From Lustre to Lucid Synchrone[a]

Marc Pouzet
UPMC/ENS/INRIA
Team PARKAS
Marc.Pouzet@ens.fr

ANR Cafein, Kick-off meeting
Toulouse, Feb. 6, 2013

[a] This talk is dedicated to the memory of Paul Caspi and all the fruitful ideas he gave to people during his carreer.

Synchronous Languages

Domain specific languages for programming real-time control software.

Based on the **synchronous model of time**:

- Consider *a priori* that computations take no time;
- Check *a posteriori* that the implementation is fast enough (WCET);

From the programmer point-of-view, **time is logical**:

- simpler, code is more portable;
- still, some important aspects are discarded.

The philosophy of these languages was to **statically reject programs** in order to ensure safety properties:

- precise semantics, preservation of determinism;
- compilation to statically schedulable code running in bounded time and space.

This departs from languages like Simulink/Stateflow where almost any program has a behavior. Lustre (Caspi, Halbwachs, Raymond) was one of these languages.

A typical Lustre program:

```
node counter(x: bool; res: bool) returns (o: int)
var v: int;
let
  v = if x then 1 else 0;
  o = if res then v else v -> pre o + v;
tel
```

Given to the compiler (e.g., the one from VERIMAG (Raymond) or PARKAS (Gerard, Guatto, Pasteur, Pouzet, [LCTES'08]), it generates three elements:

- a data-structure to store the internal state;
- a step function `counter_step`; a reset function `counter_reset`

Pros/Cons:

- safe C: no recursion, no loop, no pointer arithmetic, static memory allocation.
- some constructs are ugly (`current`); clocks are almost impossible to use;
- programming control-dominated systems is painful.

From Lustre to Lucid Synchrone

The early days (1995)...

With Paul Caspi (VERIMAG, Grenoble), we worked on the semantics and implementation of a **Lustre** with functions.

- Lustre can be embedded in Haskell in a few lines...
- Synchrony is related to **deforestation** in lazy functional languages: synchronous programs are those which can be computed without intermediate lists.
- Generalise synchronous techniques (clock calculus, compilation) to a more expressive language. Make it **conservative**: more features but with the same safety properties.
- We called these programs **Synchronous Kahn Networks** [ICFP'96] as process networks which can be executed with statically bounded FIFOs.

Lucid Synchrone (1996)

We started working a first prototype, with some new features:

- higher-order functions; type inference, clocks as types; modular compilation with no *a priori* maximal inlining.

Extending the language (1996 — 2006)

Several versions done (a least ten complete rewriting). Three major versions.

- V1 (1996): functions, data-flow equations, type, clock inference.
- V2 (2000, with Grégoire Hamon): V1 + type-based causality analysis + ML-like clock calculus.
- V3 (2005): V2 + hierarchical automata + Esterel-like signals, streams of stream functions.

Next? Continuous-time!

Mix of discrete-time and continuous-time. A current prototype (`Zélus`):

- data-flow equations; hierarchical automata + ODEs;
- runtime with (multi-step, multi-order, variable step) black-box solvers (SUNDIALS CVODE).

Milestones

In 2000, start of the collaboration with Jean-Louis Colaço (Esterel-Tech.) on the design and implementation of a new compiler for SCADE and a new language. Bruno Pagano (Esterel-Tech.) joined us in 2005.

- Clock calculus a la ML [EMSOFT'03]
- Initialization analysis [SLAP'03, STTT'04]
- Higher-order and typing [EMSOFT'04]
- Mixing data-flow and state machines [EMSOFT'05, EMSOFT'06]
- Clock-directed code generation [LCTES'08]
- Objects and modular design of systems with modes [LCTES'09]

Research hand-in-hand; ideas prototyped both in Lucid Synchrone and in ReLuC (Jean-Louis Colaço, Bruno Pagano), at Esterel-Tech.

The new langage SCADE 6 and his compiler, distributed since 2008 is based on these fundational works. The code generator is DO178B compliant. SCADE 6 is used in several critical softwares.

A brief look at Lucid Synchrone

Avoid giving types.

```
let half_add(a,b) = (s, co) where
  rec s = a xor b and co = a & b

let full_add(a,b,c) = (s, co) where
  rec (s1, c1) = half_add(a,b)
  and (s, c2) = half_add(c, s1)
  and co = c1 or c2
```

Polymorphism, higher-order functions.

```
let node delay x = x fby x

let node double_delay x = x fby x fby x

let node edge x = false -> x <> pre x

let node edge compare x = false -> compare x (pre x)
```

and we get:

```
val delay : 'a => 'a

val delay :: 'a -> 'a


val edge : 'a => 'a

val edge :: 'a -> 'a


val edge : ('a -> 'a -> bool) -> 'a => bool

val edge :: ('a -> 'a -> 'a) -> 'a -> 'a
```

In Lustre, polymorhism is restricted to the primitives (`if/then/else`).

Clocks (mix slow and fast processes)

Two primitives

`when` (sub-sampling), `merge` (over-sampling)

`c`	t	t	f	f	t	f	$\ldots$
`x`	x_0	x_1	x_2	x_3	x_4	x_5	$\ldots$
`x when c`	x_0	x_1			x_4		$\ldots$
`x whenot c`			x_2	x_3		x_5	$\ldots$
`y`	y_0	y_1			y_2		$\ldots$
`merge c y (x whenot c)`	y_0	y_1	x_2	x_3	y_2	x_5	$\ldots$

Examples

```
let zero_holder(x0, c, x) = m where
  rec m = merge c x ((x0 fby m) whenot c)

let node sum x = s where rec s = x -> pre s + x
let node sampled_sum x c = sum (x when c)

val sampled_sum : int -> bool => int
val sampled_sum :: 'a -> (_c0:'a) -> 'a on _c0

let clock ten = count 10 true
let node sum_ten x = sampled_sum x ten
```

An we get:

```
val zero_holder : 'a -> clock -> 'a => 'a
val zero_holder : 'a -> (c: 'a) -> 'a on c -> 'a


val ten : clock
val ten :: 'a


val sum_ten : int => int
val sum_ten :: 'a -> 'a on ten
```

Clocks:

- Typed-based clock inference [ICFP'96, EMSOFT'03]; lighter to program with;
- clocks are used during code generation (a clock = a guard).
- clocks are useful for identifying a minimal language kernel with simple semantics.
- Yet, this is not a programming construct for casual users...

Control structures, hierarchical automata

Clocks are a mean to define control structures. Yet, can we provide a simpler programming construct?

The Franc/Euro converter (Jean-Louis Colaço)

```
let node converter v c = (euro, fr) where
  automaton
  | Franc -> do fr = v and eur = v / 6.55957 until c then Euro
  | Euro -> do fr = v * 6.55957 and eu = v until c then Franc
  end


let node converter v c = (euro, fr) where
  automaton
    Franc -> do fr = v and eur = v / 6.55957 unless c then Euro
  | Euro -> do fr = v * 6.55957 and eu = v unless c then Franc
  end
```

Two kinds of transitions: weak (`until`) and strong (`unless`). A simple "finger" semantics: essentially one transition per instant (no cascade of strong transitions).

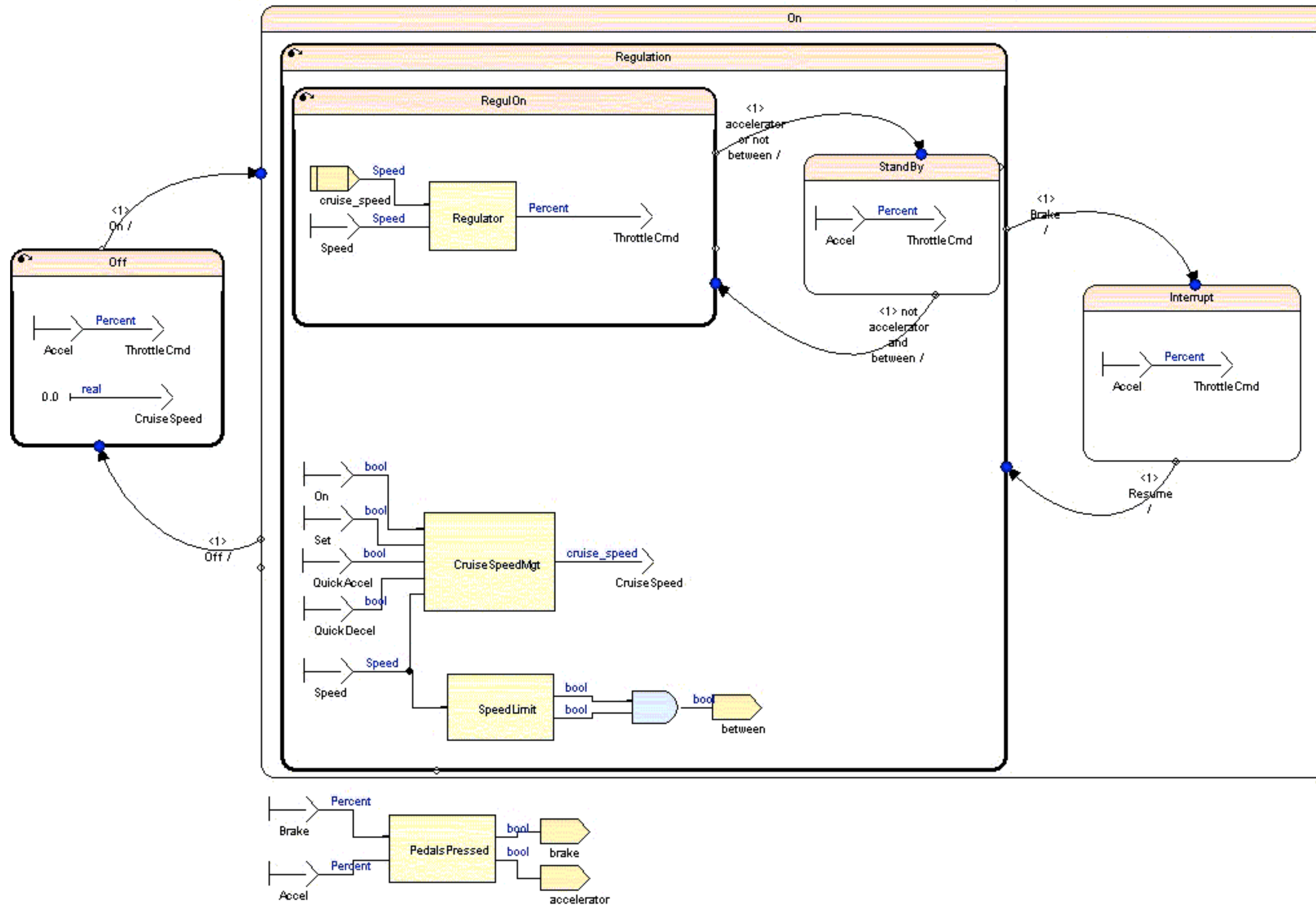
Automata with communication

A programming construct to communicate values between states. A classical problem in block-diagram languages (SCADE, Simulink/Stateflow, LabView, Modelica).

```
let node adjust i p m = o where
  automaton
  | Await -> do o = i then Up
  | Counting ->
      do automaton
          | Up -> do o = last o + 1 unless down then Down
          | Down -> do o = last o - 1 unless up then Up
         end
      unless up & down then Silent
  | Silent -> do then Up
  end
```

`last x` is the last defined value of `x`, whereas `last e` is the last observed value of `e`. Complement implicitely every state with `x = last x` when `x` is undefined.

Example: the cruise control in SCADE 6



A word on compilation

Compilation is a sequence of source-to-source transformations. In particular, automata are translated into more elementary control-structures which are in turn translated into `merge/when` constructs.

A n-ary merge is used (that filters according to a value from an enumerated type). E.g.,:

```
let node converter v c = (euro, fr) where
  match Franc fby state with
  | Franc -> do fr = v and euro = v / 6.55957
             and state = if c then Euro else Franc
  | Euro -> do fr = v * 6.55957 and euro = v
            and state = if c then Franc else Euro
  end
```

Then:

```
let node converter v c = (euro, fr) where
  rec fr = merge active
                (v when Franc(active)) ((v whenot Euro(active)) * 6.55957)
  and euro = merge active
                  ((v when Franc(active))/ 6.55957) (v whenot Euro(active))
  and state = merge active
               (if (c when Franc(active)) then Euro else Franc)
               (if (c when Euro(active)) then Franc else Euro)
  and active = Franc fby state
```

`v when Franc(active)` is present when `active = Franc`.

Now, we are home and dry; the resulting program can be translated to efficient C code by the existing compiler.

The compiler is a sequence of source-to-source rewritting, each pass eliminating a programming construct.

Some questions for us in the CAFEIN project

Currently, verification tools do not take clocks, nor control structures into account. Clocked data-flow is translated into pure data-flow.

The proof of simple properties for programs with several modes surprisingly fails. E.g., the following program (`heptagon` syntax, see [LCTES'12])):

```
const n:int = 2000
node count () = (last x:int = 0)
let automaton
      state Up
        do x = last x + 1
        until x = 45 then Down
      state Down
        do x = last x - 1
        until x = -45 then Up
      end
tel
```

Given to Frama-C, it fails in proving that $-45 \leq x \leq +45$. Yet, the Prover tool proves the property using k-induction.

Questions:

- What has been lost durin the translation so that the proof becomes harder?
- Is-it enough to reinforce invariants or add properties to the translated program so that properties become k-inductive?
- Exploit language properties; methodology: can we exploit some assume/guaranty inserted in the program (typically in every state of an automaton)?

References

[1] Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

[2] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Pensylvania, May 1996.

[3] Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998. Extended version available as a VERIMAG tech. report no. 97–07 at `www.di.ens.fr/∼pouzet/bib/bib.html`.

[4] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[5] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

[6] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):245–255, August 2004.

[7] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. A Modular Memory Optimization for Synchronous Data-Flow Languages. Application to Arrays in a Lustre Compiler. In *Languages, Compilers and Tools for Embedded Systems (LCTES'12)*, Beijing, June 12-13 2012. ACM. Best paper award.