Programming hybrid systems with synchronous languages

Marc Pouzet\textsuperscript{1,2,3} 

Albert Benveniste\textsuperscript{3}  
Timothy Bourke\textsuperscript{3,1}  
Benoît Caillaud\textsuperscript{3}

1. École normale supérieure (LIENS)  
2. Université Pierre et Marie Curie  
3. INRIA

CSDM 2011, December 7–9, Paris
Reactive systems

- They react continuously to the external environment.
- At the speed imposed by this environment.
- Statically bounded memory and response time.

Conciliate three notions in the programming model:

- Parallelism, concurrency while preserving determinism.
  
  *e.g.* control at the same time rolling and pitching
  → parallel description of the system

- Strong temporal constraints.
  
  *e.g.* the physics does not wait!
  → temporal constraints should be expressed in the system

- Safety is important (critical systems).
  → well founded languages, verification methods
parallel processes communicating through data-flows
communication in zero time: data is available as soon as it is produced.
a global logical time scale even though individual rhythms may differ
these drawings are not so different from actual computer programs
SAO (Spécification Assistée par Ordinateur)—Airbus 80’s

Describe the system as block diagrams (synchronous communicating machines)
SCADE 4 (Safety Critical Application Development Env. – Esterel-Tech.)

From computer assisted drawings to executable (sequential/parallel) code!
Lustre: a dataflow programming language


Programming with streams
Programming with streams

constants 1 = 1 1 1 1 1 ...
Lustre: a dataflow programming language


Programming with streams

constants

\[
1 \quad = \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad \ldots
\]

operators

\[
x + y \quad = \quad x_0 + y_0 \quad x_1 + y_1 \quad x_2 + y_2 \quad x_3 + y_3 \quad \ldots
\]

\[
(z = x + y \text{ means that at every instant } i : z_i = x_i + y_i)
\]
Lustre: a dataflow programming language


Programming with streams

constants\[1 = 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad \ldots\]

operators\[x + y = x_0 + y_0 \quad x_1 + y_1 \quad x_2 + y_2 \quad x_3 + y_3 \quad \ldots\]

\((z = x + y\text{ means that at every instant } i: z_i = x_i + y_i)\)

unit delay\[0 \text{ fby } (x + y) = 0 \quad x_0 + y_0 \quad x_1 + x_1 \quad x_2 + x_2 \quad \ldots\]
Programming with streams

constants

\[
1 = 1 1 1 1 \ldots
\]

operators

\[
x + y = x_0 + y_0 \quad x_1 + y_1 \quad x_2 + y_2 \quad x_3 + y_3 \quad \ldots
\]

\[
(z = x + y \text{ means that at every instant } i : z_i = x_i + y_i)
\]

unit delay

\[
0 \text{ fby } (x + y) = 0 \quad x_0 + y_0 \quad x_1 + x_1 \quad x_2 + x_2 \quad \ldots
\]

\[
\text{pre } (x + y) = \text{nil} \quad x_0 + y_0 \quad x_1 + x_1 \quad x_2 + x_2 \quad \ldots
\]
Lustre: a dataflow programming language


Programming with streams

constants

\[ 1 = 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad \ldots \]

operators

\[ x + y = x_0 + y_0 \quad x_1 + y_1 \quad x_2 + y_2 \quad x_3 + y_3 \quad \ldots \]

\((z = x + y)\) means that at every instant \(i\) : \(z_i = x_i + y_i\)

unit delay

\[ 0 \text{ fby } (x + y) = 0 \quad x_0 + y_0 \quad x_1 + x_1 \quad x_2 + x_2 \quad \ldots \]

\[ \text{pre } (x + y) = \text{nil} \quad x_0 + y_0 \quad x_1 + x_1 \quad x_2 + x_2 \quad \ldots \]

\[ 0 \rightarrow \text{pre } (x + y) = 0 \quad x_0 + y_0 \quad x_1 + x_1 \quad x_2 + x_2 \quad \ldots \]
Lustre: a dataflow programming language

```
let node iir_filter_2 x = y
where
  rec
    y = a * x + (0.0 fby u) 
    and
    u = b * x - d * y + (0.0 fby v) 
    and
    v = c * x - e * y
```

\[ v = c \times x - e \times y \]
Lustre: a dataflow programming language

```
let node iir_filter_2 x = y
  where
    rec
      y = a * x + (0.0 fby u)
      and
        u = b * x - d * y + (0.0 fby v)
      and
        v = c * x - e * y
```

\[
u = b \times x - d \times y + (0.0 \ fby \ v)\]

\[
\text{and} \ v = c \times x - e \times y
\]
Lustre: a dataflow programming language

$$\text{let node iir_filter_2 x = y where }$$
$$\text{rec y = a \ast x + (0.0 \ fby u)}$$
$$\text{and u = b \ast x - d \ast y + (0.0 \ fby v)}$$
$$\text{and v = c \ast x - e \ast y}$$
Lustre: a dataflow programming language

\[
\begin{align*}
\text{let } \text{node} \ iir\_filter\_2 \ x &= y \\
\text{rec} \quad y &= a \ast x + (0.0 \ \text{fby} \ u) \\
\text{and} \quad u &= b \ast x - d \ast y + (0.0 \ \text{fby} \ v) \\
\text{and} \quad v &= c \ast x - e \ast y
\end{align*}
\]
Lustre: a dataflow programming language

let node iir_filter_2 x = y where

rec y = a * x + (0.0 fby u)

and u = b * x - d * y + (0.0 fby v)

and v = c * x - e * y
Lustre: beautiful ideas

- A simple and pure notion of execution in discrete time
- Parallel composition is well-defined and deterministic: very important in practice for reproducibility
- Parallelism is compiled: programs can be translated into efficient sequential
- The code executes in bounded memory and bounded time
- Programs are finite-state and can be verified by model-checking
Lustre: beautiful ideas

- A simple and pure notion of execution in **discrete time**

- Parallel composition is
  - well-defined
  - **deterministic**: very important in practice for reproducibility

- Parallelism is compiled: programs can be translated into efficient sequential

- The code executes in **bounded memory** and **bounded time**

- Programs are **finite-state** and can be verified by model-checking
Lustre: beautiful ideas

- A simple and pure notion of execution in discrete time

- Parallel composition is
  - well-defined
  - deterministic: very important in practice for reproducibility

- Parallelism is compiled: programs can be translated into efficient sequential

- The code executes in bounded memory and bounded time

- Programs are finite-state and can be verified by model-checking
Lustre: beautiful ideas

- A simple and pure notion of execution in discrete time

- Parallel composition is
  - well-defined
  - deterministic: very important in practice for reproducibility

- Parallelism is compiled: programs can be translated into efficient sequential

- The code executes in bounded memory and bounded time

- Programs are finite-state and can be verified by model-checking
Lustre: beautiful ideas

- A simple and pure notion of execution in discrete time

- Parallel composition is
  - well-defined
  - deterministic: very important in practice for reproducibility

- Parallelism is compiled: programs can be translated into efficient sequential

- The code executes in bounded memory and bounded time

- Programs are finite-state and can be verified by model-checking
Lustre: beautiful ideas

- A simple and pure notion of execution in discrete time

- Parallel composition is
  - well-defined
  - deterministic: very important in practice for reproducibility

- Parallelism is compiled: programs can be translated into efficient sequential

- The code executes in bounded memory and bounded time

- Programs are finite-state and can be verified by model-checking

No need to write control programs in C!
Lustre: beautiful ideas

- A simple and pure notion of execution in discrete time

- Parallel composition is
  - well-defined
  - deterministic: very important in practice for reproducibility

- Parallelism is compiled: programs can be translated into efficient sequential

- The code executes in bounded memory and bounded time

- Programs are finite-state and can be verified by model-checking

No need to write control programs in C!

Lustre can be extended in several ways...
Extended dataflow programming: automata

```plaintext
let node counter (flip, stop) = x
where
rec lx = 0 fby x
and automaton
    | Up →
        do
        x = lx + 1
        until flip then Down
        | stop then Stop(true)
        done
    | Down →
        do
        x = lx - 1
        until flip then Up
        | stop then Stop(false)
        done
    | Stop(was_up) →
        do
        x = lx
        until flip & was_up then Up
        | flip then Down
        done
eend
```

- Parallel composition of dataflow equations and automata
- \( x \) has a different definition in each mode
- But only a single definition in a reaction

Extended dataflow programming: automata

Let node counter (flip, stop) = x
where
- lx = 0.
- and automaton
  | Up \rightarrow do
  | \hspace{1cm} x = lx + 1
  | \hspace{1cm} until flip then Down
  | \hspace{1cm} | stop then Stop(true)
  | done

| Down \rightarrow do
| \hspace{1cm} x = lx - 1
| \hspace{1cm} until flip then Up
| \hspace{1cm} | stop then Stop(false)
| done

| Stop(was_up) \rightarrow do
| \hspace{1cm} x = lx
| \hspace{1cm} until flip & was_up then Up
| | flip then Down
| done
end

- Parallel composition of dataflow equations and automata
  - x has a different definition in each mode
  - But only a single definition in a reaction
Extended dataflow programming: automata

```
let node counter (flip, stop) = x
where
  rec l\$ = 0 fby x
and automaton
  | Up →
  \hspace{1em} do
  \hspace{2em} x = l\$ + 1
  \hspace{2em} until flip then Down
  \hspace{2em} | stop then Stop(true)
  done

  | Down →
  \hspace{1em} do
  \hspace{2em} x = l\$ - 1
  \hspace{2em} until flip then Up
  \hspace{2em} | stop then Stop(false)
  done

  | Stop(was_up) →
  \hspace{1em} do
  \hspace{2em} x = l\$
  \hspace{2em} until flip & was_up then Up
  \hspace{2em} | flip then Down
  done
end
```

- Parallel composition of dataflow equations and automata
  - x has a different definition in each mode
  - But only a single definition in a reaction
Extended dataflow programming: automata

\[
\text{let node \( \text{counter} (\text{flip}, \text{stop}) = x \)
where
rec \( l_x = 0 \ fby \ x \)
and automaton

\begin{align*}
&| \text{Up} \rightarrow \\
&\quad \text{do} \\
&\quad \quad x = l_x + 1 \\
&\quad \quad \text{until} \ \text{flip then} \ Down \\
&\quad \quad \quad | \ \text{stop then} \ \text{Stop}(\text{true}) \\
&\quad \text{done}
\end{align*}

\begin{align*}
&| \text{Down} \rightarrow \\
&\quad \text{do} \\
&\quad \quad x = l_x - 1 \\
&\quad \quad \text{until} \ \text{flip then} \ \text{Up} \\
&\quad \quad \quad | \ \text{stop then} \ \text{Stop}(\text{false}) \\
&\quad \text{done}
\end{align*}

\begin{align*}
&| \text{Stop}(\text{was}_\text{up}) \rightarrow \\
&\quad \text{do} \\
&\quad \quad x = l_x \\
&\quad \quad \text{until} \ \text{flip \& was}_\text{up then} \ \text{Up} \\
&\quad \quad \quad | \ \text{flip then} \ \text{Down} \\
&\quad \text{done}
\end{align*}
\]

Parallel composition of dataflow equations and automata

- \( x \) has a different definition in each mode
- But only a single definition in a reaction
Extended dataflow programming: automata

```plaintext
let node counter (flip, stop) = x
where
  rec lx = 0 fby x
and automaton
  | Up →
    do
      x = lx + 1
    until flip then Down
    | stop then Stop(true)
    done
  | Down →
    do
      x = lx - 1
    until flip then Up
    | stop then Stop(false)
    done
  | Stop(was_up) →
    do
      x = lx
    until flip & was_up then Up
    | flip then Down
    done
end
```

- Parallel composition of dataflow equations and automata
- $x$ has a different definition in each mode
- But only a single definition in a reaction
Extended dataflow programming: automata

```
let node counter (flip, stop) = x
where
rec lx = 0 fby x
and automaton
  | Up ->
    do
      x = lx + 1
    until flip then Down
    | stop then Stop(true)
    done

  | Down ->
    do
      x = lx - 1
    until flip then Up
    | stop then Stop(false)
    done

  | Stop(was_up) ->
    do
      x = lx
    until flip & was_up then Up
    | flip then Down
    done
end
```

- Parallel composition of dataflow equations and automata
- \( x \) has a different definition in each mode
- But only a single definition in a reaction
Extended dataflow programming: automata

let node counter (flip, stop) = x
where
rec \(lx = 0\) fby x
and automaton
| Up →
  do
    x = lx + 1
  until flip then Down
  | stop then Stop(true)
  done

| Down →
  do
    x = lx - 1
  until flip then Up
  | stop then Stop(false)
  done

| Stop(was_up) →
  do
    x = lx
  until flip & was_up then Up
  | flip then Down
  done
end

- Parallel composition of dataflow equations and automata
- \(x\) has a different definition in each mode
- But only a single definition in a reaction
Extended dataflow programming: automata

```plaintext
let node counter (flip, stop) = x
where
rec lx = 0 fby x
and automaton
  | Up →
    do
      x = lx + 1
    until flip then Down
    | stop then Stop(true)
  done
    | Down →
      do
        x = lx - 1
      until flip then Up
        | stop then Stop(false)
    done
  | Stop(was_up) →
    do
      x = lx
    until flip & was_up then Up
      | flip then Down
    done
end
```

- Automata are just a convenient syntax
- They can be reduced to discrete dataflow equations by a

  source-to-source transformation
Extended dataflow programming: automata

Automata are just a convenient syntax
They can be reduced to discrete dataflow equations by a source-to-source transformation

let node counter (flip, stop) = x
where
rec lx = 0 fby x
and automaton
| Up →
  do
     x = lx + 1
  until flip then Down
  | stop then Stop(true)
done
| Down →
  do
     x = lx - 1
  until flip then Up
  | stop then Stop(false)
done
| Stop(was_up) →
  do
     x = lx
  until flip & was_up then Up
  | flip then Down
done
end

Extended dataflow programming: automata

```
let node counter (flip, stop) = x
where
rec lx = 0 fby x
and automaton
  | Up →
  |       do
  |       x = lx + 1
  |       until flip then Down
  |          | stop then Stop(true)
  |       done
  | Down →
  |       do
  |       x = lx - 1
  |       until flip then Up
  |          | stop then Stop(false)
  |       done
  | Stop(was_up) →
  |       do
  |       x = lx
  |       until flip & was_up then Up
  |          | flip then Down
  |       done
end
```

- Automata are just a convenient syntax
- They can be reduced to discrete dataflow equations by a source-to-source transformation

Extended dataflow programming: automata

```plaintext
let node counter (flip, stop) = x
where
rec lx = 0 fby x
and automaton
| Up →
  do
   x = lx + 1
   until flip then Down
   | stop then Stop(true)
   done

| Down →
  do
   x = lx - 1
   until flip then Up
   | stop then Stop(false)
   done

| Stop(was_up) →
  do
   x = lx
   until flip & was_up then Up
   | flip then Down
   done
end
```

- Automata are just a convenient syntax
- They can be reduced to discrete dataflow equations by a source-to-source transformation

---

Extended dataflow programming: automata

```plaintext
let node counter (flip, stop) = x
where
rec lx = 0 fby x
and automaton
  | Up →
    do
      x = lx + 1
    until flip then Down
    | stop then Stop(true)
    done

  | Down →
    do
      x = lx - 1
    until flip then Up
    | stop then Stop(false)
    done

  | Stop(was_up) →
    do
      x = lx
    until flip & was_up then Up
    | flip then Down
    done
end
```

- Automata are just a convenient syntax
- They can be reduced to discrete dataflow equations by a source-to-source transformation

![Diagram of automata with nodes and transitions]

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

Extended dataflow programming: automata

```plaintext
let counter = (flip, stop) = x

where

rec lx = 0 fby x

and automaton

| Up → |
  | do |
  | x = lx + 1 |
  | until flip then Down |
  | stop then Stop(true) |
  | done |

| Down → |
  | do |
  | x = lx - 1 |
  | until flip then Up |
  | stop then Stop(false) |
  | done |

| Stop(was_up) → |
  | do |
  | x = lx |
  | until flip & was_up then Up |
  | flip then Down |
  | done |
end
```

- Automata are just a convenient syntax
- They can be reduced to discrete dataflow equations by a source-to-source transformation

---

Extended dataflow programming: automata

```plaintext
let node counter (flip, stop) = x
where
rec lx = 0 fby x
and automaton
  | Up →
    do
      x = lx + 1
    until flip then Down
    | stop then Stop(true)
  done

  | Down →
    do
      x = lx - 1
    until flip then Up
    | stop then Stop(false)
  done

  | Stop(was_up) →
    do
      x = lx
    until flip & was_up then Up
    | flip then Down
  done
end
```

- Automata are just a convenient syntax
- They can be reduced to discrete dataflow equations by a source-to-source transformation

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

Extended dataflow programming: automata

```
let node counter (flip, stop) = x
where
rec lx = 0 fby x
and automaton
  | Up →
  |     do
  |       x = lx + 1
  |       until flip then Down
  |     | stop then Stop(true)
  | done

  | Down →
  |     do
  |       x = lx - 1
  |       until flip then Up
  |     | stop then Stop(false)
  | done

  | Stop(was_up) →
  |     do
  |       x = lx
  |       until flip & was_up then Up
  |     | flip then Down
  | done
end
```

- Automata are just a convenient syntax
- They can be reduced to discrete dataflow equations by a source-to-source transformation

![Diagram of automata]

Industrial version of Lustre/Lucid Synchrone with automata

Used in critical systems (DO-178B certified)

Airbus flight control; Train (interlocking, on-board); Nuclear safety
So, what’s left to do?

- We want a language for programming complex discrete systems and modelling their physical environments
- (Also: embedded software that includes physical models)
So, what’s left to do?

- We want a language for programming complex discrete systems and modelling their physical environments
- (Also: embedded software)
- Understand and improve the design of such modelling tools

Dassault Systèmes Delmia and Catia http://www.3ds.com/products
So, what’s left to do?

- We want a language for programming complex discrete systems and modelling their physical environments
  - (Also: embedded software that includes physical models)

- Something like Simulink/Stateflow, but
  - Simpler and more consistent semantics and compilation
  - Better understand interactions between discrete and continuous
  - Simpler treatment of automata
  - Certifiability for the discrete parts

Understand and improve the design of such modelling tools
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
Simulink/Stateflow

▶ Simulation → development
▶ two distinct simulation engines
▶ semantics & consistency: non-obvious
Lee and Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. EMSOFT ’07.

Our approach

▶ Source-to-source compilation
▶ Automata ⟷ data-flow
▶ Extend other languages (SCADE 6)
Approach

- Add Ordinary Differential Equations to an existing synchronous language

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

- Simulate with an external off-the-shelf numerical solver (Sundials CVODE, Hindmarsh et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. 2005.)

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

- Add Ordinary Differential Equations to an existing synchronous language

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

- Simulate with an external off-the-shelf numerical solver (Sundials CVODE, Hindmarsh et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. 2005.)

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

- Add Ordinary Differential Equations to an existing synchronous language

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

- Simulate with an external off-the-shelf numerical solver (Sundials CVODE, Hindmarsh et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. 2005.)

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

discrete synchronous language: assume infinitely fast execution
Discrete vs Hybrid time

discrete synchronous language: assume infinitely fast execution

ignore execution time

Q. How to relate discrete and continuous time correctly?
Discrete vs Hybrid time

discrete synchronous language: assume infinitely fast execution

hybrid synchronous language: assume an infinitesimal increment of the base clock—a non-standard semantics

Q. How to simulate effectively?
Discrete vs Hybrid time

discrete synchronous language: assume infinitely fast execution

\[ \mathbb{N} \]
Discrete vs Hybrid time

**Discrete synchronous language:** assume infinitely fast execution

**Hybrid synchronous language:** assume an infinitesimal increment of the base clock—a non-standard semantics

**Q.** How to simulate effectively?
Discrete vs Hybrid time

discrete synchronous language: assume infinitely fast execution

### How to relate discrete and continuous time correctly?
Discrete vs Hybrid time

discrete synchronous language: assume infinitely fast execution

time is a logical sequence of instantaneous reactions—no relation to physical time

hybrid synchronous language: assume infinitely precise base clock

⋆ assume an infinitesimal increment of the base clock—a non-standard semantics

Q. How to simulate effectively?
Discrete vs Hybrid time

discrete synchronous language: assume infinitely fast execution

time is a logical sequence of instantaneous reactions—no relation to physical time

hybrid synchronous language: assume infinitely precise base clock
Discrete vs Hybrid time

**discrete** synchronous language: assume infinitely fast execution

- time is a logical sequence of instantaneous reactions—no relation to physical time

**hybrid** synchronous language: assume infinitely precise base clock

- simulate effectively
Discrete vs Hybrid time

**Discrete synchronous language:** assume infinitely fast execution

- time is a logical sequence of instantaneous reactions—no relation to physical time

**Hybrid synchronous language:** assume infinitely precise base clock

- How to simulate effectively?
Discrete vs Hybrid time

**Discrete synchronous language:** assume infinitely fast execution

- time is a logical sequence of instantaneous reactions—no relation to physical time

**Hybrid synchronous language:** assume infinitely precise base clock

- How to relate discrete and continuous time correctly?
- How to simulate effectively?
Discrete vs Hybrid time

discrete synchronous language: assume infinitely fast execution

\[ \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \rightarrow \mathbb{N} \]

time is a logical sequence of instantaneous reactions—no relation to physical time

hybrid synchronous language: assume infinitely precise base clock

\[ \uparrow \rightarrow \mathbb{R} \]
Discrete vs Hybrid time

**discrete** synchronous language: assume infinitely fast execution

\[ \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \]

time is a logical sequence of instantaneous reactions—no relation to physical time

**hybrid** synchronous language: assume infinitely precise base clock

\[ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \]

\[ \mathbb{N} \quad \mathbb{R} \]

Q. How to relate discrete and continuous time correctly?

Q. How to simulate effectively?
Discrete vs Hybrid time

**Discrete synchronous language:** assume infinitely fast execution

time is a logical sequence of instantaneous reactions—no relation to physical time

**Hybrid synchronous language:** assume infinitely precise base clock

Q. How to relate discrete and continuous time correctly?

Q. How to simulate effectively?
Discrete vs Hybrid time

discrete synchronous language: assume infinitely fast execution

time is a logical sequence of instantaneous reactions—no relation to physical time

hybrid synchronous language: assume infinitely precise base clock

assume an infinitesimal increment of the base clock—a non-standard semantics
Discrete vs Hybrid time

discrete synchronous language: assume infinitely fast execution

Q. How to relate discrete and continuous time correctly?

hybrid synchronous language: assume infinitely precise base clock

Q. How to simulate effectively?
Which programs make sense?

Given:

```plaintext
let node sum(x) = cpt where
rec cpt = (0.0 fby cpt) +. x
```
Which programs make sense?

Given:

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

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

Evaluate:

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

and

\[
y = \text{sum(time)}
\]
Which programs make sense?

Given:

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

Evaluate:

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

Interpretation:

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

Given:

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

Evaluate:

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

Interpretation:

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

Given:

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

Evaluate:

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

Interpretation:

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

Given:

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

Evaluate:

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

Interpretation:

- Option 1: \( \mathbb{N} \subseteq \mathbb{R} \)
- Option 2: depends on solver
- Option 3: infinitesimal steps
- Option 4: type and reject
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
\[
y = \text{sum}(\text{time})
\]

Interpretation:

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

\[\xmark\]
Which programs make sense?

Given:

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

Evaluate:

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

Interpretation:

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

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

Evaluate:

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

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
Basic typing
Milner-like type system

The type language

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

Initial conditions

\[ \begin{align*}
(+) & : \text{int} \times \text{int} \xrightarrow{A} \text{int} \\
(=) & : \forall \beta . \beta \times \beta \xrightarrow{A} \text{bool} \\
\text{if} & : \forall \beta . \text{bool} \times \beta \times \beta \xrightarrow{A} \beta \\
\cdot \text{fby} \cdot & : \forall \beta . \beta \times \beta \xrightarrow{D} \beta \\
\text{up}(\cdot) & : \text{float} \xrightarrow{C} \text{zero}
\end{align*} \]
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...

“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 integrity — or correctness — of the results in a model that uses Stateflow charts, you must ensure that your model complies with 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 averaging 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 feature, a Stateflow chart can minimize side effects and compute outputs based on a constant 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.

Update local data only in transition, entry, and exit actions
To maintain precision in continuous-time simulation, you should update local data (continuous or discrete) only during physical events at major time steps.

In Stateflow charts, physical events cause state transitions. Therefore, write to local data only in actions that execute during transitions, as follows:

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

Consider the following chart.

```
A [k1] \rightarrow \rightarrow \rightarrow \rightarrow B [k2]
```

In this example, the action (k1) executes even when conditions c1 and c2 are false. In this case, c1 gets updated in a minor time step because there is no state transition.

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

Do not call Simulink functions in state during actions or transition conditions
These rules apply to continuous-time charts because you cannot call Simulink 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 condition, an error message appears when you simulate your model.

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

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

Do not read outputs and derivatives in states or transitions
This restriction ensures smooth outputs in a major time step because it prevents a Stateflow chart from using values that may no longer be valid in the current minor time step. Instead, a Stateflow chart always computes derivatives 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.

➤ ‘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
Compilation: source-to-source transformation

let hybrid ball () =
    let
        rec der v = (−. g / m) init v0
            reset (−. 0.8 *. last v) every up(−. h)
        and der h = v init h0
    in (v, h)

let node ball (z1, (lh, lv), ()) =
    let rec i = true fby false

    and dv = (−. g / m)
    and v = if i then v0
        else if z1 then −. 0.8 *. lv
        else lv

    and dh = v
    and h = if i then h0 else lh

    and upz1 = −. h

    in ((v, h), upz1, (h, v), (dh, dv))
let hybrid ball () =
  let
  rec der v = (−. g / m) init v0
  reset (−. 0.8 * . last v) every up(−. h)
  and der h = v init h0
  in (v, h)

let node ball (z1, (lh, lv), ()) =
  let rec i = true fby false
  and dv = (−. g / m)
  and v = if i then v0
  else if z1 then −. 0.8 * . lv
  else lv
  and dh = v
  and h = if i then h0 else lh
  and upz1 = −. h
  in ((v, h), upz1, (h, v), (dh, dv))
Compilation: source-to-source transformation

```ocaml
let hybrid ball () =
  let rec der v = (−. g / m) init v0
      reset (−. 0.8 *. last v) every up(−. h)
  and der h = v init h0
  in (v, h)

let node ball (z1, (lh, lv), ()) =
  let rec i = true fby false
  and dv = (−. g / m)
  and v = if i then v0
    else if z1 then −. 0.8 *. lv
    else lv
  and dh = v
  and h = if i then h0 else lh
  and upz1 = −. h
```

Careful mixing of discrete and continuous ▶ control discrete changes to respect invariant ▶ branching (i.e. automata) is tricky
let hybrid ball () =
  let
  rec der v = (−. g / m) init v0
  reset (−. 0.8 * . last v) every up(−. h)
  and der h = v init h0
  in (v, h)

let node ball (z1, (lh, lv), ()) =
  let rec i = true fby false
  and dv = (−. g / m)
  and v = if i then v0
  else if z1 then −. 0.8 * . lv
  else lv
  and dh = v
  and h = if i then h0 else lh
  and upz1 = −. h
  in ((v, h), upz1, (h, v), (dh, dv))
Compilation: source-to-source transformation

```ocaml
let hybrid ball () =
  let rec der v = (-. g / m) init v0
  reset (-. 0.8 *. last v) every up(-. h)
  and der h = v init h0
  in (v, h)

let node ball (z1, (lh , lv), ()) =
  let rec i = true fby false
  and dv = (-. g / m)
  and v = if i then v0
  else if z1 then -. 0.8 *. lv
  else lv
  and dh = v
  and h = if i then h0 else lh
  and upz1 = -. h
  in ((v, h), upz1, (h, v), (dh, dv))
```
Compilation: source-to-source transformation

```latex
let hybrid ball () =
  let rec der v = (−. g / m) init v0
  reset (−. 0.8 * last v) every up(−. h)
  and der h = v init h0
  in (v, h)

let node ball (z1, (lh, lv), ()) =
  let rec i = true fby false
  and dv = (−. g / m)
  and v = if i then v0
  else if z1 then −. 0.8 * lv
  else lv
  and dh = v
  and h = if i then h0 else lh
  and upz1 = −. h
  in ((v, h), upz1, (h, v), (dh, dv))
```
Demonstrations

- Bouncing ball (standard)
- Bang-bang temperature controller (Simulink/Stateflow)
- Sticky Masses (Ptolemy)
Conclusion

- 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
- Our approach
  1. Synchronous data-flow language with automata and ODEs
  2. Static type system to separate discrete from continuous behaviors
  3. Relate discrete to continuous via zero-crossings
  4. Compilation via source-to-source transformations
  5. Simulation using off-the-shelf numerical solvers
- Prototype compiler in OCaml using Sundials CVODE solver
Conclusion

- 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
- Our approach
  1. Synchronous data-flow language with automata and ODEs
  2. Static type system to separate discrete from continuous behaviors
  3. Relate discrete to continuous via zero-crossings
  4. Compilation via source-to-source transformations
  5. Simulation using off-the-shelf numerical solvers
- Prototype compiler in OCaml using Sundials CVODE solver
Conclusion

- 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
- Our approach
  1. Synchronous data-flow language with automata and ODEs
  2. Static type system to separate discrete from continuous behaviors
  3. Relate discrete to continuous via zero-crossings
  4. Compilation via source-to-source transformations
  5. Simulation using off-the-shelf numerical solvers
- Prototype compiler in OCaml using Sundials CVODE solver
Conclusion

- 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
- Our approach
  1. Synchronous data-flow language with automata and ODEs
  2. Static type system to separate discrete from continuous behaviors
  3. Relate discrete to continuous via zero-crossings
  4. Compilation via source-to-source transformations
  5. Simulation using off-the-shelf numerical solvers
- Prototype compiler in OCaml using Sundials CVODE solver
Future work

Short term

- Compiler implementation
  - Language
  - Examples / case studies: more!

Bibliography at: www.di.ens.fr/pouzet
Future work

Short term

- Compiler implementation
- Language
- Examples / case studies: more!

Bibliography at: www.di.ens.fr/pouzet
Future work

Short term

- Compiler implementation
- Language
- Examples / case studies: more!
Future work

Short term

- Compiler implementation
- Language
- Examples / case studies: more!

Longer term

- Clock calculus: an finer analysis of piece-wise continuous/continuous/discrete
- Causality analysis: (partial) detection of discrete Zeno-behavior
- Semantics: how abstract is the solver?
- Real-time simulation (trade-off accuracy and execution time)

Bibliography at: www.di.ens.fr/ pouzet