \(c \& \neg y' < \text{eps}\) \(\rightarrow\) then Await

| \(c \rightarrow\) then Bounce\((-0.9 \times y')\)

| done

end

in \(y\)

let node ball\(((y_0, y_0', \text{start}), ((l_y, l_y'), z)) = \)

let

rec \(y = y_0 \rightarrow l_y\)

and automaton

| Await \(\rightarrow\)

| do

| dy' = 0.0

| and \(y' = l_y'\)

| and \(dy = 0.0\)

| and \(\text{upz} = (0.0, \text{false})\)

| until start \(\rightarrow\) then Bounce\((y_0')\)

| done

| Bounce\((v) \rightarrow\)

| local \(c\) in

| do

| dy' = -9.81

| and \(y' = v \rightarrow l_y'\)

| and \(dy = y'\)

| and \(c = z\)

| and \(\text{upz} = (\neg y, \text{true})\)

| until \(c \& \neg y' < \text{eps}\) \(\rightarrow\) then Await

| \(c \rightarrow\) then Bounce\((-0.9 \times y')\)

| done

end

in \((y, ((y, y'), (dy, dy'), \text{upz}))\)

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

**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

```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
      and der y = y'
      and c = up(-. y)
    until c on (y' < eps) then Await
    | c then Bounce(-0.9 *. y')
  done

in y
```

```plaintext
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

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

- 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
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

```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

let node ball(((y0, y'0, start), ((ly, ly'), z))
  let rec y = y0 -> ly
  and automaton |
    | Await → do
    |   dy' = 0.0
    | 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
```
Source-to-source transformation details

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

let node ball((y0, y'0, start), ((ly, ly'), z)) =
  let rec y = y0 -> ly
  and automaton |
    Await \to |
      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) \to |
      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 |
      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)