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

Albert Benveniste¹  Benoît Caillaud¹
Timothy Bourke¹  Marc Pouzet¹,²,³

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

- 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

Background

Hybrid Synchronous Language
  Semantics
  Compilation
  Execution
  Typing

Conclusion
Modeling

Model **discrete** systems with data-flow equations

Model **physical** systems with Ordinary Differential Equations (ODEs)

\[ \dot{y}(t) = f(t, y) \]

**(Causal) First-order ODEs**

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

\[ y(0) = y_i \]
Bouncing ball model

\[ F = m \cdot a \]
\[ -g = m \cdot \frac{d^2 h(t)}{dt^2} \]
\[ \frac{d^2 h(t)}{dt^2} = -\frac{g}{m} \]

\[ \dot{v} = -\frac{g}{m} \]
\[ \dot{h} = v \]
\[ v(0) = v_0 \]
\[ h(0) = h_0 \]

\[ v(t) = v_0 + \int_0^t (-\frac{g}{m}) \cdot d\tau \]
\[ h(t) = h_0 + \int_0^t v(\tau) \cdot d\tau \]
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
  - 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.

\[ \dot{x}(t) = f(t, x) \]
\[ x(0) = x_i \]

instantaneous derivatives

variables

initial values

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

\[
\text{der } x = e_d \quad \text{init } x_i \quad \text{reset } e_1 \quad \text{every } \text{up}(e_{z_1}) \\
\cdots \quad | \quad e_n \quad \text{every } \text{up}(e_{z_n})
\]

\[ x = (\text{pre } h + 1) \quad \text{every } \text{up}(e) \quad \text{default } e_c \quad \text{init } e_i \\
| \quad \text{purely sync } \text{every } \text{event} \]

Very simple: no clocks, no automata, no higher-order
Bouncing ball
program

\[ \dot{v} = -\frac{g}{m} \quad v(0) = v_0 \]
\[ \dot{h} = v \quad h(0) = h_0 \]

reset \( v \) to \(-0.8 \cdot v\) when \( h \) becomes 0

```plaintext
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**

\[ \mathbb{R} \]

+ infinitesimals \((\partial)\)

**non-standard reals**

\[ \star \mathbb{R} \]

\[ \cdots < t - 3\partial < t - 2\partial < t - \partial < t < t + \partial < t + 2\partial < t + 3\partial < \cdots \]

- dense and discrete
- base clock for both continuous and discrete behaviors
- \( \forall t \). \( \cdot t \) is the previous instant, \( t^* \) is the next instant

---

**integr**\((T)(s)(s_0)(hs)(t)\) \(= s'(t) \) where

- \( s'(t) = s_0(t) \) if \( t = \min(T) \)
- \( s'(t) = s'(\cdot t) + \partial s(\cdot t) \) if \( \text{handler}^\#(T)(hs)(t) = \text{NoEvent} \)
- \( s'(t) = v \) if \( \text{handler}^\#(T)(hs)(t) = \text{Xcrossing}(v) \)

**up**\((T)(s)(t)\) \(= \) false if \( t = \min(T) \)

- \( \text{up}^\#(T)(s)(t^*) = \) true if \( (s(\cdot t) \leq 0) \wedge (s(t) > 0) \) and \( t \in T \)
- \( \text{up}^\#(T)(s)(t^*) = \) false otherwise

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

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

\[(upz, y, dy) = main_\sigma(z, y)\]
\[f(t, y) = \text{let } (_, _, dy) = main_\sigma(false, y) \text{ in } dy\]
\[g(t, y) = \text{let } (upz, _, _) = main_\sigma(false, y) \text{ in } upz\]
\[d(z, y) = \text{let } (upz, y, _) = main_\sigma(z, y) \text{ in } (upz, y)\]

- 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

\[
\text{der } y = 1.0 \text{ init } 0.0 \quad \text{and} \quad x = (0.0 \rightarrow \text{pre} \ x) + y
\]

\[
x = 0.0 \rightarrow (\text{pre} \ x + .1.0) \quad \text{and} \quad \text{der} \ y = x \text{ init } 0.0
\]

- y is a variable that changes \textit{continuously}
- x is \textit{discrete} register
- The relationship between the two is ill-defined
Typing

The type language

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

Initial conditions

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

\[ G, H \vdash_c \text{der} \ y = 1.0 \ \text{init} \ 0.0 \]
\[ G, H \vdash_D x = (0.0 \ \text{fby} \ (x + 1)) \]
\[ G, H \vdash_c x' = (0.0 \ \text{fby} \ (x + 1)) \]

\[ G, H \vdash_? \ \text{der} \ y = \cdots \ \text{and} \ x = \cdots \ \times \]

\[ G, H \vdash_c \ \text{der} \ y = \cdots \ \text{and} \ x' = \cdots \ \checkmark \]

\[ G, H \vdash_D x = \cdots \ \text{and} \ x = \cdots \ \checkmark \]

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

\[ h : \text{float} \times \text{float} \xrightarrow{k} \text{float} \times \text{float} \]

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

\[ j : (\text{float}_D) \times (\text{float}_C) \xrightarrow{} (\text{float}_D) \times (\text{float}_C) \]
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