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

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

Paris, Synchronics days
Oct. 2010, 18th

[a] 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 -> pre f + s and s = 0.2 * (x - pre f)
```

and the initial value problem:

```
der(y') = -9.81 init 0.0 and der(y) = y' init 10.0
```

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

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

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.

```
der(time) = 1.0 init 0.0 and x = 0.0 fby x + time
```

or:

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

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

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 an continuous, absolute time.

Equations with reset

Two independent groups of equations.

```
  der(p) = 1.0 init 0.0 reset 0.0 every up(p - 1.0)
and
  x = 0.0 fby x + p
and
  der(time) = 1.0 init 0.0
and
  z = up(sin (freq * time))
```

Properly translated in Simulink, changing `freq` changes the output of `x`!

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

```
  y = f(x) every up(z) init 0
```

All these programs are **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

- `up`(e) tests the zero-crossing of expression e (from negative to positive).
- Handlers have priorities.

```
z = 1 every up(x) | 2 every up(y) init 0
```

- `last`(x) for the left-limit of signal x.

```
z = last z + 1 every up(x) | last z - 1 every up(y) 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: $\texttt{int} \times \texttt{int} \xrightarrow{\texttt{A}} \texttt{int}$
- `counter` get type signature: $\texttt{bool} \times \texttt{bool} \xrightarrow{\texttt{D}} \texttt{int}$
- `edge` get type signature: $\forall \alpha . \alpha \xrightarrow{\texttt{D}} \alpha$

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: $\texttt{bool} \times \texttt{bool} \xrightarrow{\texttt{C}} \texttt{int}$.

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: $\texttt{float} \times \texttt{float} \times \texttt{float} \xrightarrow{\texttt{C}} \texttt{float} \times \texttt{float}$

The language kernel

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

$$
\begin{array}{rcl}
d & ::= & \texttt{let}\ k\ f(p) = e \mid d; d \\
e & ::= & x \mid v \mid op(e) \mid e\,\texttt{fby}\,e \mid \texttt{last}(x) \\
  &   & \mid \texttt{up}(e) \mid f(e) \mid (e, e) \mid \texttt{let}\ E\ \texttt{in}\ e \\
p & ::= & (p, p) \mid x \\
h & ::= & e\ \texttt{every}\ e \parallel \ldots \parallel e\ \texttt{every}\ e \\
E & ::= & x = e \mid \texttt{der}(x) = e\ \texttt{init}\ e\ \texttt{reset}\ h \\
  &   & \mid x = h\ \texttt{default}\ e\ \texttt{init}\ e \\
  &   & \mid x = h\ \texttt{init}\ e \mid E\ \texttt{and}\ E
\end{array}
$$

Typing

The type language

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

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

Initial conditions

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

The Type system

Global and local environment

$$G ::= [f_1 : \sigma_1; ...; f_n : \sigma_n] \qquad H ::= [\,] \mid H, x : t \mid H, \texttt{last}(x) : t$$

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:

$$\forall k, \texttt{A} \leq k$$

A sketch of Typing rules

(DER)

$$\frac{G, H \vdash_{\mathtt{C}} e_1 : \mathtt{float} \qquad G, H \vdash_{\mathtt{C}} e_2 : \mathtt{float} \qquad G, H \vdash h : \mathtt{float}}{G, H \vdash_{\mathtt{C}} \mathtt{der}(x) \ = \ e_1 \ \mathtt{init} \ e_2 \ \mathtt{reset} \ h : [\mathtt{last}(x) : \mathtt{float}]}$$

(AND)

$$\frac{G, H \vdash_k E_1 : H_1 \qquad G, H \vdash_k E_2 : H_2}{G, H \vdash_k E_1 \ \mathtt{and} \ E_2 : H_1 + H_2}$$

(EQ)

$$\frac{G, H \vdash_k e : t}{G, H \vdash_k x \ = \ e : [x : t]}$$

(APP)

$$\frac{t \xrightarrow{k} t' \in \mathit{Inst}(G(f)) \qquad G, H \vdash_k e : t}{G, H \vdash_k f(e) : t'}$$

A sketch of the semantics

The base clock: ∂ infinitesimal, the set

$$BaseClock(\partial) = \{n\partial \mid n \in {}^\star\mathbb{N}\}$$

is isomorphic to ${}^\star\mathbb{N}$ as a total order.

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

Clock and signals A *clock* T is a subset of $BaseClock(\partial)$. 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 `on` b defines a subset of T comprising those instants where $b(t)$ is true:

$$T \texttt{ on } b = \{t \mid (t \in T) \wedge (b(t) = \texttt{true})\}$$

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

$$T \texttt{ on up}(s) = \{t^\bullet \mid (t \in T) \wedge (s({}^\bullet t) \leq 0) \wedge (s(t) > 0)\}$$

The effect of `up`(e) is delayed by one cycle.

Discrete *vs* Continuous

Let x be a signal with clock domain T_x, it is typed *discrete* ($\mathtt{D}(T)$) 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 *continuous* ($\mathtt{C}(T)$). That is:

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

Correction of the type system:

When an is typed D (resp. C), it is indeed activated on a discrete (resp. continuous) clock.

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

Compilation

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

- `up`(e) becomes a fresh boolean input z and generate an equation $up_z = e$.
- `der`$(x) = e$ `init` e_0 becomes $dx = e$ `init` e_0.
- A continuous state variable becomes an input.

```
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.

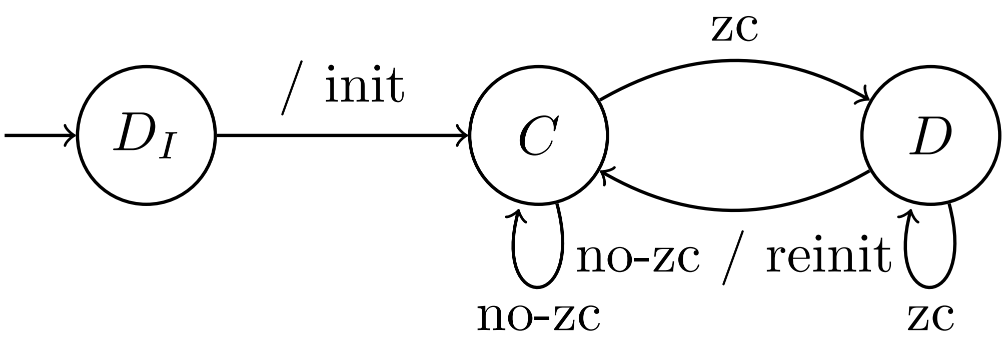
Now, ignoring details of syntax, the function `counter_ten` can be processed by any synchronous compiler, and the generated transition function verifies the invariant.

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 $\texttt{up}(e)$ lead to different results.

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

$$T \;\texttt{on}\; \texttt{up}(s) = \{t^{\bullet} \mid (t \in T) \wedge (s({}^{\bullet}t) \leq 0) \wedge (s(t) > 0)\}$$

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

$$T \;\texttt{on}\; \texttt{up}(s) = \{t \mid (t \in T) \wedge (s({}^{\bullet}t) \leq 0) \wedge (s(t) > 0)\}$$

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

[1] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Non-standard semantics of hybrid systems: ODE. Submitted for publication, October 2010.

[2] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'11)*, Chicago, USA, April 2011.

[3] Albert Benveniste, Benoit Caillaud, and Marc Pouzet. Non-standard semantics of hybrid systems: DAE. Submitted for publication, October 2010.

[4] Albert Benveniste, Benoit Caillaud, and Marc Pouzet. The Fundamentals of Hybrid Systems Modelers. In *49th IEEE International Conference on Decision and Control (CDC)*, Atlanta, Georgia, USA, December 15-17 2010.