Divide and Recycle: Types and Compilation for a Hybrid Synchronous Language

Albert Benveniste[1] Benoît Caillaud[1]
Timothy Bourke[1] Marc Pouzet[1,2,3]

1. INRIA

2. Institut Universitaire de France

3. École normale supérieure (LIENS)

LCTES 2011, CPS Week, April 11–14, Chicago, IL, USA

Motivation

Hybrid Systems Modelers — Simulink, Ptolemy, ...

- Platforms for simulation and **development**
- More and more important
  - Semantics
  - Efficiency and predictability
  - Fidelity / Consistency

Conservative extension of a synchronous data-flow language

What distinguishes our approach?

- Compilation with existing tools
  (after source-to-source transformation)
- Static typing
- Semantics based on non-standard analysis

Outline

Modeling

Model discrete systems with data-flow equations

Model physical systems with Ordinary Differential Equations (ODEs)

$$\dot{\mathbf{y}}(t) = f(t, \mathbf{y})$$

instantaneous derivatives

variables

$$\mathbf{y}(0) = \mathbf{y}_i$$

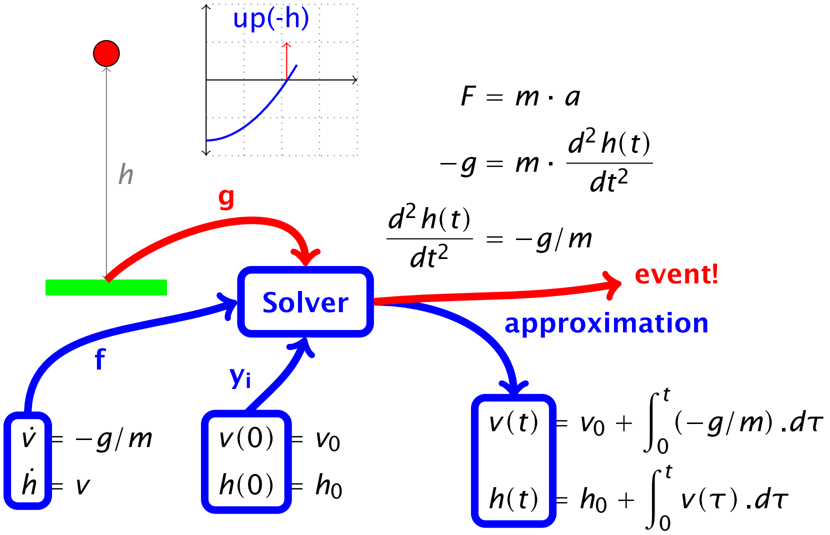
initial values

(Causal) First-order ODEs

- Causal: inputs on right,
  outputs on left
- First-order:
  one equation = one variable

Bouncing ball

model



$$F = m \cdot a$$

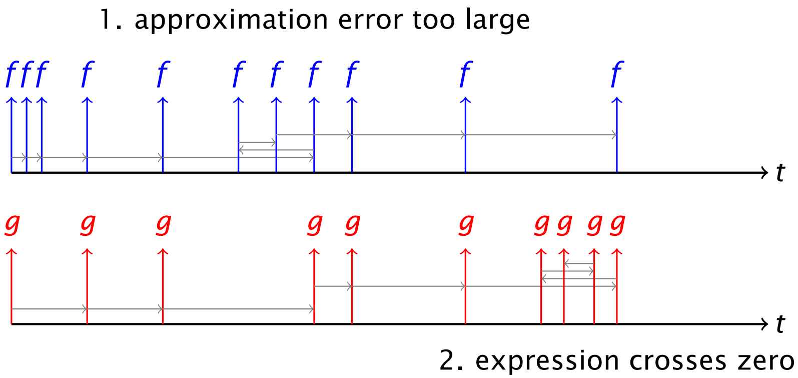
$$-g = m \cdot \frac{d^2 h(t)}{dt^2}$$

$$\frac{d^2 h(t)}{dt^2} = -g/m$$

$$\dot{v} = -g/m \qquad v(0) = v_0 \qquad v(t) = v_0 + \int_0^t (-g/m)\,.d\tau$$

$$\dot{h} = v \qquad h(0) = h_0 \qquad h(t) = h_0 + \int_0^t v(\tau)\,.d\tau$$

Solver execution



- Bigger and bigger steps (bound by h_{min} and h_{max})
- t does not necessarily advance monotonically
  - Ok for continuous states (managed by solver)
  - Cannot change state within f or g

Basic Hybrid Language

Start with Lucid Synchrone (subset); add first-order ODEs with reset.

$$\dot{\mathbf{x}}(t) = f(t, \mathbf{x}) \qquad \qquad \mathbf{x}(0) = \mathbf{x}_i$$

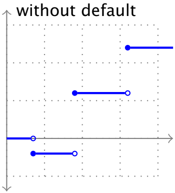
instantaneous derivatives

variables

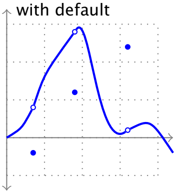
initial values

Rather than $\dot{x} = e_d$ and $x(0) = x_i$, write

$$\begin{array}{llllll} \textbf{der}\ x = e_d\ \textbf{init}\ x_i & \textbf{reset} & e_1 & \textbf{every} & \mathrm{up}(e_{z_1}) \\ \quad\cdots & \mid & e_n & \textbf{every} & \mathrm{up}(e_{z_n}) \end{array}$$



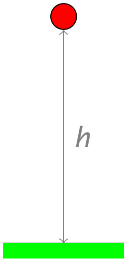
$$\begin{array}{ll} x = & (\mathrm{pre}\ \mathrm{h} + 1)\ \textbf{every}\ \mathrm{up}(e)\ \textbf{default}\ e_c\ \textbf{init}\ e_i \\ \quad \mid & \textit{purely sync}\ \textbf{every}\ \textit{event} \end{array}$$



Very simple: no clocks, no automata, no higher-order

Bouncing ball

program



$$\dot{v} = -g/m \qquad v(0) = v_0$$

$$\dot{h} = v \qquad h(0) = h_0$$

reset v to $-0.8 \cdot v$ when h becomes 0

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

Semantics

reals — *non-standard reals*

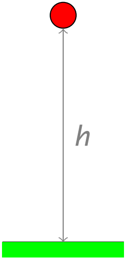
$\mathbb{R}$ + infinitesimals (∂) ${}^{\star}\mathbb{R}$

$$\cdots < t-3\partial < t-2\partial < t-\partial < t < t+\partial < t+2\partial < t+3\partial < \cdots$$

- dense and discrete
- base clock for both continuous and discrete behaviors
- $\forall t.\ {}^{\bullet}t$ is the previous instant, $t^{\bullet}$ is the next instant

$$
\begin{array}{lcll}
integr^{\#}(T)(s)(s_0)(hs)(t) & = & s'(t) & \text{where} \\
s'(t) & = & s_0(t) & \text{if } t = \min(T) \\
s'(t) & = & s'({}^{\bullet}t) + \partial s({}^{\bullet}t) & \text{if } handler^{\#}(T)(hs)(t) = NoEvent \\
s'(t) & = & v & \text{if } handler^{\#}(T)(hs)(t) = Xcrossing(v) \\
 & & & \\
up^{\#}(T)(s)(t) & = & \mathtt{false} & \text{if } t = \min(T) \\
up^{\#}(T)(s)(t^{\bullet}) & = & \mathtt{true} & \text{if } (s({}^{\bullet}t) \leq 0) \wedge (s(t) > 0) \text{ and } (t \in T) \\
up^{\#}(T)(s)(t^{\bullet}) & = & \mathtt{false} & \text{otherwise} \\
\ldots & & \ldots & \ldots
\end{array}
$$

Compilation




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

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

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

  and dh
  and h

  and up

  in ((v, h), upz1, (h, v), (dh, dv))
```

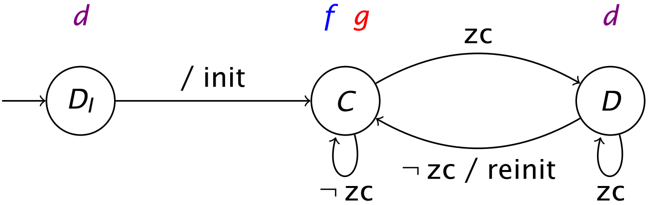
transfor transform zero-crossings

All continuous parts execute in 1st instant

- type system prevents C inside D
- no branching or activations

Execution (Simulation)

$$\begin{array}{lcll} (upz, y, dy) & = & main_\sigma(z, y) & \\ f\,(t, y) & = & \text{let } (\_, \_, dy) = main_\sigma(false, y) & \text{in } dy \\ g\,(t, y) & = & \text{let } (upz, \_, \_) = main_\sigma(false, y) & \text{in } upz \\ d\,(z, y) & = & \text{let } (upz, y, \_) = main_\sigma(z, y) & \text{in } (upz, y) \end{array}$$



- Only d may have side effects
- Neither f, nor g may change the (internal) discrete state

Typing
Motivation

This compilation/execution scheme only works for some programs!

We need a type system to:

- Reject programs that do not respect the invariant:
  - discrete computations in D only
  - continuous evolutions in C only
- Reject unreasonable programs
  - where behavior depends 'too much' on simulation parameters (like the step size, or number of iterations)

Typing

Unreasonable programs

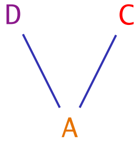
```
der y = 1.0 init 0.0        and    x = (0.0 → pre x) + y

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

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



Initial conditions

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

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

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

Conclusion

- Simple extension of a synchronous data-flow language
  - Add first-order ODEs
  - and zero-crossing events
- Non-standard semantics
  - Gives a 'continuous base clock'
  - Simplifies definitions, clarifies certain features
- Static block-based typing system
  - Divide system into continuous and discrete parts
- Compilation
  - Source-to-source transformation
  - Recycle existing compilers
- Execution
  - Simulate using Sundials CVODE solver

Ocaml Sundials CVODE interface and compiler available