Divide and recycle: types and compilation for a hybrid synchronous language

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

Paris, Synchronics days
Oct. 2010, 18th

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

• **Explicit** vs **Implicit** hybrid system modelers: Simulink, Scicos vs Modelica.

• In this talk, we consider only explicit ones.

• A lot of work on the formal verification of hybrid systems but relatively few on programming language aspects.

**Objective:**

• Extend a Lustre-like 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).

**Contribution:**

• **Divide** with a novel type system.

• **Recycle** existing tools, 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:

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

The first program can be written in any synchronous language, e.g. LUSTRE.

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

The second program can be written in any hybrid modeler, e.g. SIMULINK.

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

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

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.

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

or:

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

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; \textbf{time} on an 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 \(x\)!

If \(f\) is running on a continuous time basis, what would be the meaning of:

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

All these programs are \textbf{wrongly typed} and should be statically rejected. Simulink does it!
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*.

**Notation**

- $\text{up}(e)$ tests the zero-crossing of expression $e$ (from negative to positive).
- Handlers have priorities.
  
  \[
  z = 1 \text{ every } \text{up}(x) \mid 2 \text{ every } \text{up}(y) \text{ init } 0
  \]
- \(\text{last}(x)\) for the left-limit of signal \(x\).
  
  \[
  z = \text{last } z + 1 \text{ every } \text{up}(x) \mid \text{last } z - 1 \text{ every } \text{up}(y) \text{ init } 0
  \]
Examples

Combinatorial and sequential function (discrete time).

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 edge x = true -> pre x <> x

- add get type signature: int × int → int
- counter get type signature: bool × bool → int
- edge get type signature: ∀α.α → α
Connecting a discrete to continuous time

let hybrid counter_ten(top, tick) = o where

(* a periodic timer *)

der(time) = 1.0 /. 10.0 init 0.0 reset 0.0 every zero
and zero = up(time -. 1.0)

(* discrete function *)

and o = counter(top, tick) when zero init 0

The type signature is: \( \text{bool} \times \text{bool} \rightarrow \text{int} \).

Remark: provide ad-hoc programming constructs for periodic timers.
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: float × float × float \(\rightarrow\) float × float
The language kernel

- Synchronous (discrete) Lustre-like functions.
- Ordinary Differential Equations (ODE) with reset handlers

```
\[
\begin{align*}
  d &::= \text{let } k \ f(p) = e \mid d; d \\
  e &::= x \mid v \mid op(e) \mid e \ \text{fby} \ e \mid \text{last}(x) \\
  &\quad \mid \text{up}(e) \mid f(e) \mid (e,e) \mid \text{let } E \ \text{in} \ e \\
  p &::= (p,p) \mid x \\
  h &::= e \ \text{every} \ e \mid \ldots \mid e \ \text{every} \ e \\
  E &::= x = e \mid \text{der}(x) = e \ \text{init} \ e \ \text{reset} \ h \\
  &\quad \mid x = h \ \text{default} \ e \ \text{init} \ e \\
  &\quad \mid x = h \ \text{init} \ e \mid E \ \text{and} \ E
\end{align*}
\]
```
Typing

The type language

\[
\sigma ::= \forall \beta_1, ..., \beta_n. t \rightarrow t
\]
\[
t ::= t \times t \mid \beta \mid bt
\]
\[
k ::= D \mid C \mid A
\]
\[
bt ::= \text{float} \mid \text{int} \mid \text{bool} \mid \text{zero}
\]

We restrict to a first order language. Extension to higher-order later (but simple).

Initial conditions

\[
(+): \text{int} \times \text{int} \rightarrow^A \text{int}
\]
\[
(=): \forall \beta. \beta \times \beta \rightarrow^A \text{bool}
\]
\[
\text{if}: \forall \beta. \text{bool} \times \beta \times \beta \rightarrow^A \beta
\]
\[
\text{pre}(.): \forall \beta. \beta \rightarrow^D \beta
\]
\[
\text{fby}.: \forall \beta. \beta \times \beta \rightarrow^D \beta
\]
\[
\text{up}(.): \text{float} \rightarrow^C \text{zero}
\]
The Type system

Global and local environment

\[ G ::= [f_1 : \sigma_1; \ldots; f_n : \sigma_n] \hspace{1cm} H ::= [\ ] \mid H, x : t \mid H, \text{last}(x) : t \]

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

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

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

(EQ)
\[
G, H \vdash_k e : t
\]
\[
G, H \vdash_k x = e : [x : t]
\]

(APP)
\[
t \stackrel{k}{\rightarrow} t' \in \text{Inst}(G(f)) \quad G, H \vdash_k e : t
\]
\[
G, H \vdash_k f(e) : t'
\]
A sketch of the semantics

The base clock: \( \partial \) infinitesimal, the set

\[
BaseClock(\partial) = \{ n\partial \mid n \in \mathbb{N} \}
\]

is isomorphic to \( \mathbb{N} \) as a total order.

For \( t = t_n = n\partial \in BaseClock(\partial) \), \( \dot{t} = t_{n-1} \) and \( \ddot{t} = t_{n+1} \).

Clock and signals A clock \( T \) is a subset of \( BaseClock(\partial) \). A signal \( s \) is a total function \( s : T \rightarrow V \).

If \( T \) is a clock and \( b \) a signal \( b : T \rightarrow \mathbb{B} \), then \( T \) 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 \( s : T \rightarrow \mathbb{R} \), we write \( T \) on \( \text{up}(s) \) for the instants when \( s \) crosses zero, that is:

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

The effect of \( \text{up}(e) \) is delayed by one cycle.
Discrete vs Continuous

Let \( x \) be a signal with clock domain \( T_x \), it is typed \emph{discrete} (\( \text{D}(T) \)) either if it has been so declared, or if its clock is the result of a zero-crossing or a sub-clock of a discrete clock. Otherwise it is typed \emph{continuous} (\( \text{C}(T) \)). That is:

1. \( \text{C}(\text{BaseClock}(\partial)) \)
2. If \( \text{C}(T) \) and \( s : T \mapsto \mathbb{R} \) then \( \text{D}(T \text{ on up}(s)) \)
3. If \( \text{D}(T) \) and \( s : T \mapsto \mathbb{B} \) then \( \text{D}(T \text{ on } s) \)
4. If \( \text{C}(T) \) and \( s : T \mapsto \mathbb{B} \) then \( \text{C}(T \text{ on } s) \)

**Correction of the type system:**

When an is typed \( \text{D} \) (resp. \( \text{C} \)), it is indeed activated on a discrete (resp. continuous) clock.
\[
\text{integr}^\#(T)(s)(s_0)(hs)(t) = s'(t) \quad \text{where}
\]
\[
s'(t) = s_0(t) \quad \text{if} \ t = \min(T)
\]
\[
s'(t) = s'(\cdot t) + \partial s(\cdot t) \quad \text{if} \ \text{handler}^\#(T)(hs)(t) = \text{NoEvent}
\]
\[
s'(t) = v \quad \text{if} \ \text{handler}^\#(T)(hs)(t) = \text{Xcrossing}(v)
\]

\[
\text{up}^\#(T)(s)(t) = \text{false} \quad \text{if} \ t = \min(T)
\]
\[
\text{up}^\#(T)(s)(t^\bullet) = \text{true} \quad \text{if} \ (s(\cdot t) \leq 0) \land (s(t) > 0) \ \text{and} \ (t \in T)
\]
\[
\text{up}^\#(T)(s)(t^\bullet) = \text{false} \quad \text{otherwise}
\]
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 the discrete subset. Compile the result with an existing synchronous compiler such that it verifies the following invariant:

The discrete state, i.e., the values of delays, will not change if all of the zero-crossing conditions are false.
Example (counter)

Add extra input and outputs.

- \( \text{up}(e) \) becomes a fresh boolean input \( z \) and generate an equation \( \text{up}_z = e \).
- \( \text{der}(x) = e \text{ init } e_0 \) becomes \( dx = e \text{ init } e_0 \).
- A continuous state variable becomes an input.

```plaintext
let node counter_ten([z], [time], (top, tick)) =
  (o, [upz], [dtime])

where
  dtime = 1.0 / 10.0 init 0.0 reset 0.0 every z
and o = counter(top, tick) when z init 0
and upz = time -. 1.0
```

In practice, represent these extra inputs with arrays.

Now, ignoring details of syntax, the function `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 library. An Ocaml interface has been developed.

**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 \( up(e) \) lead to different results.

- **Delta-delay**: the effect of a zero-crossing is delayed by one instant.

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

- **Instantaneous**: the effect is immediate.

\[
T \text{ on } up(s) = \{ t | (t \in T) \land (s(t) \leq 0) \land (s(t) > 0) \}
\]

We have considered the first solution.

- Simple to compile. But the discrete state can last several instants.

- The second one is (a little) more complicated to compile. But all zero-crossing can be statically scheduled. Only one instant in the discrete state.

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

Proposal

• To mix signals on discrete time and signal on continuous time.
• A Lustre-like proposal to combine stream equations with ODE.
• Divide with a type-system, recycle a existing compiler to use a numerical solver as a black-box.

Extension

• (Hybrid) hierarchical automata can be translated into the basic language
• Implementation in a real language
References


