A Hybrid Synchronous Language with Hierarchical Automata

Static Typing and Translation to Synchronous Code

Albert Benveniste[1] Benoît Caillaud[1]
Timothy Bourke[1,2] Marc Pouzet[2,1]

1. INRIA

2. ENS, Paris

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

Aim

Programming languages perspective:

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

Better understand the combination of discrete and continuous components

The usual questions (and techniques):

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

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

Aim

Programming languages perspective:

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

Better understand the combination of discrete and continuous components

The usual questions (and techniques):

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

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

Aim

Programming languages perspective:

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

Better understand the combination of discrete and continuous components

The usual questions (and techniques):

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

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

Aim

Programming languages perspective:

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

Better understand the combination of discrete and continuous components

The usual questions (and techniques):

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

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

Approach

- Add Ordinary Differential Equations to an existing synchronous language
- Two concrete reasons:
  - Increase modeling power (hybrid programming)
  - Exploit existing compiler (target for code generation)
- Simulate with an external off-the-shelf numerical solver (Sundials CVODE, Hindmarsh et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Mathematical Software*, 31(3):363–396, 2005.)
- Conservative extension: synchronous functions are compiled, optimized, and executed as per usual.
- Extends previous work: add hierarchical automata to LCTES 2011

Understand (continuous) automata and their parallel composition from a synchronous language viewpoint

(causality relations, activations (clocks), semantics)

Approach

- Add Ordinary Differential Equations to an existing synchronous language
- Two concrete reasons:
  - Increase modeling power (hybrid programming)
  - Exploit existing compiler (target for code generation)
- Simulate with an external off-the-shelf numerical solver (Sundials CVODE, Hindmarsh et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Mathematical Software*, 31(3):363–396, 2005.)
- Conservative extension: synchronous functions are compiled, optimized, and executed as per usual.
- Extends previous work: add hierarchical automata to LCTES 2011

Understand (continuous) automata and their parallel composition from a synchronous language viewpoint

(causality relations, activations (clocks), semantics)

Approach

- Add Ordinary Differential Equations to an existing synchronous language
- Two concrete reasons:
  - Increase modeling power (hybrid programming)
  - Exploit existing compiler (target for code generation)
- Simulate with an external off-the-shelf numerical solver (Sundials CVODE, Hindmarsh et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Mathematical Software*, 31(3):363–396, 2005.)
- Conservative extension: synchronous functions are compiled, optimized, and executed as per usual.
- Extends previous work: add hierarchical automata to LCTES 2011

Understand (continuous) automata and their parallel composition from a synchronous language viewpoint

(causality relations, activations (clocks), semantics)

Approach

- Add Ordinary Differential Equations to an existing synchronous language
- Two concrete reasons:
  - Increase modeling power (hybrid programming)
  - Exploit existing compiler (target for code generation)
- Simulate with an external off-the-shelf numerical solver (Sundials CVODE, Hindmarsh et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Mathematical Software*, 31(3):363–396, 2005.)
- 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. 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 $\rightsquigarrow$ data-flow
- Extend other languages (SCADE 6)

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:

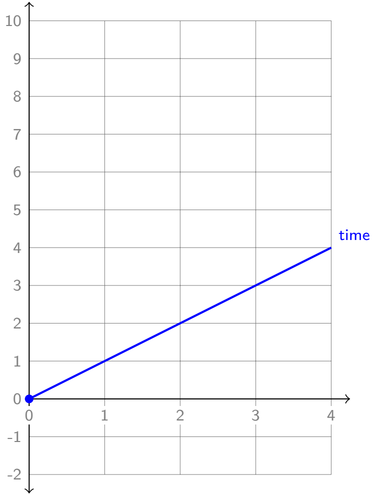
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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:

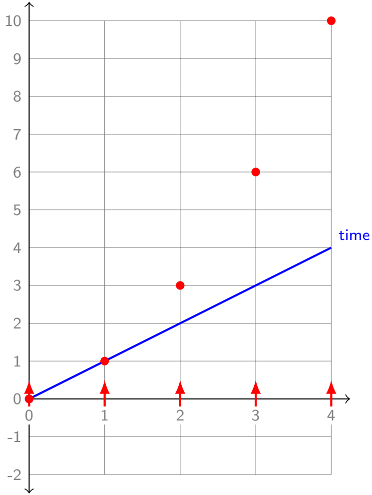
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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:

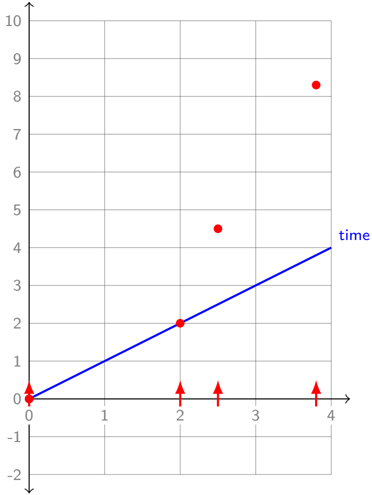
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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:

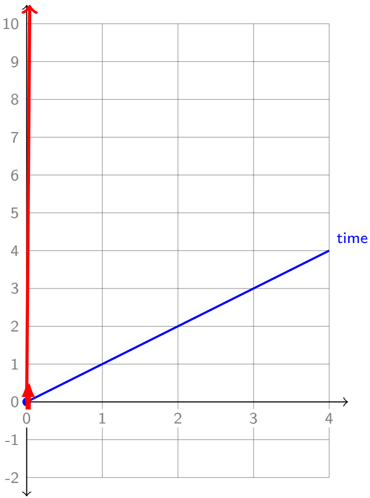
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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:

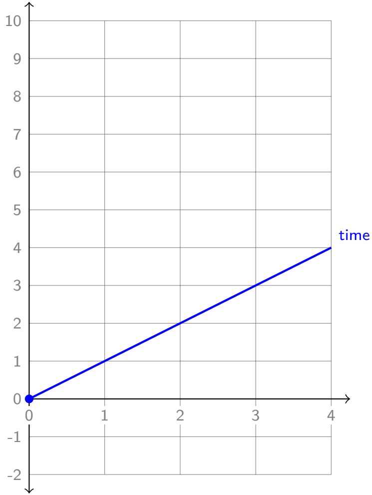
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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:

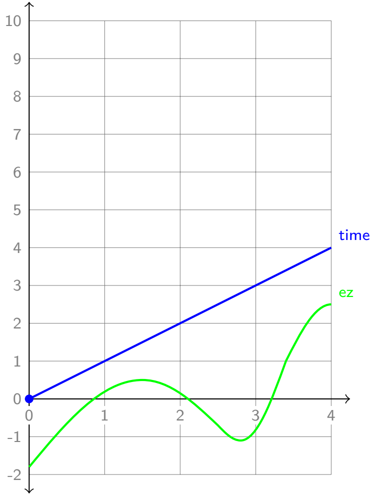
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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:

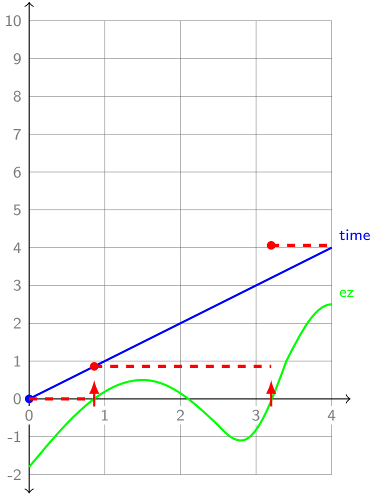
```
let node sum(x) = cpt where
  rec cpt = (0.0 fby cpt) +. x
```

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

Typing
Motivation

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

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

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

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

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

Typing
Motivation

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

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

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

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

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

Typing

Motivation

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

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

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

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

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

Typing

Motivation

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

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

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

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

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

Typing

Unreasonable programs

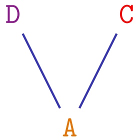
```
der y = 1.0 init 0.0        and    x = (0.0 → pre x) + y

x = 0.0 → (pre x +. 1.0)    and    der y = x init 0.0
```

- y is a variable that changes *continuously*
- x is *discrete* register
- The relationship between the two is ill-defined

Typing

The type language

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



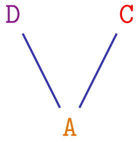
Initial conditions

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

Typing

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

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

Typing

$$G, H \vdash_{\mathtt{C}} \ \textbf{der}\ \mathrm{y} \ = \ 1.0 \ \textbf{init} \ 0.0 \qquad G, H \vdash_{\mathtt{D}} \ x \ = \ (0.0 \ \textbf{fby} \ (\mathrm{x} + 1))$$

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

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

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

$$j : (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}}) \longrightarrow (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}})$$

Typing

$G, H \vdash_{\mathtt{C}}$ **der** y $=$ 1.0 **init** 0.0 $\qquad G, H \vdash_{\mathtt{D}} x = $ (0.0 **fby** (x + 1))

$G, H \vdash_{?}$ **der** y $= \cdots$ **and** $x = \cdots$

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

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

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

$$j : (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}}) \longrightarrow (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}})$$

Typing

$G, H \vdash_{\mathtt{C}}$ **der** y $=$ 1.0 **init** 0.0 $\qquad G, H \vdash_{\mathtt{D}} x \ = \ $ (0.0 **fby** (x + 1))

$G, H \vdash_{?}$ **der** y $= \cdots$ **and** $x = \cdots$ ✗

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

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

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

$$j : (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}}) \longrightarrow (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}})$$

Typing

$G, H \vdash_{\mathtt{C}}$ **der** y $=$ 1.0 **init** 0.0

$G, H \vdash_{\mathtt{D}} x = (0.0$ **fby** $(x + 1))$

$G, H \vdash_{\mathtt{C}} x' = (0.0$ **fby** $(x + 1))$ **every** up(ez) **init** 0.0

$G, H \vdash_{?}$ **der** y $= \cdots$ **and** $x = \cdots$ ✗

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

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

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

$$j : (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}}) \longrightarrow (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}})$$

Typing

$G, H \vdash_{\mathtt{C}}$ **der** y $=$ 1.0 **init** 0.0

$G, H \vdash_{\mathtt{D}}$ x $=$ (0.0 **fby** (x + 1))

$G, H \vdash_{\mathtt{C}}$ x' $=$ (0.0 **fby** (x + 1))
every up(ez) **init** 0.0

$G, H \vdash_{?}$ **der** y $= \cdots$ **and** x $= \cdots$ ✗

$G, H \vdash_{?}$ **der** y$= \cdots$ **and** x' $= \cdots$

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

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

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

$$j : (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}}) \longrightarrow (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}})$$

Typing

$G, H \vdash_{\mathtt{C}}$ **der** y $=$ 1.0 **init** 0.0

$G, H \vdash_{\mathtt{D}}$ x $=$ (0.0 **fby** (x + 1))

$G, H \vdash_{\mathtt{C}}$ x' $=$ (0.0 **fby** (x + 1))
every up(ez) **init** 0.0

$G, H \vdash_{?}$ **der** y $= \cdots$ **and** x $= \cdots$ ✗

$G, H \vdash_{\mathtt{C}}$ **der** y$= \cdots$ **and** $x' = \cdots$ ✓

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

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

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

$$j : (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}}) \longrightarrow (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}})$$

Typing

$G, H \vdash_{\mathtt{C}}$ **der** y $=$ 1.0 **init** 0.0

$G, H \vdash_{\mathtt{D}}$ x $=$ (0.0 **fby** (x + 1))

$G, H \vdash_{\mathtt{C}}$ x' $=$ (0.0 **fby** (x + 1))
every up(ez) **init** 0.0

$G, H \vdash_{?}$ **der** y $= \cdots$ **and** x $= \cdots$ ✗

$G, H \vdash_{\mathtt{C}}$ **der** y$= \cdots$ **and** $x' = \cdots$ ✓

$G, H \vdash_{?}$ $x = \cdots$ **and** $x = \cdots$

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

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

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

$$j : (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}}) \longrightarrow (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}})$$

Typing

$G, H \vdash_{\mathtt{C}}$ **der** y $=$ 1.0 **init** 0.0

$G, H \vdash_{\mathtt{D}}$ x $=$ (0.0 **fby** (x + 1))

$G, H \vdash_{\mathtt{C}}$ x' $=$ (0.0 **fby** (x + 1))
every up(ez) **init** 0.0

$G, H \vdash_{?}$ **der** y $= \cdots$ **and** $x = \cdots$ ✗

$G, H \vdash_{\mathtt{C}}$ **der** y$= \cdots$ **and** $x' = \cdots$ ✓

$G, H \vdash_{\mathtt{D}}$ $x = \cdots$ **and** $x = \cdots$ ✓

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

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

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

$$j : (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}}) \longrightarrow (\mathtt{float}_{\mathtt{D}}) \times (\mathtt{float}_{\mathtt{C}})$$

Typing

$G, H \vdash_{\texttt{C}}$ **der** y $=$ 1.0 **init** 0.0

$G, H \vdash_{\texttt{D}}$ x $=$ (0.0 **fby** (x + 1))

$G, H \vdash_{\texttt{C}}$ x' $=$ (0.0 **fby** (x + 1)) **every** up(ez) **init** 0.0

$G, H \vdash_{?}$ **der** y $= \cdots$ **and** x $= \cdots$ ✗

$G, H \vdash_{\texttt{C}}$ **der** y$= \cdots$ **and** $x' = \cdots$ ✓

$G, H \vdash_{\texttt{D}}$ $x = \cdots$ **and** $x = \cdots$ ✓

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

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

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

$$j : (\texttt{float}_{\texttt{D}}) \times (\texttt{float}_{\texttt{C}}) \longrightarrow (\texttt{float}_{\texttt{D}}) \times (\texttt{float}_{\texttt{C}})$$

What about continuous automata?

Stateflow User's Guide The Mathworks, pages 16-26 to 16-29, 2011.

16 Modeling Continuous-Time Systems in Stateflow® Charts

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.

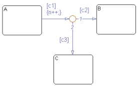
In Stateflow charts, physical events cause state transitions. Therefore, write to local data only in actions that execute during transitions, as follows:

16-26

Design Considerations for Continuous-Time Modeling in Stateflow® Charts

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

  Consider the following chart.



In this example, the action `{n++}` executes even when conditions `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

16-27

16 Modeling Continuous-Time Systems in Stateflow® Charts

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.

16-28

. . .

- '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

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

```
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

```
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

```
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

```
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

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

 automaton
 | Await →
   do

     der y = 0.0

    until start then Bounce(y'0)
   done

 | Bounce(v) →
    local c, y' in
   do

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

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

 in
 y
```

Typing rules

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

Automata

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

 automaton
 | Await →
   do
     der y = 0.0      C
   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)          C
   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

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

 automaton
 | Await →
   do

     der y = 0.0                D

    until start then Bounce(y'0)
   done    zero :: C

 | Bounce(v) →
    local c, y' in
   do

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

    until c on (y' < eps) then Await
        | c then Bounce(−0.9 *. y')
   done  zero :: C        D
 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

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

 automaton
 | Await →
   do

     der y = 0.0                D

    until start then Bounce(y'0)
   done     zero :: C

 | Bounce(v) →
    local c, y' in
   do

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

    until c on (y' < eps) then Await
        | c then Bounce(−0.9 *. y')
   done    zero :: C          D
 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

```
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)
  - $\cdot\,\mathtt{on}\,\cdot : \mathtt{zero} \times \mathtt{bool} \xrightarrow{\mathtt{A}} \mathtt{zero}$

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)
  - $\cdot\,\mathtt{on}\,\cdot : \mathtt{zero} \times \mathtt{bool} \xrightarrow{\mathtt{A}} \mathtt{zero}$

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)
  - $\cdot\,\mathtt{on}\,\cdot : \mathtt{zero} \times \mathtt{bool} \xrightarrow{A} \mathtt{zero}$

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)
  - $\cdot\,\mathtt{on}\,\cdot : \mathtt{zero} \times \mathtt{bool} \xrightarrow{\mathtt{A}} \mathtt{zero}$

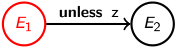
Strong and weak transitions

transition **discrete**

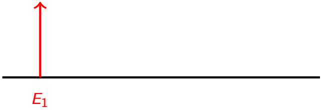
$$E_1 \xrightarrow{\textbf{unless}\ z} E_2$$

Strong and weak transitions

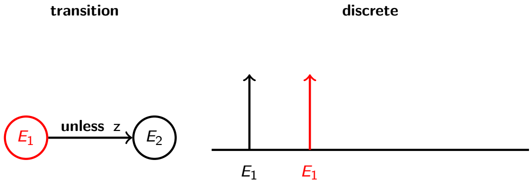
transition



discrete



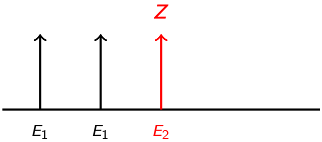
Strong and weak transitions



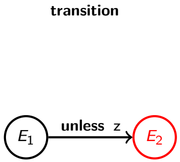
Strong and weak transitions

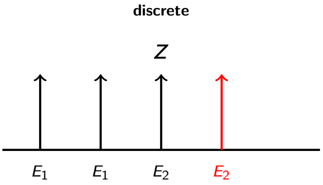
transition

discrete



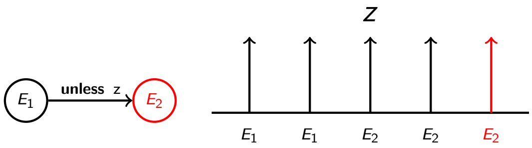
Strong and weak transitions



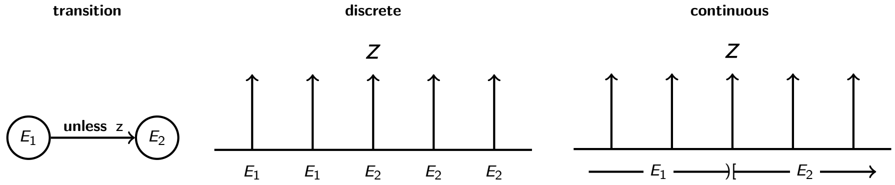


Strong and weak transitions

transition **discrete**

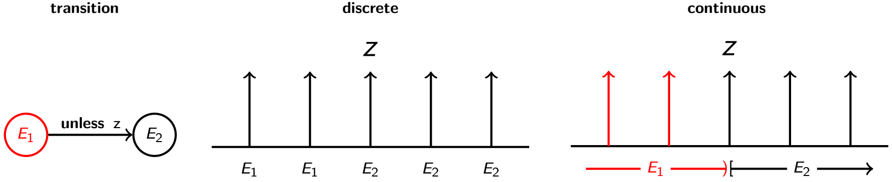


Strong and weak transitions



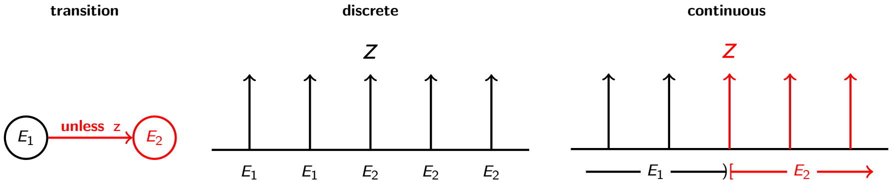
- Synchronous languages ignore the gaps between reactions
- But a hybrid language cannot
- Strong preemption: ok *(state entry on discrete step)*

Strong and weak transitions



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

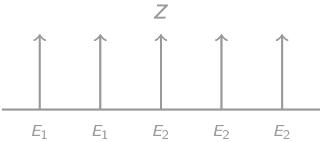
Strong and weak transitions

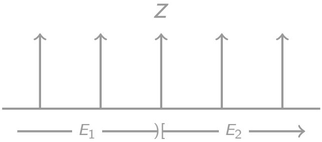


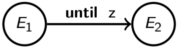
- Synchronous languages ignore the gaps between reactions
- But a hybrid language cannot
- Strong preemption: ok *(state entry on discrete step)*

Strong and weak transitions

transition **discrete** **continuous**

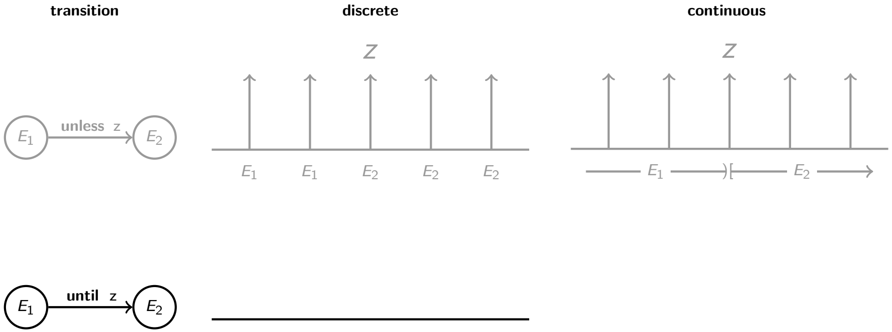






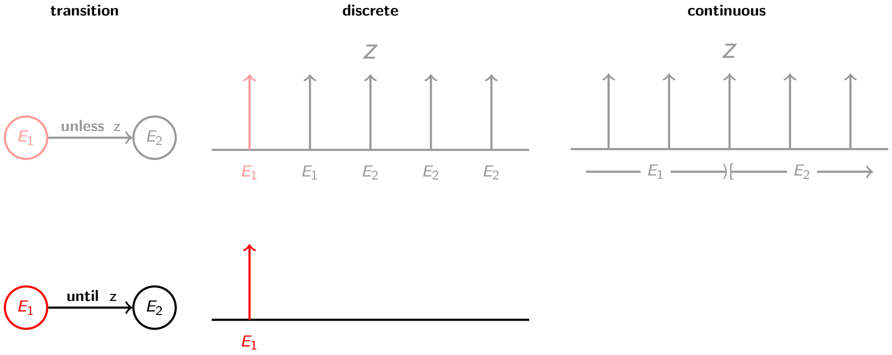
- Weak preemption: ...

Strong and weak transitions



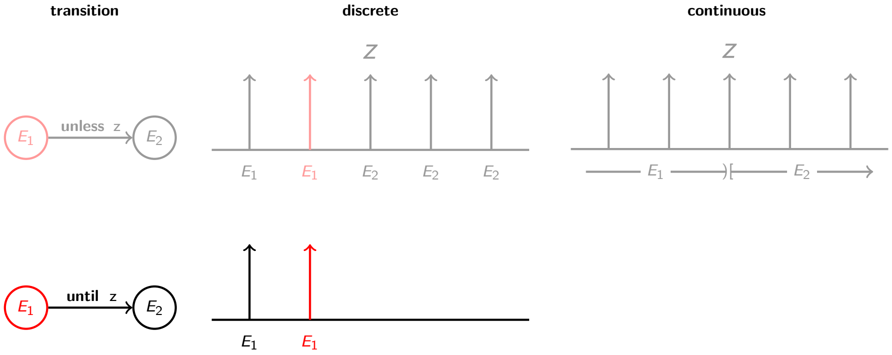
- Weak preemption: ...

Strong and weak transitions



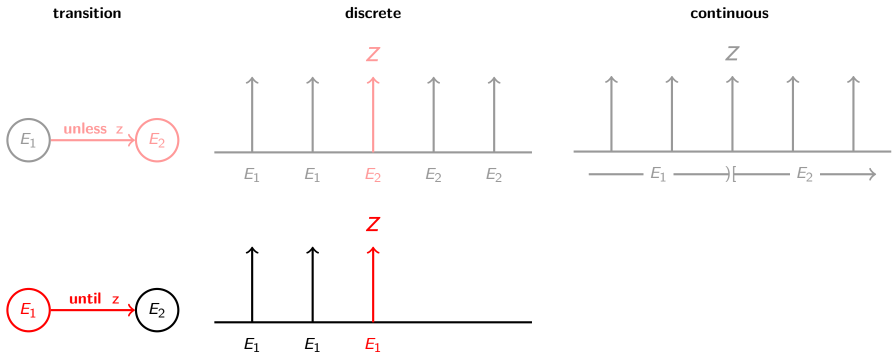
- Weak preemption: ...

Strong and weak transitions



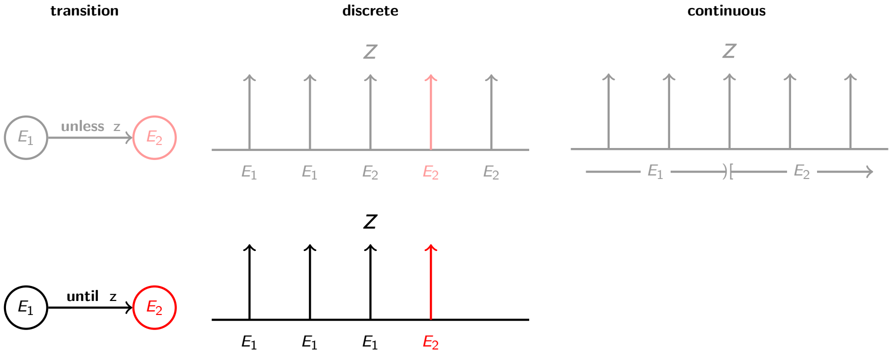
- Weak preemption: ...

Strong and weak transitions



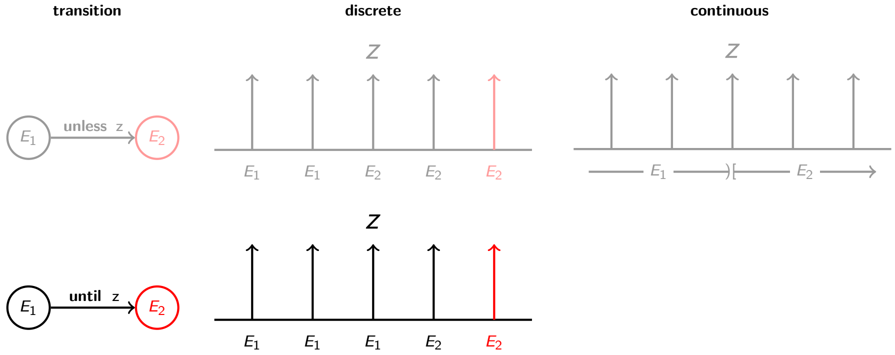
- Weak preemption: ...

Strong and weak transitions



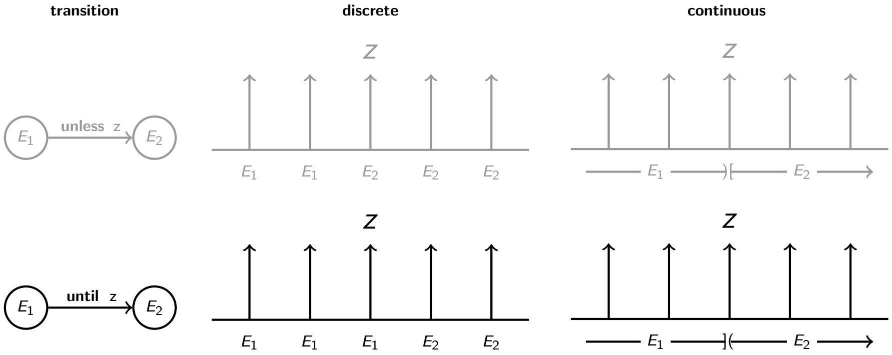
- Weak preemption: . . .

Strong and weak transitions



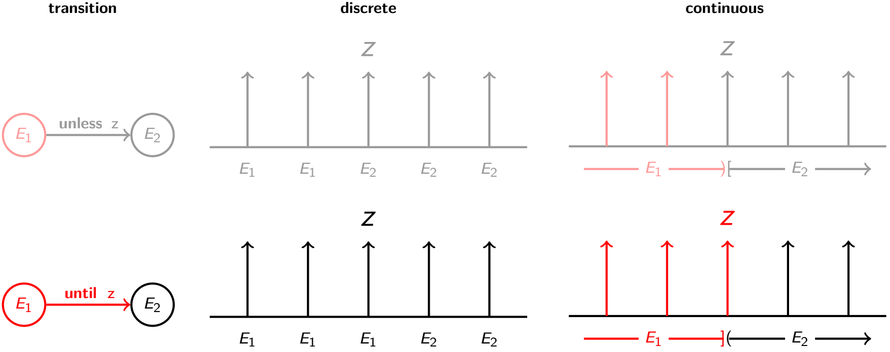
- Weak preemption: ...

Strong and weak transitions



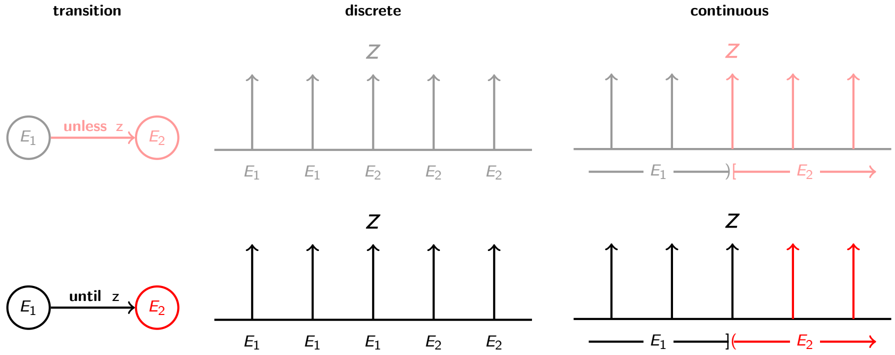
- Weak preemption: trickier

Strong and weak transitions



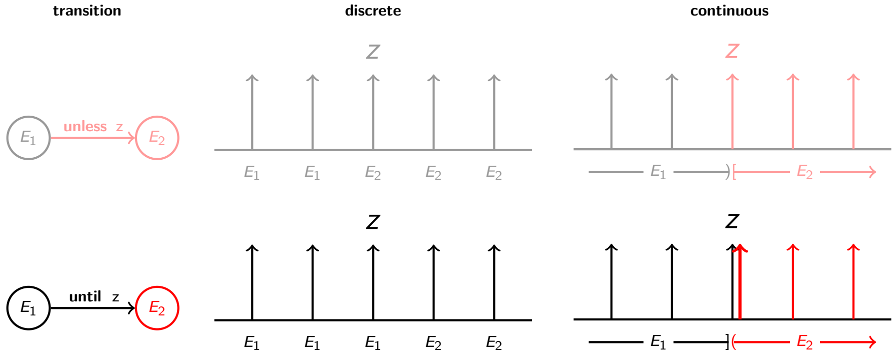
- Weak preemption: trickier
- state exit on discrete step

Strong and weak transitions



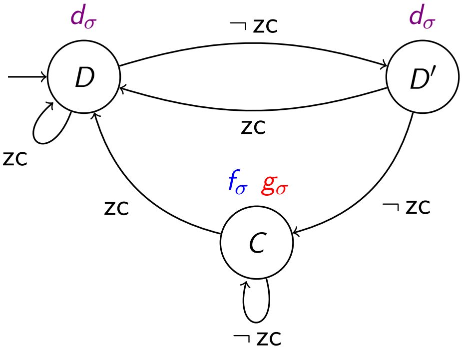
- Weak preemption: trickier
- state exit on discrete step

Strong and weak transitions



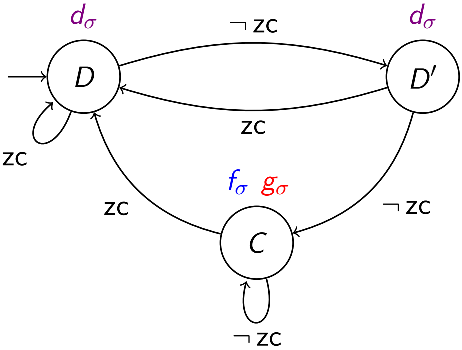
- Weak preemption: trickier
- state exit on discrete step
- need an extra discrete step for state entry

Execution (Simulation)



- Only d may have side effects and change the discrete state (σ)
- 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 (σ)
- 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 σ is stable (no need for D')

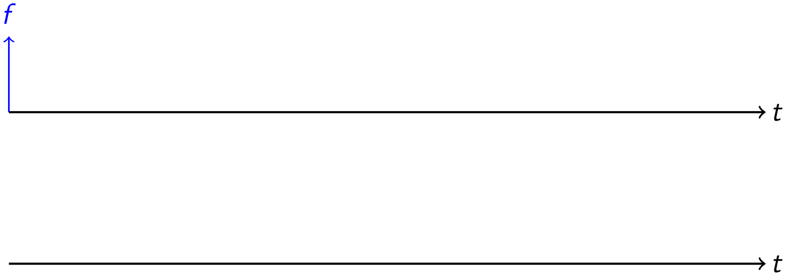
Solver execution

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

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

Solver execution

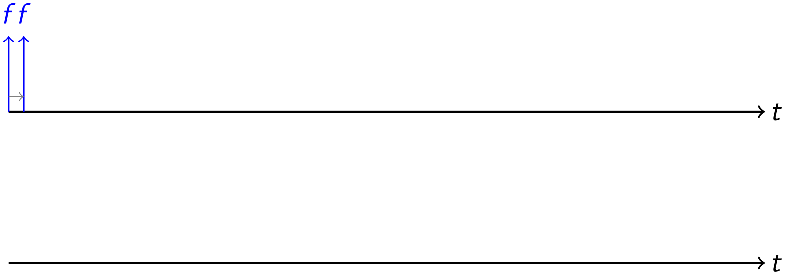
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

Solver execution

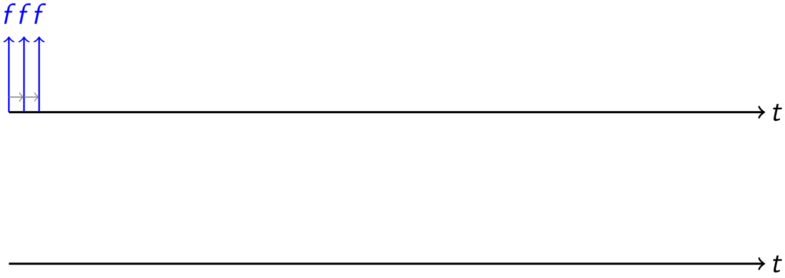
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

Solver execution

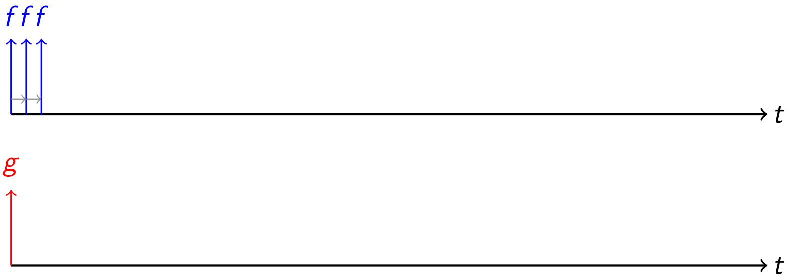
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

Solver execution

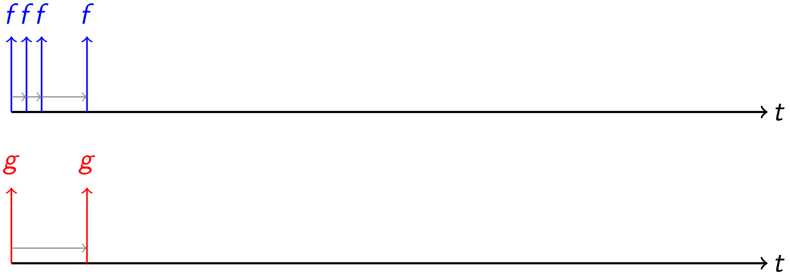
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

Solver execution

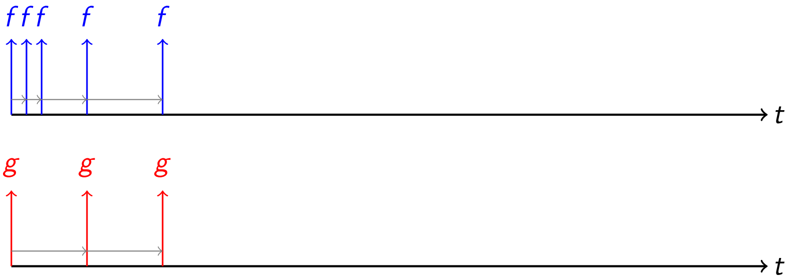
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

Solver execution

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

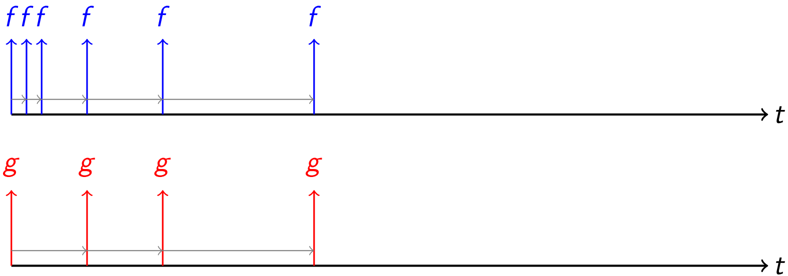


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - Cannot change state within f or g
  - Guaranteed for well-typed programs

Solver execution

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

1. approximation error too large

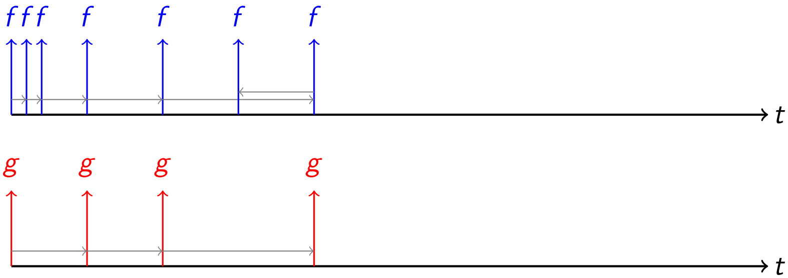


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - Cannot change state within f or g
  - Guaranteed for well-typed programs

Solver execution

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

1. approximation error too large

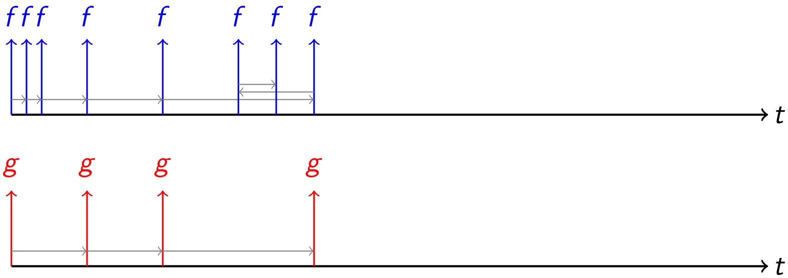


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - Cannot change state within f or g
  - Guaranteed for well-typed programs

Solver execution

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

1. approximation error too large

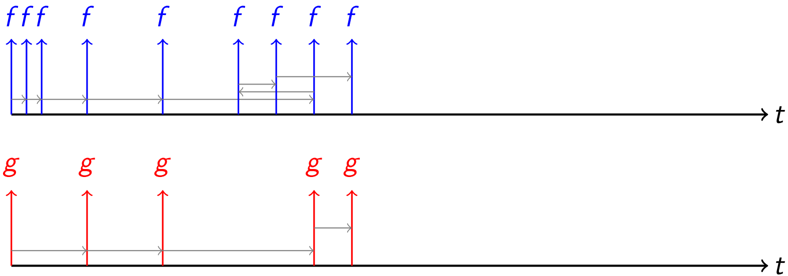


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - Cannot change state within f or g
  - Guaranteed for well-typed programs

Solver execution

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

1. approximation error too large

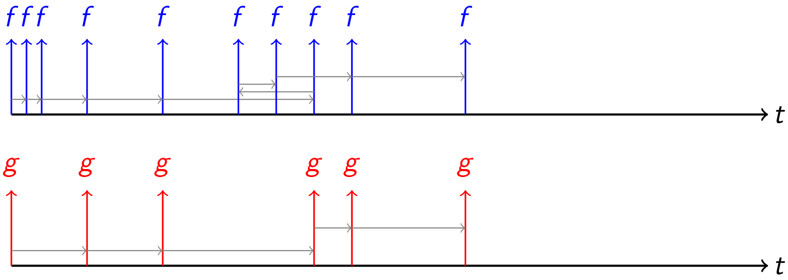


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - Cannot change state within f or g
  - Guaranteed for well-typed programs

Solver execution

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

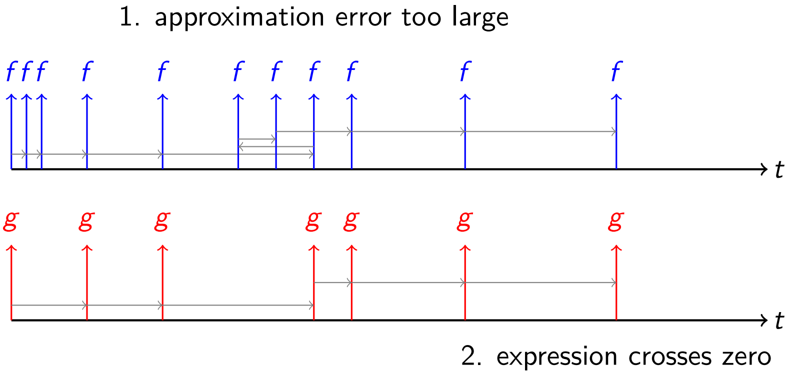
1. approximation error too large



- 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

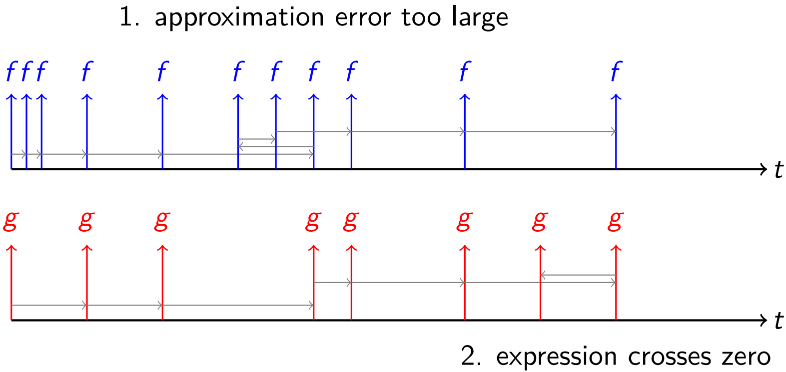
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



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

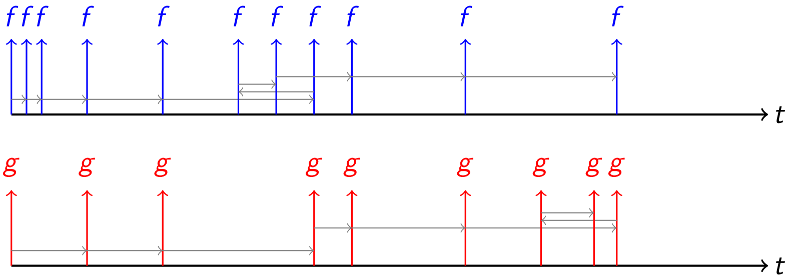


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - Cannot change state within f or g
  - Guaranteed for well-typed programs

Solver execution

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

1. approximation error too large

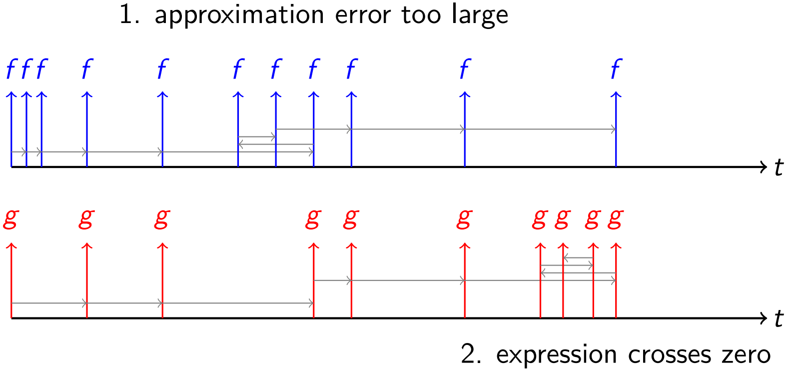


2. expression crosses zero

- 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

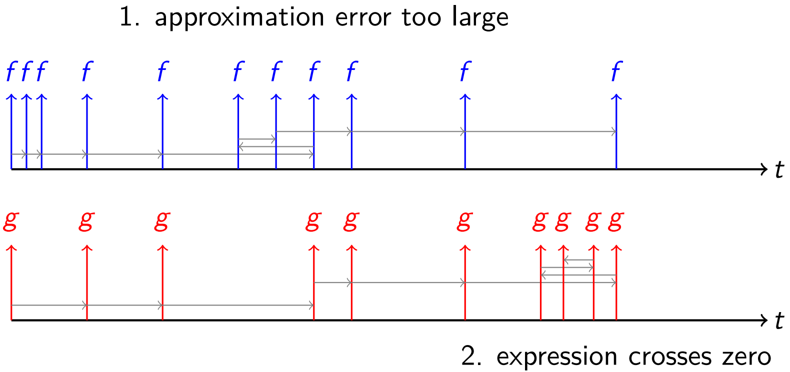
Give solver two functions: $dy = f_\sigma(t, y)$, $upz = g_\sigma(t, y)$



- 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

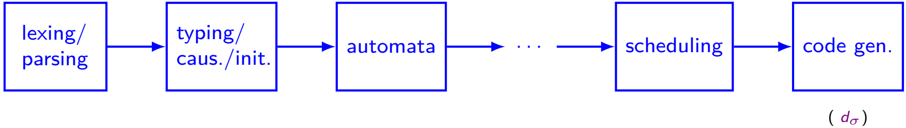
Solver execution

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

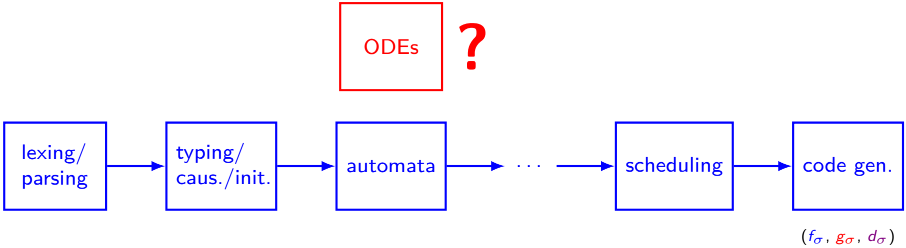


- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - Cannot change state within f or g
  - Guaranteed for well-typed programs

Source-to-source transformation

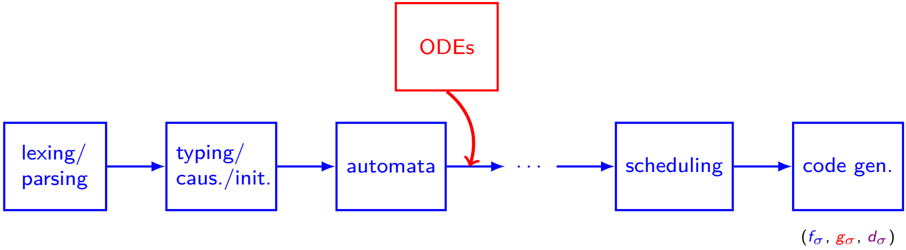


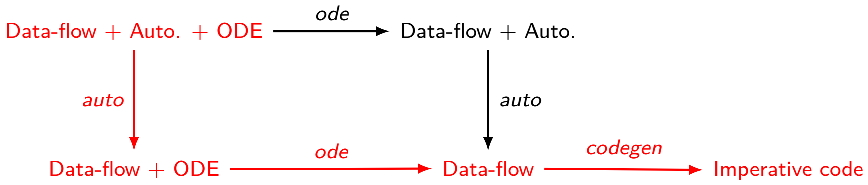
Source-to-source transformation



$$\begin{tikzcd}
\text{Data-flow} + \text{Auto.} + \text{ODE} \arrow[r, "\textit{ode}"] \arrow[d, "\textit{auto}"'] & \text{Data-flow} + \text{Auto.} \arrow[d, "\textit{auto}"] & \\
\text{Data-flow} + \text{ODE} \arrow[r, "\textit{ode}"] & \text{Data-flow} \arrow[r, "\textit{codegen}"] & \text{Imperative code}
\end{tikzcd}$$

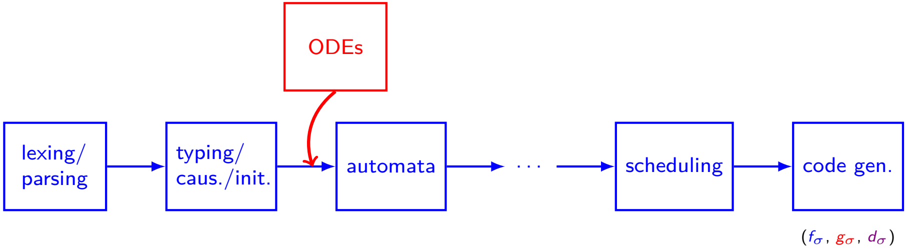
Source-to-source transformation





- Pro: simpler definition of ODE
- Con: subtle invariant over intermediate language

Source-to-source transformation



$$
\begin{tikzcd}
\text{Data-flow + Auto. + ODE} \arrow[r, "ode"] \arrow[d, "auto"'] & \text{Data-flow + Auto.} \arrow[d, "auto"] & \\
\text{Data-flow + ODE} \arrow[r, "ode"] & \text{Data-flow} \arrow[r, "codegen"] & \text{Imperative code}
\end{tikzcd}
$$

- Pro: intermediate result is well-typed
- Pro/Con: ODE code must include cases for automata

Source-to-source transformation details

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



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


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

Source-to-source transformation details

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



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


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

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

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

- Source-to-source transformation (to give f_σ, g_σ, d_σ)

Source-to-source transformation details

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

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

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

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

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

- Source-to-source transformation (to give f_σ, g_σ, d_σ)
- Transform each hybrid function into a discrete one

Source-to-source transformation details

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

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

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

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

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

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

Source-to-source transformation details

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

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

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

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

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

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

Source-to-source transformation details

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


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


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

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

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

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

Source-to-source transformation details

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


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


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

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

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

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

Source-to-source transformation details

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

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

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

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

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

- Zero-crossing operators, up(·), are also ‘externalized’
- Detection always occurs externally; boolean values internally

Source-to-source transformation details

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

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

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

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

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

- Zero-crossing operators, $\mathtt{up}(\cdot)$, 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)