Extension of Synchronous Data-flow Languages: main achievements, new perspectives

Marc Pouzet

DI, École normale supérieure
Marc.Pouzet@ens.fr

Seminar MaMux, IRCAM, February 3, 2012

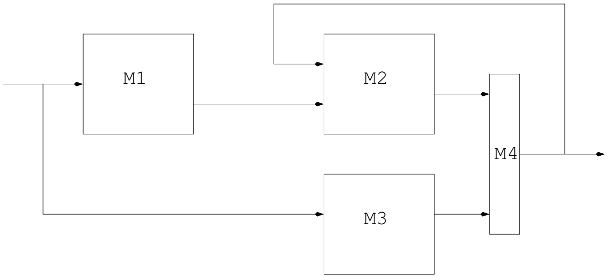
Reactive systems

- 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
  $\hookrightarrow$ parallel description of the system
- Strong temporal constraints.
  e.g, the physics does not wait!
  $\hookrightarrow$ temporal constraints should be expressed in the system
- Safety is important (critical systems).
  $\hookrightarrow$ well founded languages, verification methods

Synchronous Kahn Networks

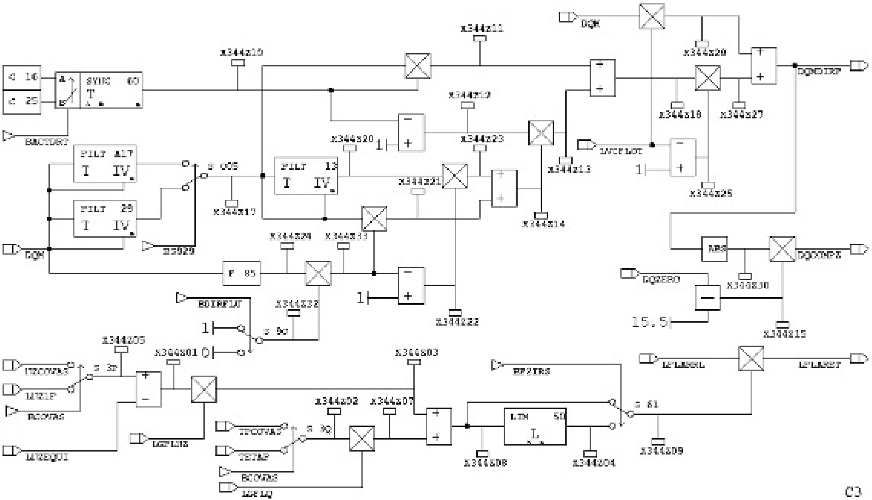


- parallel processes communicating through data-flows
- communication in zero time: data is available as soon as it is produced.
- a global logical time scale even though individual rhythms may differ

these drawings are computer programs

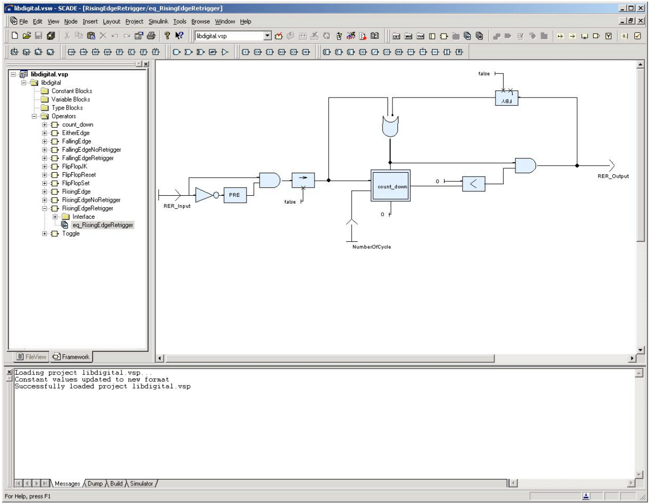
SAO (Spécification Assistée par Ordinateur)—Airbus 80's

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

Caspi, Pilaud, Halbwachs, and Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. 1987.

Programming with streams

Lustre: a dataflow programming language

Caspi, Pilaud, Halbwachs, and Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. 1987.

Programming with streams

constants $\quad 1 \quad = \quad 1 \quad 1 \quad 1 \quad 1 \quad \cdots$

Lustre: a dataflow programming language

Caspi, Pilaud, Halbwachs, and Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. 1987.

Programming with streams

constants	1	$=$	1	1	1	1	$\cdots$
operators	x + y	$=$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$\cdots$

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

Lustre: a dataflow programming language

Caspi, Pilaud, Halbwachs, and Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. 1987.

Programming with streams

constants	1	$=$	1	1	1	1	$\cdots$
operators	x + y	$=$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$\cdots$
unit delay	0 **fby** (x + y)	$=$	0	$x_0 + y_0$	$x_1 + x_1$	$x_2 + x_2$	$\cdots$

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

Lustre: a dataflow programming language

Caspi, Pilaud, Halbwachs, and Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. 1987.

Programming with streams

constants	1	$=$	1	1	1	1	$\cdots$
operators	x + y	$=$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$\cdots$

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

unit delay	0 **fby** (x + y)	$=$	0	$x_0 + y_0$	$x_1 + x_1$	$x_2 + x_2$	$\cdots$
	pre (x + y)	$=$	*nil*	$x_0 + y_0$	$x_1 + x_1$	$x_2 + x_2$	$\cdots$

Lustre: a dataflow programming language

Caspi, Pilaud, Halbwachs, and Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. 1987.

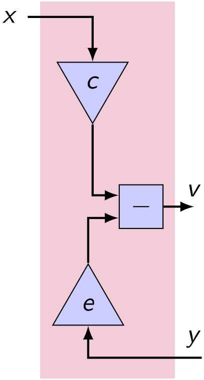
Programming with streams

constants	1	$=$	1	1	1	1	$\cdots$
operators	x + y	$=$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$\cdots$

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

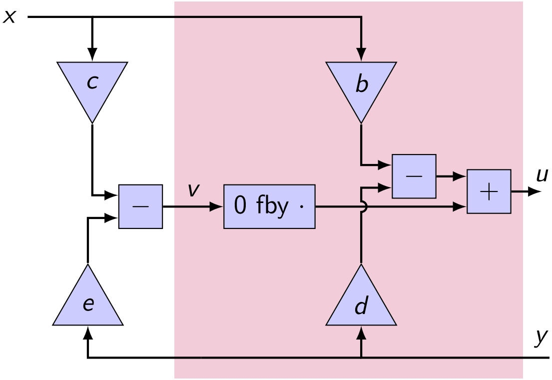
unit delay	0 **fby** (x + y)	$=$	0	$x_0 + y_0$	$x_1 + x_1$	$x_2 + x_2$	$\cdots$
	pre (x + y)	$=$	*nil*	$x_0 + y_0$	$x_1 + x_1$	$x_2 + x_2$	$\cdots$
	0 $\rightarrow$ **pre** (x + y)	$=$	0	$x_0 + y_0$	$x_1 + x_1$	$x_2 + x_2$	$\cdots$

Lustre: a dataflow programming language



`v = c * x - e * y`

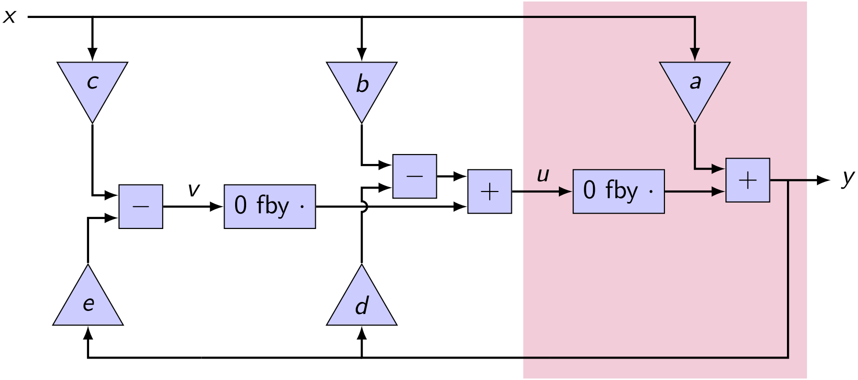
Lustre: a dataflow programming language



u = b * x − d * y + (0.0 **fby** v)

and v = c * x − e * y

Lustre: a dataflow programming language

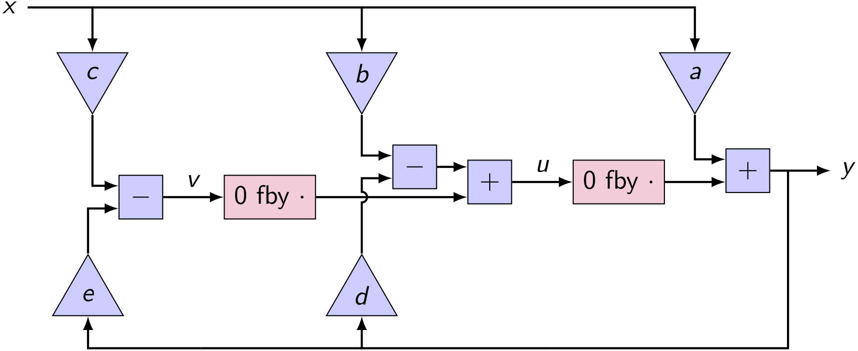


rec y = a ∗ x + (0.0 **fby** u)

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

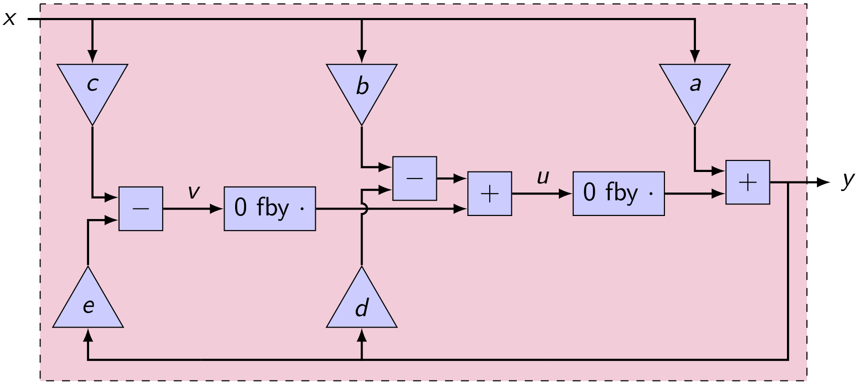
and v = c ∗ x − e ∗ y

Lustre: a dataflow programming language



$$
\begin{aligned}
&\textbf{rec}\ \ \mathrm{y} = \mathrm{a} * \mathrm{x} + (0.0\ \ \textbf{fby}\ \ \mathrm{u}) \\
&\textbf{and}\ \ \mathrm{u} = \mathrm{b} * \mathrm{x} - \mathrm{d} * \mathrm{y} + (0.0\ \ \textbf{fby}\ \ \mathrm{v}) \\
&\textbf{and}\ \ \mathrm{v} = \mathrm{c} * \mathrm{x} - \mathrm{e} * \mathrm{y}
\end{aligned}
$$

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
  - 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 and need to be extended in several ways. . .

The expressiveness of Lustre

- First order functional language managing streams, no recursion.
- Types are declared; no polymorphism; no control-structures; limited clock calculus.

Increase its expressiveness:

- Modularity (libraries), abstraction mechanisms.
- Polymorphism; type and clock inference.
- Control structures; imperative features (but in a safe way).
- More efficient compilation; compile-time static analysis.

Lucid Synchrone

[ICFP'96]
Try to mix all the best of these two paradigms:

- Synchronous data-flow as a way to deal with time.
- Features from ML to increase expressiveness: E.g., type inference, polymorphism, higher-order, (some form of) recursion.

Follow some principles:

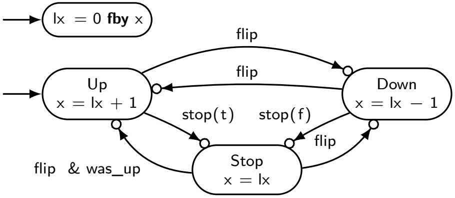
- Streams and function composition.
- The synchronous property is checked by a dedicated type system called the clock calculus. Inferred clocks express static constraints on synchronization.
- Clocks are used to give a precise semantics to all programming constructs.
- Several other type-based analysis (e.g., initialisation, causality).

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
```



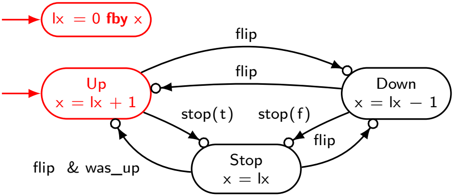
- Parallel composition of dataflow equations and automata
- x has a different definition in each mode
- But only a single definition in a reaction

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
```



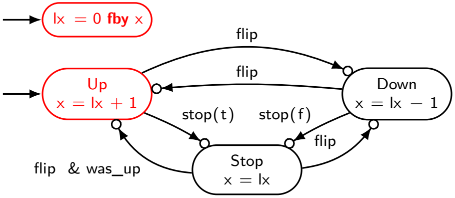
- Parallel composition of dataflow equations and automata
- x has a different definition in each mode
- But only a single definition in a reaction

Extended dataflow programming: automata

```
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
```



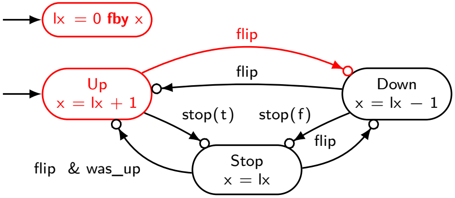
- Parallel composition of dataflow equations and automata
- x has a different definition in each mode
- But only a single definition in a reaction

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
```



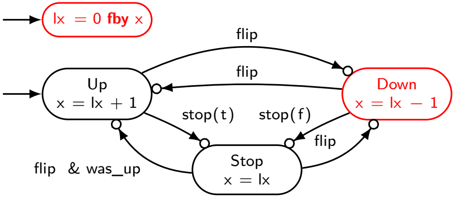
- Parallel composition of dataflow equations and automata
- x has a different definition in each mode
- But only a single definition in a reaction

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
```



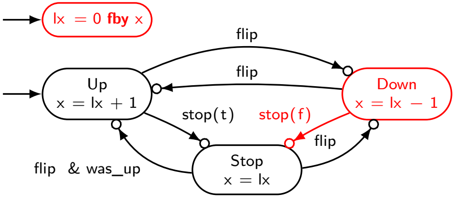
- Parallel composition of dataflow equations and automata
- x has a different definition in each mode
- But only a single definition in a reaction

Extended dataflow programming: automata

```
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
```



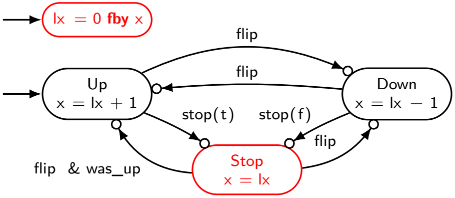
- Parallel composition of dataflow equations and automata
- x has a different definition in each mode
- But only a single definition in a reaction

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
```



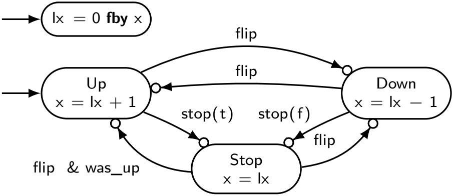
- Parallel composition of dataflow equations and automata
- x has a different definition in each mode
- But only a single definition in a reaction

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
```



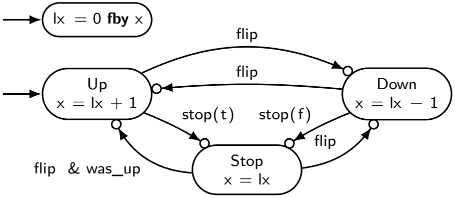
- Automata are just a convenient syntax
- They can be reduced to discrete dataflow equations by a source-to-source transformation

Extended dataflow programming: automata

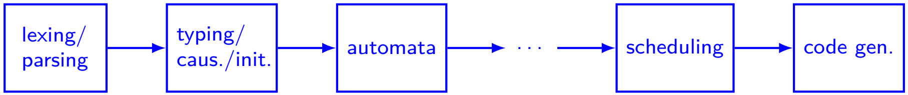
```
let node counter (flip , stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
        do
          x = lx + 1
        until flip then Down
            | stop then Stop(true)
        done

     | 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



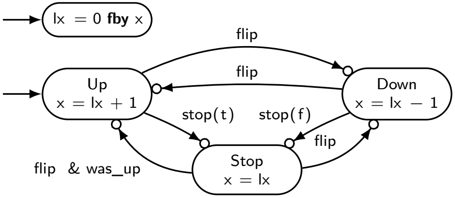
olaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. EMSOFT'2005.

Extended dataflow programming: automata

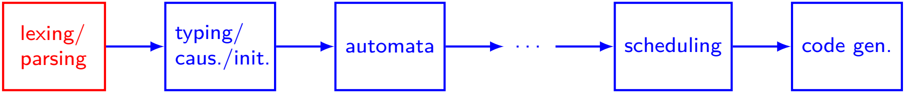
```
let node counter (flip , stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | 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



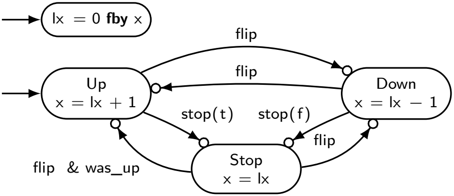
olaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. EMSOFT'2005.

Extended dataflow programming: automata

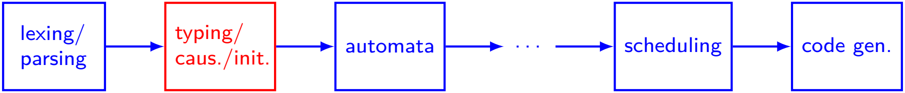
```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | 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



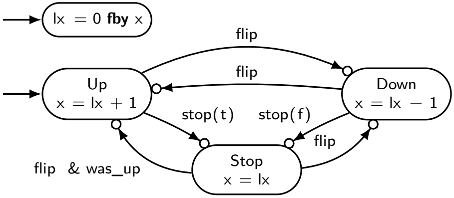
olaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. EMSOFT'2005.

Extended dataflow programming: automata

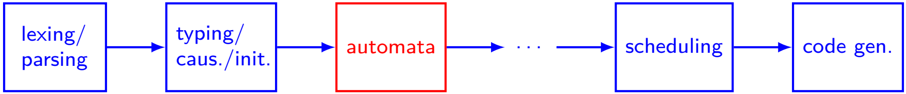
```
let node counter (flip , stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | 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



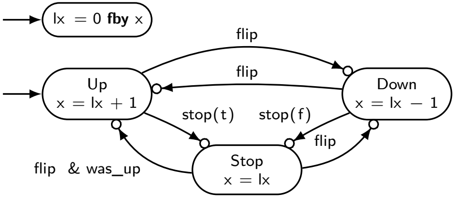
olaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. EMSOFT'2005.

Extended dataflow programming: automata

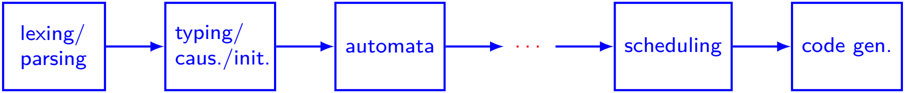
```
let node counter (flip , stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | 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



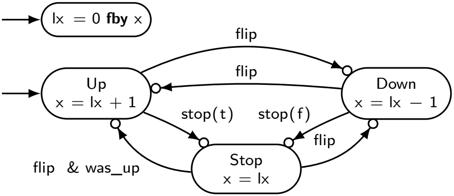
olaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. EMSOFT'2005.

Extended dataflow programming: automata

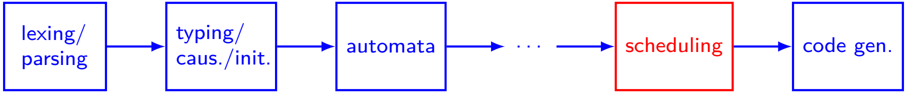
```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | 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



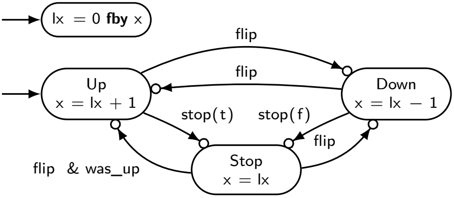
olaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. EMSOFT'2005.

Extended dataflow programming: automata

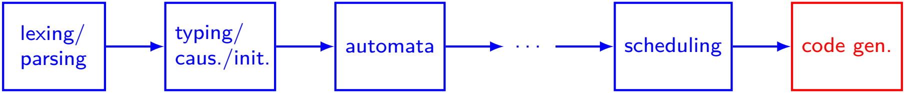
```
let node counter (flip, stop) = x
 where
 rec lx = 0 fby x
 and automaton
     | Up →
       do
         x = lx + 1
       until flip then Down
           | stop then Stop(true)
       done

     | 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
```

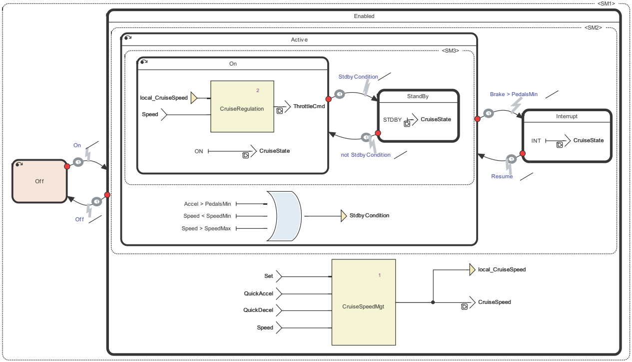


- Automata are just a convenient syntax
- They can be reduced to discrete dataflow equations by a source-to-source transformation



olaço, Pagano and Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. EMSOFT'2005.

SCADE 6



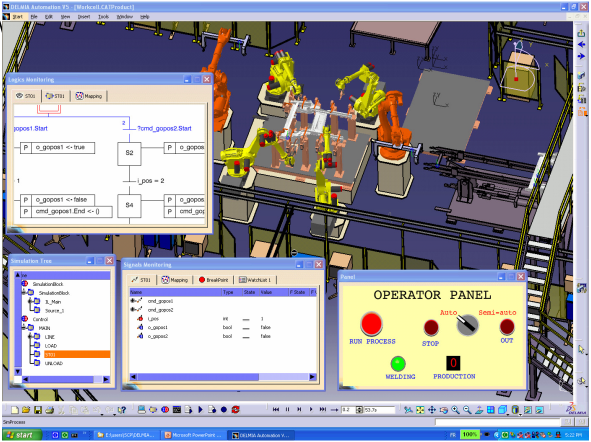
- Entirely new language and compiler; commercialised since 2008.
- Most features of Lucid Synchrone are included
- Used in critical systems (DO-178B certified)
- Airbus flight control; Train (interlocking, on-board); Nuclear safety

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 softw



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. Operational semantics of hybrid systems. HSCC 2005.
Lee and Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. EMSOFT'07.

Lee and Zheng. Operational semantics of hybrid systems. HSCC 2005.
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. Operational semantics of hybrid systems. HSCC 2005.
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 $\rightsquigarrow$ development
- two distinct simulation engines
- semantics & consistency: non-obvious

Lee and Zheng. Operational semantics of hybrid systems. HSCC 2005.
Lee and Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. EMSOFT'07.

Our approach

- Source-to-source compilation
- Automata $\leadsto$ 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: [CDC'11,JCSS'11]

discrete synchronous language: assume infinitely fast execution

$$\longrightarrow \mathbb{N}$$

Discrete vs Hybrid time: [CDC'11,JCSS'11]

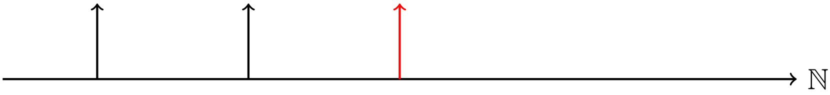
discrete synchronous language: assume infinitely fast execution

Discrete vs Hybrid time: [CDC'11,JCSS'11]

discrete synchronous language: assume infinitely fast execution

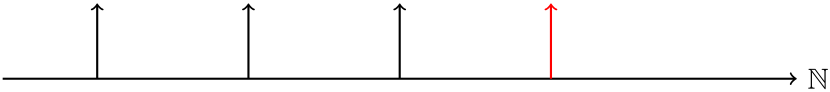
Discrete vs Hybrid time: [CDC'11,JCSS'11]

discrete synchronous language: assume infinitely fast execution



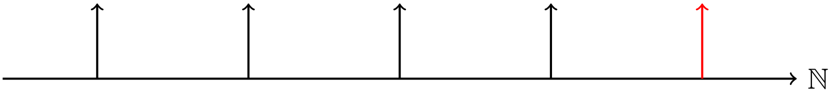
Discrete vs Hybrid time: [CDC'11,JCSS'11]

discrete synchronous language: assume infinitely fast execution



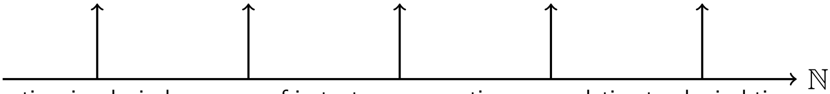
Discrete vs Hybrid time: [CDC'11,JCSS'11]

discrete synchronous language: assume infinitely fast execution



Discrete vs Hybrid time: [CDC'11,JCSS'11]

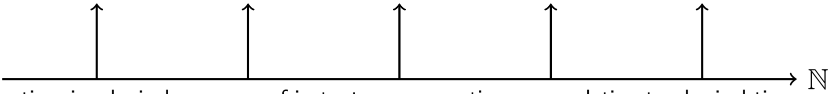
discrete synchronous language: assume infinitely fast execution



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

Discrete vs Hybrid time: [CDC'11,JCSS'11]

discrete synchronous language: assume infinitely fast execution

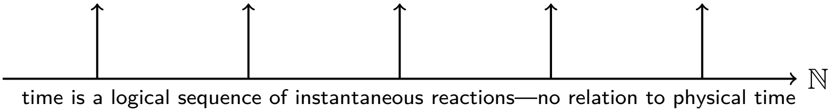


time is a logical sequence of instantaneous reactions—no relation to physical time

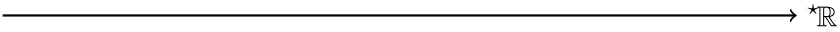
hybrid synchronous language: assume infinitely precise base clock

Discrete vs Hybrid time: [CDC'11,JCSS'11]

discrete synchronous language: assume infinitely fast execution

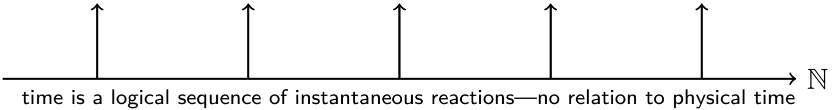


hybrid synchronous language: assume infinitely precise base clock

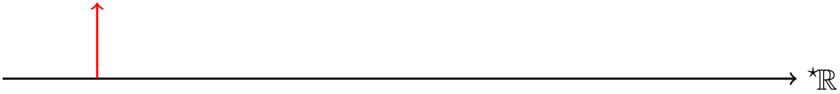


Discrete vs Hybrid time: [CDC'11,JCSS'11]

discrete synchronous language: assume infinitely fast execution

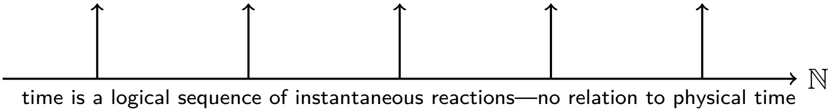


hybrid synchronous language: assume infinitely precise base clock

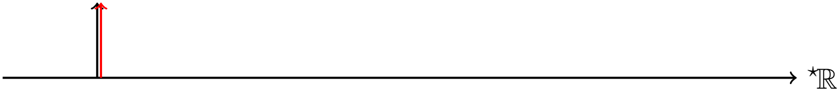


Discrete vs Hybrid time: [CDC'11,JCSS'11]

discrete synchronous language: assume infinitely fast execution

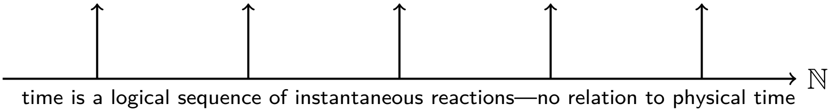


hybrid synchronous language: assume infinitely precise base clock



Discrete vs Hybrid time: [CDC'11,JCSS'11]

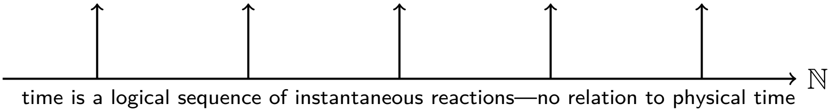
discrete synchronous language: assume infinitely fast execution



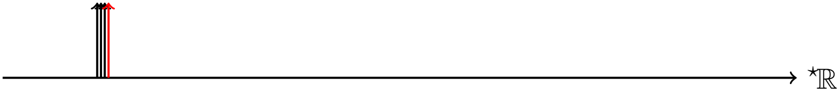
hybrid synchronous language: assume infinitely precise base clock

Discrete vs Hybrid time: [CDC'11,JCSS'11]

discrete synchronous language: assume infinitely fast execution

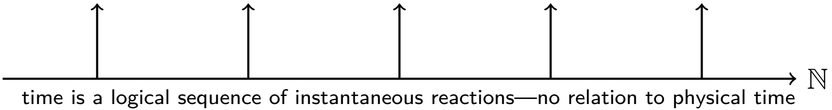


hybrid synchronous language: assume infinitely precise base clock



Discrete vs Hybrid time: [CDC'11,JCSS'11]

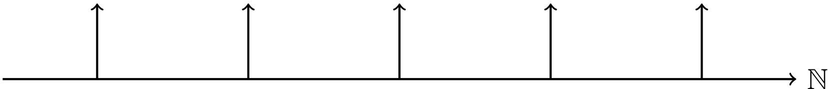
discrete synchronous language: assume infinitely fast execution



hybrid synchronous language: assume infinitely precise base clock

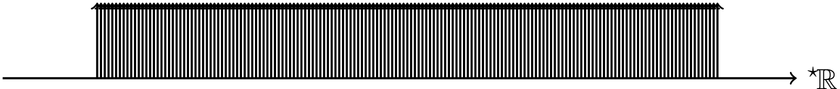
Discrete vs Hybrid time: [CDC'11,JCSS'11]

discrete synchronous language: assume infinitely fast execution



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

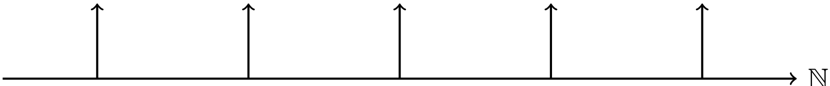
hybrid synchronous language: assume infinitely precise base clock



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

Discrete vs Hybrid time: [CDC'11,JCSS'11]

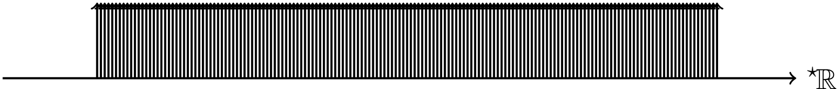
discrete synchronous language: assume infinitely fast execution



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

Q. How to relate discrete and continuous time correctly?

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?

Which programs make sense?

Given:

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

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)
```

Which programs make sense?

Given:

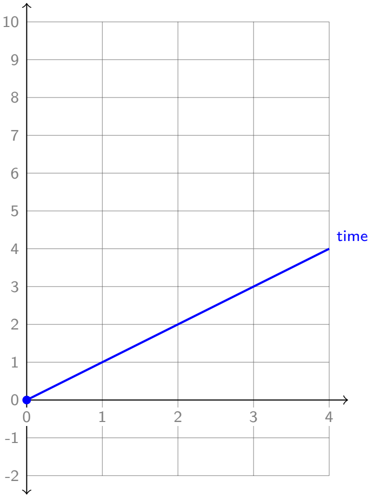
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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:

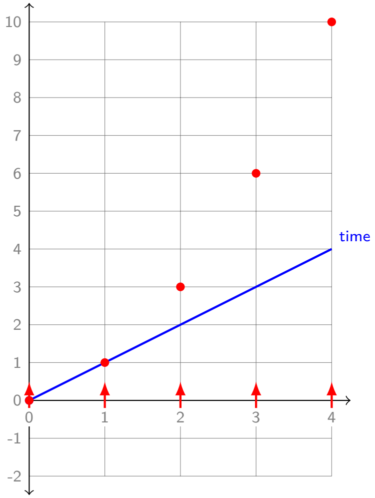
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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



Which programs make sense?

Given:

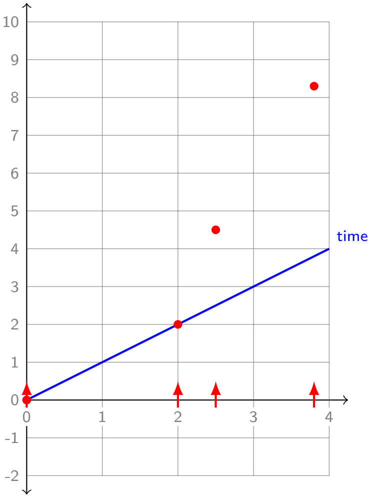
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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:

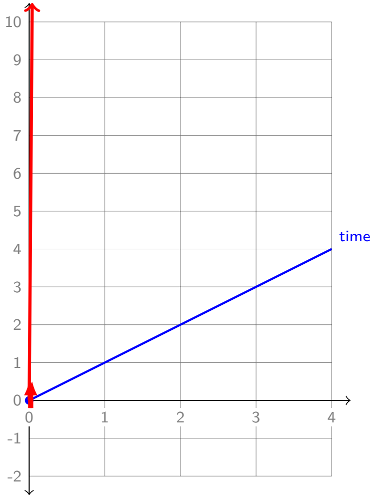
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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



Which programs make sense?

Given:

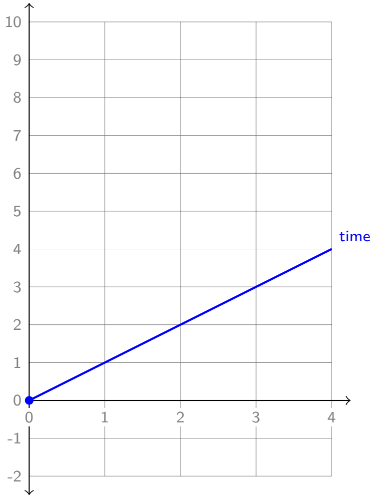
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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:

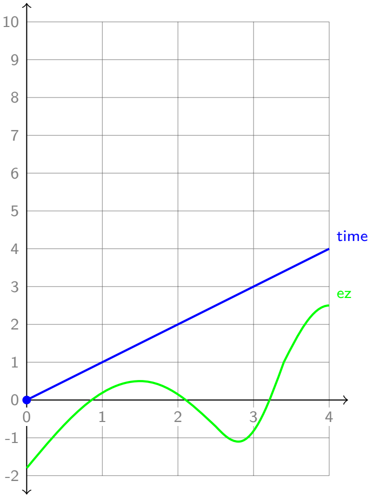
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

Evaluate:

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



Which programs make sense?

Given:

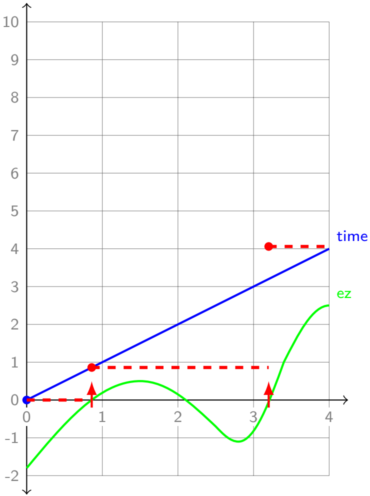
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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)

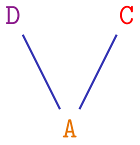
Try to minimize the effects of solver parameters and choices

Basic typing

Milner-like type system [LCTES'11,EMSOFT'11]

The type language

$$
\begin{array}{lcl}
bt & ::= & \texttt{float} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{zero} \\
t & ::= & bt \mid t \times t \mid \beta \\
\sigma & ::= & \forall \beta_1, ..., \beta_n . t \xrightarrow{k} t \\
k & ::= & \texttt{D} \mid \texttt{C} \mid \texttt{A}
\end{array}
$$



Initial conditions

$$
\begin{array}{lcl}
(+) & : & \texttt{int} \times \texttt{int} \xrightarrow{\texttt{A}} \texttt{int} \\
(=) & : & \forall \beta . \beta \times \beta \xrightarrow{\texttt{A}} \texttt{bool} \\
\texttt{if} & : & \forall \beta . \texttt{bool} \times \beta \times \beta \xrightarrow{\texttt{A}} \beta \\
\cdot \, \texttt{fby} \, \cdot & : & \forall \beta . \beta \times \beta \xrightarrow{\texttt{D}} \beta \\
\texttt{up}(\cdot) & : & \texttt{float} \xrightarrow{\texttt{C}} \texttt{zero}
\end{array}
$$

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 *smoothness* — of the results in continuous-time modeling, you must constrain your charts to a restricted subset of Stateflow chart semantics. The restricted semantics ensure that inputs do not depend on unpredictable factors — or *side effects* — such as:

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

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

A Stateflow chart generates informative errors to help you correct semantic violations.

Summary of Rules for Continuous-Time Modeling

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

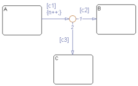
Update local data *only* in transition, `entry`, and `exit` actions

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

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

- 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 this case, `n` gets updated in a minor time step because there is no state transition.

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

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

This rule applies to continuous-time charts because you cannot call functions during minor time steps. You can call Simulink functions in state `entry` or `exit` actions and transition actions. However, if you try to call Simulink

functions in state `during` actions or transition conditions, an error message appears when you simulate your model.

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

Compute derivatives only in `during` actions

A Simulink model reads 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 outputs from local discrete data, local continuous data, and chart inputs.

Use discrete variables to govern conditions in `during` actions

This restriction prevents mode changes from occurring between major time steps. When placed in `during` actions, conditions that affect control flow should be governed by discrete variables because they do not change between major time steps.

Do not use input events in continuous-time charts

The presence of input events makes a chart behave like a triggered subsystem and therefore unable to simulate in continuous-time. For example, the following model generates an error if the chart uses a continuous update method.

...

- '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

Demonstrations

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

Conclusion

Lucid Synchrone: `www.di.ens.fr:/~pouzet/lucid-synchrone`

- A laboratory language for experimenting extensions for SCADE.
- type/clock inference, initialization analysis, causality analysis.
- Higher-order functions.
- Hierarchical automata, signals.
- Modular code generation.

Most are included in SCADE 6.

A promising perspective for synchronous languages is the treatment of mixed (discrete/continuous) signals.

- A synchronous language extended with ODEs and/or DAEs.
- Recycle the semantics, language constructs and compilation techniques.
- A prototype is under way.

Milestones

- Synchronous Kahn networks [ICFP'96]
- Clocks as dependent types [ICFP'96]
- Modular compilation (co-induction *vs* co-iteration) [CMCS'98]
- ML-like clock calculus [Emsoft'03]
- causality analysis [ESOP'01]
- initialization analysis [SLAP'03, STTT'04]
- higher-order and typing [Emsoft'04]
- data-flow and state machines [Emsoft'05, Emsoft'06]
- N-Synchronous Kahn Networks [Emsoft'05, POPL'06, APLAS'08, MPC'10]
- Clock-directed code generation of synchronous data-flow [LCTES'08]
- Modular Static Scheduling [Emsoft'09, JDAES'10]
- Synchronous semantics based on non-standard for hybrid systems [CDC'10,JCSS'11]
- A Lustre-like language with ODEs [LCTES'11,Emsoft'11]