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

Marc Pouzet
LIENS
Institut Universitaire de France
Marc.Pouzet@ens.fr

GDR – AFSEC
Lille, June 9th, 2011.

---

a. Joint work with Albert Benveniste, Timothy Bourke and Benoit Caillaud
Motivation and Context

- **Hybrid modelers** allow to program both a (discrete) controller and its physical (continuous) environment in the very same language.
- We focus on **explicit** modelers
- A lot of result on the formal verification of the sub-class of **hybrid automata** but relatively few on **programming language questions**. : typing, semantics, compilation.

What is the problem?
- Hybrid modelers (e.g., Simulink) widely used but they lack a formally defined semantics and code generation.

A new approach:
- **Extend** a synchronous language where dataflow equations are mixed with ODE.
- Make it **conservative**, i.e., nothing must change for the discrete subset (same typing, same code generation).
- Static typing to **divide** discrete from continuous signals
- **Recycle** existing synchronous compilers and numerical solvers to execute them.
Parallel composition : homogeneous case

Two equations with discrete time :

\[ f = 0.0 \rightarrow \text{pre} \ f + s \quad \text{and} \quad s = 0.2 \times (x - \text{pre} \ f) \]

and the initial value problem :

\[ \text{der}(y') = -9.81 \ \text{init} \ 0.0 \quad \text{and} \quad \text{der}(y) = y' \ \text{init} \ 10.0 \]

The first program can be written in any synchronous language, e.g. LUSTRE. The semantics of \( f \) and \( s \) are infinite sequences :

\[ \forall n \in \mathbb{N}^*, f_n = f_{n-1} + s_n \quad \text{and} \quad f_0 = 0 \quad \forall n \in \mathbb{N}, s_n = 0.2 \times (x_n - f_{n-1}) \]

The second program can be written in any hybrid modeler, e.g. SIMULINK, and its semantics is :

\[ \forall t \in \mathbb{R}^+, y'(t) = 0.0 + \int_0^t -9.81 \, dt = -9.81 \, t \]

\[ \forall t \in \mathbb{R}^+, y(t) = 10.0 + \int_0^t y'(t) \, dt = 10.0 - 9.81 \int_0^t t \, dt \]

Parallel composition is clear since equations share the same time scale.
Parallel composition : heterogeneous case

Two equations: a signal defined at discrete instants, the other continuously.

\[
der(t) = 1.0 \ \text{init} \ 0.0 \ \text{and} \ x = 0.0 \ \text{fby} \ x + t
\]
or:

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

- time is a continuous signal (\(\forall t \in \mathbb{R}_+, \text{time}(t) = t\)).
- x is discrete (\(\forall n \in \mathbb{N}, x_n = n + 1\)).

It would be tempting to define the first equation as: \(\forall n \in \mathbb{N}, x_n = x_{n-1} + \text{time}(n)\)
And the second as:

\[
\forall n \in \mathbb{N}^*, x_n = x_{n-1} + 1.0 \ \text{and} \ x_0 = 1.0
\]

\[
\forall t \in \mathbb{R}_+, y(t) = 0.0 + \int_0^t x(t) \, dt
\]
i.e., \(x(t)\) as a piecewise constant function from \(\mathbb{R}_+\) to \(\mathbb{R}_+\) with \(\forall t \in \mathbb{R}_+, x(t) = x_{\lfloor t \rfloor}\).

In both cases, this would be a mistake. x is defined on a discrete, logical time; time on a continuous, absolute time.
Equations with reset

Two independent groups of equations.

\[
\text{der}(p) = 1.0 \ \text{init} \ 0.0 \ \text{reset} \ 0.0 \ \text{every} \ \text{up}(p - 1.0)
\]
and
\[
x = 0.0 \ \text{fby} \ x + p
\]
and
\[
\text{der}(\text{time}) = 1.0 \ \text{init} \ 0.0
\]
and
\[
z = \text{up}({\sin (\text{freq} \times \text{time})})
\]

Properly translated in Simulink, changing \texttt{freq} changes the output of \textit{x}!

If \texttt{f} is running on a continuous time basis, what would be the meaning of:

\[
y = \texttt{f(x)} \ \text{every} \ \text{up}(z) \ \text{init} \ 0
\]

All these programs are \textbf{wrongly typed} and should be statically rejected.
Discrete vs Continuous time signals

A signal is discrete if it is activated on a discrete clock.

A clock is termed discrete if it has been declared so or if it is the result of a zero-crossing or a sub-sampling of a discrete clock. Otherwise, it is termed continuous.

Ensure it statically with a type language and type system for a synchronous language extended with ODEs.
Notations

- \texttt{up}(e) tests the zero-crossing of expression \(e\) (from negative to positive).
- If \(x = \texttt{up}(e)\), all handlers using \(x\) are governed by the same zero-crossing.
- Handlers have priorities. E.g., a piece-wise constant signal \(z\).

\[
z = 1 \text{ every } \texttt{up}(x) \mid 2 \text{ every } \texttt{up}(y) \text{ init } 0
\]

- \texttt{last}(x) for the left-limit of signal \(x\).

\[
z = \texttt{last } z + 1 \text{ every } \texttt{up}(x) \mid \texttt{last } z - 1 \text{ every } \texttt{up}(y) \text{ init } 0
\]

- ODEs can be reset on a discrete signal.

\[
\text{der } y' = -9.81 \text{ init } 0.0 \text{ reset } -0.9 \ast . \text{ last } y \text{ every } \texttt{up}(-y)
\]
Examples

Combinatorial and sequential function (discrete time).

```plaintext
let add (x,y) = x + y

let node counter(top, tick) = o where
   o = if top then i else 0 fby o + 1
   and i = if tick then 1 else 0

let node bangbang (error, low, high) = go where rec
   automaton
      | False -> do go = false until (error > high) then True done
      | True  -> do go = true until (error < low) then False done
end
```

- add get type signature : \( \text{int} \times \text{int} \rightarrow \text{int} \)
- counter get type signature : \( \text{bool} \times \text{bool} \rightarrow \text{int} \)
- bangbang get type signature : \( \forall \alpha. \alpha \times \alpha \times \alpha \rightarrow \text{bool} \)
The Bouncing ball

let hybrid bouncing(x0,y0,x’0,y’0) = (x,y) where
    der(x) = x’ init x0
and
    der(x’) = 0.0 init x’0
and
    der(y) = y’ init y0
and
    der(y’) = -. g init y’0 reset -. 0.9 *. last y’ every up(-. y)

Its type signature is: \( \text{float} \times \text{float} \times \text{float} \rightarrow \text{float} \times \text{float} \)
Connecting discrete to continuous time systems

let hybrid counter_ten(top, tick) = o where

(* a periodic timer 2.1(4) *)

    der(time) = 1.0 init -2.1 reset -4 every z

and z = up(time)

(* discrete function *)

and o = counter(top, tick) every z init 0

Type signature : bool × bool \(\rightarrow\) int.

The following is rejected

    der(p) = 1.0 init 0.0 reset 0.0 every up(p - 1.0)

and

    x = 0.0 fby x + p

Remark : provide ad-hoc programming constructs for periodic timers.
Typing

The type language

\[ \sigma ::= \forall \beta_1, \ldots, \beta_n.t \xrightarrow{k} t \]

\[ t ::= t \times t \mid \beta \mid bt \]

\[ k ::= D \mid C \mid A \]

\[ bt ::= \text{float} \mid \text{int} \mid \text{bool} \mid \text{zero} \]

Initial conditions

\[ (+) : \text{int} \times \text{int} \xrightarrow[A]{A} \text{int} \]

\[ (=) : \forall \beta.\beta \times \beta \xrightarrow[A]{A} \text{bool} \]

\[ \text{if} : \forall \beta.\text{bool} \times \beta \times \beta \xrightarrow[A]{A} \beta \]

\[ \text{der}() : \text{float} \xrightarrow[C]{C} \text{float} \]

\[ \text{pre}() : \forall \beta.\beta \xrightarrow[D]{D} \beta \]

\[ \text{fby}(). \quad \text{.fby.} : \forall \beta.\beta \times \beta \xrightarrow[D]{D} \beta \]

\[ \text{.->.} : \forall \beta.\beta \times \beta \xrightarrow[A]{A} \beta \]

\[ \text{up}() : \text{float} \xrightarrow[C]{C} \text{zero} \]
The Type system

Global and local environment

\[ G ::= [f_1 : \sigma_1; \ldots; f_n : \sigma_n] \]
\[ H ::= [\ ] \mid H, x : t \mid H, \text{last}(x) : t \]

Typing predicates

- \[ G, H \vdash_k e : t \]: Expression \( e \) has type \( t \) and kind \( k \). \( G, H \vdash_k e : t \)
- \[ H, H \vdash_k E : H' \]: Equation \( E \) produces environment \( H' \) and has kind \( k \).

Subtyping

An combinatorial function can be passed where a discrete or continuous one is expected:

\[ \forall k, A \leq k \]
A sketch of Typing rules

\[ \text{(DER)} \]
\[
G, H \vdash_c e_1 : \text{float} \quad G, H \vdash_c e_2 : \text{float} \quad G, H \vdash h : \text{float}
\]
\[
G, H \vdash_c \text{der}(x) = e_1 \text{ init } e_2 \text{ reset } h : [\text{last}(x) : \text{float}]
\]

\[ \text{(AND)} \]
\[
G, H \vdash_k E_1 : H_1 \quad G, H \vdash_k E_2 : H_2
\]
\[
G, H \vdash_k E_1 \text{ and } E_2 : H_1 + H_2
\]

\[ \text{(APP)} \]
\[
t \xrightarrow{k} t' \in \text{Inst}(G(f)) \quad G, H \vdash_k e : t
\]
\[
G, H \vdash_k f(e) : t'
\]

\[ \text{(EQ)} \]
\[
G, H \vdash_k e : t
\]
\[
G, H \vdash_k x = e : [x : t]
\]
(VAR)
\[ G, H + [x : t] \vdash_k x : t \]

(VAR-LAST)
\[ G, H + [\text{last}(x) : t] \vdash_k x : t \]

(EQ-DISCRETE)
\[ G, H \vdash h : t \quad G, H \vdash e : t \]
\[ \quad \frac{\quad G, H \vdash c x = h \ \text{init} e : [\text{last}(x) : t]}{\quad G, H \vdash_c x = h \ \text{init} e : [\text{last}(x) : t]} \]

(HANDLER)
\[ \forall i \in \{1, \ldots, n\} \quad G, H \vdash_D e_i : t \quad G, H \vdash_c z_i : \text{zero} \]
\[ \quad \frac{\quad G, H \vdash e_1 \ \text{every} \ z_1 | \ldots | e_n \ \text{every} \ z_n : t}{\quad G, H \vdash e_1 \ \text{every} \ z_1 | \ldots | e_n \ \text{every} \ z_n : t} \]
Non-standard Semantics [CDC’10, JCSS (2011)]

- Base clock both **dense** and **discrete**; \( \text{BaseClock} = \{n \partial \mid n \in \mathbb{N}^*\} \)
- \( \forall t. \ \star t \) is the previous instant, \( t^\bullet \) is the next instant

**Clock and signals** A clock \( T \) is a subset of \( \text{BaseClock} \). A signal \( s \) is a total function \( s : T \mapsto V \).

If \( T \) is a clock and \( b \) a signal \( b : T \mapsto \mathbb{B} \), then \( T \text{ on } b \) defines a subset of \( T \) comprising those instants where \( b(t) \) is true:

\[
T \text{ on } b = \{t \mid (t \in T) \land (b(t) = \text{true})\}
\]

If \( s : T \mapsto \mathbb{R}^* \), we write \( T \text{ on up}(s) \) for the instants when \( s \) crosses zero, that is:

\[
T \text{ on up}(s) = \{t^\bullet \mid (t \in T) \land (s(t^\bullet) \leq 0) \land (s(t) > 0)\}
\]
Compilation

The non-standard semantics is not operational. It serves as a reference to establish the correctness of the compilation. Two problems to address:

1. The compilation of the discrete part, that is, the synchronous subset of the language.
2. The compilation of the continuous part which is to be linked to a black-box numerical solver.

Principle

Translate the program into an only discrete one. Compile the result with an existing synchronous compiler such that it verifies the following invariant:

The discrete state, i.e., the values of delays, does not change when all of the zero-crossing conditions are false.

Said differently: when those conditions are false, the function is combinatorial.
Example (counter)

Add extra input and outputs.
- \textit{up}(e) becomes a fresh boolean input \( z \) and generate an equation \( \text{up}_z = e \).
- \textit{der}(x) = e \init e_0 \reset h \) becomes \( dx = e \) and \( x = h \init e_0 \default l_x \).
- A continuous state variable \textit{last}(x) becomes an input \( l_x \).

\begin{verbatim}
let node counter_ten([z],[ltime],(top, tick)) = (o,[upz],[time],[dtime])
where
dtime = 1.0 and time = default ltime init 0.0 reset 0.0 every z
and o = counter(top, tick) every z init 0
and upz = time -. 1.0
\end{verbatim}

In practice, represent these extra inputs with arrays.

Now, ignoring details of syntax, the function \texttt{counter_ten} can be processed by any synchronous compiler, and the generated transition function verifies the invariant.
Interfacing with a numerical solver

We used the Sundials CVODE (LLNL) library. An Ocaml interface has been done.

**Structure of the execution**: Run the transition function with two modes, a continuous one and a discrete one

- **Continuous phase**: processed by the numerical solver which stops when a zero-crossing event has been detected.
- **Discrete phase**: compute the consequence of (one or several) zero-crossing(s).
**Delta-delayed synchrony vs Instantaneous synchrony**

For cascaded zero-crossing, two interpretations of $\text{up}(e)$ lead to different results.

- **Delta-delay**: the effect of a zero-crossing is delayed by one instant.
  \[ T \text{ on } \text{up}(s) = \{ t^\bullet | (t \in T) \land (s(t) \leq 0) \land (s(t) > 0) \} \]

- **Instantaneous**: the effect is immediate.
  \[ T \text{ on } \text{up}(s) = \{ t | (t \in T) \land (s(t) \leq 0) \land (s(t) > 0) \} \]

We have considered the two solutions.

- The first one is simpler to compile. But the discrete state can last several micro-instants.
- The second one is (a little) more complicated to compile.

**Simultaneous events** A zero-crossing is a boolean signal; they are treated with a priority. Exactly what Simulink does.
Conclusion

A look at hybrid modelers from a programming language perspective with a focus on their semantics, typing and compilation.

Proposal
- A hybrid synchronous language with a Lucid Synchrone flavor.
- A semantics based on non-standard analysis so as to reason in a global, synchronous, manner.
- Divide with a type system, recycle a existing compiler to use a numerical solver as a black-box.
- A prototype implementation.

Extension
- Add timers (periodic clocks) as particular zero-crossing events.
- Combination of solvers, i.e., use different solvers at the same time.
Références


Ex 1: reset an integrator on a zero-crossing event

let hybrid main () =
  let rec der x = 1.0 init -1.0
      and der y = 0.0 init 1.0 reset -1.0 every up(x) in
  (x, y)
Ex 2: Unbounded instantaneous cascades of zero-crossing

let hybrid main () =
  let rec der x = 0.0 init -1.0
    reset -. 1.0 every up(y) | 1.0 every up(.- y) | 1.0 every up(z)
  and der y = 0.0 init -1.0
    reset 1.0 every up(x) | -1.0 every up(.- x)
  and der z = 1.0 init -1.0 in
(x,y,z)

\[ \partial \]
\[ [+1] \]
\[ [-1] \]
\[ x \]
\[ y \]
\[ 1+ \]
\[ \partial \]
\[ 2\partial \]
\[ 3\partial \]
\[ 4\partial \]
\[ 5\partial \]
\[ 6\partial \]

- \( \partial \) represent a “very small” step size in that finitely many \( \partial \)'s sum up to \( \approx 0 \).
- At \( t = 1 \), \( x \) and \( y \) starts an infinite cascade of zero-crossing while time remains blocked. **This is certainly pathological.**
Ex 3 : Sliding mode control

let hybrid main (y0) =
    let rec der x = 0.0 init -. sgn(y0) reset -1.0 every up(y)
        | 1.0 every up(-. y)
    and der y = x init y0 in
y
Explanation of Example 3

- Let \( y_0 < 0.0 \). \( y \) increases at constant speed until its first zero-crossing, just after \( t = |y_0| \).
- Then, \( y \) chatters infinitesimally around 0 as its speed alternate between \(-1\) and \(+1\) with infinitesimal step \( \partial \).

This example is not pathological It is equivalent to:

```ocaml
let hybrid main (y0) =
    let rec der y = z init y0
    and der x = 0.0 init -. sgn(y0) reset 0 every up(y) in y
```

Unbounded cascades of zero-crossings

- Time did not progress in example (2) while the very same zero-crossing condition has been taken twice.
- Is-it a run-time error? What about a causality analysis which accept programs (1) and (3) but reject program (2)?