A conservative extension of ECLAT with ESTEREL and LUSTRE constructs (WIP)

Loïc Sylvestre[1]

[1]Université de Toulouse - IRIT (TRACES team)

November 26 2025 - SYNCHRON '25 à Aussois

Context

- The ECLAT programming language
  - functional programmation language (*à la* OCAML) with synchronous semantics
  - compiled to VHDL for programming FPGAs.
- ECLAT initial design choice: keep it simple
  - avoid causality/dependency issues
  - ECLAT programs are **acyclic by construction**
  - and **communications between parallel "threads" are delayed** (via built-in shared memory accesses with deterministic concurrency)
  - **but this is quite restrictive**...
- A long-term wish: embedding LUSTRE in ECLAT, to keep it simple but more expressive.
- Ongoing work, an exploratory path:
  - extend ECLAT with instantaneous signal broadcasting (and while you are at it, extend ECLAT with a complete ESTEREL kernel)
  - let programmers define their own fixpoint operator and other derivated constructs, in ECLAT, as polymorphic higher-order functions.

Comparing Eclat, Esterel and Lustre

If **not(A)** then **B** and **C** (ECLAT *is closer to* LUSTRE)

ESTEREL

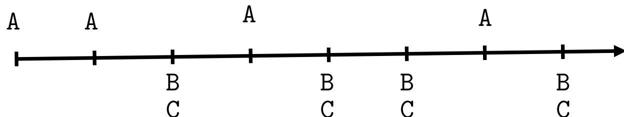
```
module NOTABC:
  input A;
  output B, C;
  loop
    present A else emit B; emit C end;
    pause
  end
end
```

ECLAT

```
let notabc (a:bool) : bool*bool =
    let x = not(a) in
    (x,x) ;;
```

```
node notabc (a:bool)
      returns (b,c:bool);
let
  b = c;
  c = not(a);
tel
```

LUSTRE



a	T	T	F	T	F	F	T	F	...
b	F	F	T	F	T	T	F	T	...
c	F	F	T	F	T	T	F	T	...

Wait $N > 0$ instants and terminate (ECLAT *is closer to* ESTEREL)

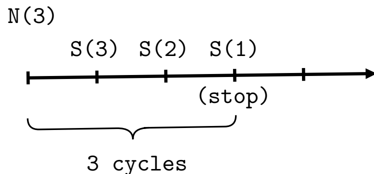
ESTEREL

```
module WAIT_N;
  input N : integer;
  signal S : integer in
  trap T in
    pause; emit S(pre(?N));
    loop
      if ?S = 1 then exit T
      else
        pause; emit S(pre(?S)-1)
      end
    end
  end
end
```

ECLAT

```
let rec wait_n (n:int) : unit =
  if n = 1 then () else wait_n(n-1) ;;
```

NB: a pause per direct or tail-recursive call to `wait_n`



Wait $N > 0$ instants and terminate (ECLAT *is closer to* ESTEREL)

ESTEREL

```
module WAIT_N;
  input N : integer;
  signal S : integer in
  trap T in
    pause; emit S(pre(?N));
    loop
      if ?S = 1 then exit T
      else
        pause; emit S(pre(?S)-1)
      end
    end
  end
end
```

ECLAT

```
let rec wait_n (n:int) : unit =
  if n = 1 then () else wait_n(n-1) ;;
```

NB: a pause per direct or tail-recursive call to `wait_n`

```
node wait_n (rst:bool ; n:int)
  returns (rdy:bool);
var s : int;
let
    s  = if (false fby rst) then (0 fby n)
         else
           (0 fby (if s = 1 then s else s-1));
    rdy = (not rst) and (s = 1);
tel
```

LUSTRE

Entry point of the reactive program (ECLAT *is closer to* LUSTRE)

ESTEREL

```
module MAIN;
  input A; N : integer;
  output B, C, RDY;
  [
    run NOTABC
  ||
    loop
      run WAIT_N;
      emit RDY;
      pause
    end
  ]
end
```

ECLAT

```
let main ((n,a):int*bool) : bool*bool*bool =
  let (b,c) = notabc(a) in
  let ((),rdy) = exec wait_n(n) default () in
  (b,c,rdy) ;;
```

```
node main (a:bool ; n:int)
   returns (b,c,rdy:bool);
var rst : bool;
let
  (b,c) = notabc(a);
  rdy = wait_n(rst,n);
  rst = true fby rdy;
tel
```

LUSTRE

ECLAT syntax and semantics

ECLAT core language

program	π	$::=$ **fun** $p \rightarrow e$
pattern	p	$::= () \mid x \mid (p, p)$
expression	e	$::=$
simple expressions		$x \mid c \mid \phi(e) \mid (e, e)$
sequential composition		$\mid$ **let** $p = e$ **in** $e \mid e;\ e$
parallel composition		$\mid (e \,\|\|\, e)$
conditional		$\mid$ **if** e **then** e **else** e
local function definition		$\mid$ **fun** $p \rightarrow e$
function application		$\mid f\ e$
register		$\mid$ **reg** (**fun** $p \rightarrow e$) **init** e
step-by-step execution		$\mid$ **exec** e **default** e_0
pause and tail-recursion		$\mid$ **pause** $\mid$ **let rec** $f\ p = e$ **in** e
constant	c	$::= () \mid$ `true` $\mid$ `false` $\mid n$
primitive	ϕ	$::= + \mid - \mid < \mid \cdots$

Synchronous semantics

program	π	$::=$	**fun** $p \to e$
pattern	p	$\in$	$\mathcal{P}$
expression	e	$\in$	$\mathcal{E}$
value	v	$\in$	$\mathcal{V} \subseteq \mathcal{E}$
stream	ω	$::=$	$v \cdot \omega$
memory state	μ	$\in$	Σ

$$\frac{\pi(v)/\mu \longrightarrow^{\star} v'/\mu' \qquad \mu' \vdash \pi(\omega) \Downarrow \omega'}{\mu \vdash \underbrace{(\textbf{fun } p \to e)}_{\pi}(v \cdot \omega) \Downarrow (v' \cdot \omega')}$$

- $\mu \vdash \pi(\omega) \Downarrow \omega'$: given memory μ, program π applied to stream ω computes ω'
  - length-preserving function on streams (one input per instant)
  - defined as an instantaneous step function applied at each instant: gets the current input v and returns the current output v' by a finite sequence of reductions (microsteps):

$$\pi(v)/\mu \longrightarrow e_1/\mu_1 \longrightarrow \cdots v'/\mu'$$

Reduction semantics, by substitution, with call-by-value evaluation contexts (extracts)

$$\textbf{evaluation context} \quad F \quad ::= \quad \textbf{let } p = \square \textbf{ in } e \mid \square;\ e \mid \textbf{exec}_\ell\ e \textbf{ default } \square \mid \cdots$$

Context

$$\frac{e/\mu \longrightarrow e'/\mu'}{F[e]/\mu \longrightarrow F[e']/\mu'}$$

Let

$$\textbf{let } p = v \textbf{ in } e/\mu \longrightarrow e[p \mapsto v]/\mu$$

Seq

$$();e/\mu \longrightarrow e/\mu$$

Pause

$$F[\textbf{pause}]/\mu \longrightarrow \textbf{pause};\ F[()]/\mu$$

Par

$$(\textbf{pause } e \,||\, \textbf{pause } e')/\mu \longrightarrow \textbf{pause } (e \,||\, e')/\mu$$

NB: each time a pause occurs, it is propagated and finally removed by an outer **exec** block at the end of the instant (if the program is well-typed). The continuation of the computation is saved as a thunk ($\textbf{fun } () \to e$) in the memory μ.

Static typing

- abstracting the execution time (D):
  - always instantaneous ($\mathbf{0}$),
  - 0 or more instants ($\mathbf{1}$).
- distinguish basic types $\mathring{\tau}$ (`unit`, `bool`, `int`, pair of basic types) *versus* types τ (including functional types)
- functional type of the form $\tau \xrightarrow{D} \mathring{\tau}'$
  - limited form of higher-orderness (the return type is a basic type)
  - inferred duration D is used to ensure reactivity, for example:

$$\textsc{Prog}\quad \frac{\varnothing[p : \mathring{\tau}] \vdash e : \mathring{\tau}' | \mathbf{0}}{\vdash (\textbf{fun } p \rightarrow e) : \mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}'}$$

$$\textsc{Pause}\quad \Gamma \vdash \textbf{pause} : \texttt{unit} | \mathbf{1}$$

$$\textsc{Exec}\quad \frac{\Gamma \vdash e : \mathring{\tau} | D \qquad \Gamma \vdash e_0 : \mathring{\tau} | \mathbf{0}}{\Gamma \vdash \textbf{exec } e \textbf{ default } e_0 : \mathring{\tau} \times \texttt{bool} | \mathbf{0}}$$

A conservative extension of ECLAT with ESTEREL constructs

Add the missing constructs

ECLAT

expression	e ::=	⋯
nothing	|	`()` (constant of type unit)
pause	|	`pause`
sequence	|	e_1; e_2
parallel pair	|	(e_1 || e_2)
conditional	|	`if` e_1 `then` e_2 `else` e_3

syntactic sugar: [e_1 || e_2]
stands for (`let` ((),()) = (e_1|| e_2)
`in` ())

conservative extension to support the whole ESTEREL kernel

emission	|	`emit` $x(e)$
reception	|	?x
local signal	|	`signal` x `in` e
loop	|	`loop` e `end`
trap	|	`trap` x `in` e
exit	|	`exit` x
suspend	|	`suspend` e `when` x

Valued signals

expression	e	$::=$	$\cdots \mid \textbf{signal}\ x\ \textbf{in}\ e \mid \textbf{emit}\ x(e) \mid ?S$
evaluation context	F	$::=$	$\cdots \mid \textbf{emit}\ s(\square)$
type	τ	$::=$	$\cdots \mid \texttt{signal<}\mathring{\tau}\texttt{>}$
basic type	$\mathring{\tau}$	$::=$	$\cdots$

$$\frac{\Gamma[x : \texttt{signal<}\mathring{\tau}_x\texttt{>}] \vdash e : \mathring{\tau} | D}{\Gamma \vdash \textbf{signal}\ x\ \textbf{in}\ e : \mathring{\tau} | D}$$

$$\frac{\Gamma(x) = \texttt{signal<}\mathring{\tau}_x\texttt{>} \qquad \Gamma \vdash (e) : \mathring{\tau}_x | D}{\Gamma \vdash \textbf{emit}\ x(e) : \texttt{unit} | D} \qquad \frac{\Gamma(x) = \texttt{signal<}\mathring{\tau}\texttt{>}}{\Gamma \vdash ?x : \tau | \mathbf{0}}$$

syntactic sugar: $\boxed{\textbf{emit}\ s \equiv \textbf{emit}\ s(\texttt{true})}$ e.g. $(\textbf{emit}\ A; ?A)$

Valued signals

expression	e	$::=$	$\cdots \mid \textbf{signal}\ x\ \textbf{in}\ e \mid \textbf{emit}\ x(e) \mid ?S$
evaluation context	F	$::=$	$\cdots \mid \textbf{emit}\ s(\square)$
type	τ	$::=$	$\cdots \mid$ `signal<`$\mathring{\tau}$`>`
basic type	$\mathring{\tau}$	$::=$	$\cdots$

$e/\mu \xrightarrow{\mathbf{S}} e'/\mu'$: in the environment of signals $\mathbf{S}$, the configuration e/μ reduces to e'/μ'.

$$\textbf{signal}\ x\ \textbf{in}\ v/\mu \xrightarrow{\mathbf{S}} v/\mu$$

(not executable: need to guess v_x)

$$\frac{x \notin \mathrm{dom}(\mathbf{S}) \qquad e/\mu \xrightarrow{\mathbf{S}[x \mapsto v_x]} e'/\mu'}{\textbf{signal}\ x\ \textbf{in}\ e/\mu \xrightarrow{\mathbf{S}} \textbf{signal}\ x\ \textbf{in}\ e'/\mu'}$$

$$\frac{\mathbf{S}(x) = v}{\textbf{emit}\ x(v)/\mu \xrightarrow{\mathbf{S}} ()/\mu} \qquad \frac{\mathbf{S}(x) = v}{?x/\mu \xrightarrow{\mathbf{S}} v/\mu}$$

No signal status: values are encoded as bitvectors, each bit is absent (0) if not emitted

trap/exit, **suspend** and **loop**

$$
\begin{array}{llll}
\textbf{expression} & e & ::= & \cdots \mid \textbf{trap}\ x\ \textbf{in}\ e \mid \textbf{exit}\ x \mid \textbf{suspend}\ e\ \textbf{when}\ x \mid \textbf{loop}\ e\ \textbf{end} \\
\textbf{type} & \tau & ::= & \cdots \mid \texttt{trap} \\
\textbf{basic type} & \mathring{\tau} & ::= & \cdots
\end{array}
$$

- Add typing rules:

$$
\frac{\Gamma[x : \texttt{trap}] \vdash e : \texttt{unit} | D}{\Gamma \vdash \textbf{trap}\ x\ \textbf{in}\ e : \texttt{unit} | D}
\qquad
\frac{\Gamma(x) = \texttt{trap}}{\Gamma \vdash \textbf{exit}\ x : \texttt{unit} | \mathbf{0}}
$$

$$
\frac{\Gamma \vdash e : \mathring{\tau} | D \qquad \Gamma(x) = \texttt{signal<bool>}}{\Gamma \vdash \textbf{suspend}\ e\ \textbf{when}\ x : \mathring{\tau} | D}
\qquad
\frac{\Gamma \vdash e : \tau | \mathbf{1}}{\Gamma \vdash \textbf{loop}\ e\ \textbf{end} : \texttt{unit} | \mathbf{1}}
$$

- and reduction rules, e.g.:

$$
\textbf{loop}\ e\ \textbf{end}/\mu \xrightarrow{\mathsf{S}} e;\ \textbf{loop}\ e\ \textbf{end}
$$

Defining ESTEREL derivated constructs and LUSTRE operators

Define ESTEREL derivated constructs in ECLAT

```
1  let halt() = loop pause end ;;
2  let sustain s = loop emit s; pause end ;;
3  let await s = trap T in loop pause; if ?s then exit T else () end ;;
4
5  let abort (f, s) =
6    trap T in [(suspend f() when s; exit T) || (await s; exit T )] ;;
7
8  let loop_each (f, s) = loop abort ((fun () -> (f(); halt())), s) end ;;
9
10 let abro (a, b, r, o) =
11   loop_each((fun () -> [await a || await b]; emit o), r) ;;
12
13 let abcro(a, b, c, r, o) =
14   signal t in [ abro(a, b, r, t) || abro(t, c, r, o) ] ;;
```

Define LUSTRE operators in ECLAT

```
1  let absent () = signal s in ?s ;;
2  let mux(a, b, c) = if a then b else c ;;
3
4  let fby (x, y) =
5    let (o, _) = reg (fun (_, prey) -> (prey, y)) init (absent(), x)
6    in o ;;
7
8  let when(f, clk) = if clk then f() else absent() ;;
9  let merge(clk, a, b) = if clk then a else b ;;
10
11 let fixpoint (f) =
12   signal s in emit s(f(?s)); ?s ;;
13
14 (* for example *)
15 let sum i = fixpoint (fun s -> fby(0,s) + i) ;;
```

More syntactic sugar...

```
node f p_i returns p_o =
    p_1 = e_1;
    p_2 = e_2;
    ...
    p_n = e_n;;
```

is encoded internally as:

```
let f p_i =
  let (p_1, p_2, ··· p_n) = fixpoint (fun (p_1, p_2, ··· p_n) -> (e_1, e_2, ··· e_n))
  in p_o ;;
```

Also:

- infix notation for **fby**
- e **when** e' stands for `when((fun () → e),e')`

A toy example

Let's implement an accelerator, `collatz`, computing the *Collatz* (*Syracuse*) sequence ending with value 1:

$$collatz(i) = \begin{cases} 1 \text{ if } i = 1 \\ collatz(i/2) \text{ if } i \text{ is pair} \\ collatz(3 * i + 1) \text{ if } i \text{ is even} \end{cases}$$

In ECLAT, it is programmed as follows:

```
ECLAT
1 let rec collatz(i) =
2   if i = 1 then i else
3   if i mod 2 = 0 then collatz (i / 2)
4   else collatz (3 * i + 1) ;;
5
```

with a pause at each (direct or tail-recursive) call to function collatz (because these is defined with keyword **rec**).

```
6
7
8
9
10
11
12
13
14                [ let _ = collatz(a) in …
15                || let _ = collatz(b) in …
16                                                ]
17
18
```

```
6
7
8
9
10
11
12
13  trap T in
14  [  let _ = collatz(a) in exit T
15  || let _ = collatz(b) in exit T
16                                          ]
17
18
```

```
6
7
8
9
10
11
12  signal cy in                                                    ESTEREL
13      trap T in
14      [  let _ = collatz(a) in exit T
15      || let _ = collatz(b) in exit T
16      || loop emit cy(0 fby ?cy + 1) ; pause end ])
17
18
```

```
 6
 7
 8  node main (a, b, button) returns (dur, occ, rdy) =          LUSTRE
 9
10
11    (dur, rdy) =
12               (exec signal cy in                              ESTEREL
13                       (trap T in
14                        [ let _ = collatz(a) in exit T
15                        || let _ = collatz(b) in exit T
16                        || loop emit cy (0 fby ?cy + 1) ; pause end ])
17                         ?cy
18                default (0 fby dur)) ;;
```

```
6
7
8   node main (a, b, button) returns (dur, occ, rdy) =                LUSTRE
9       occ     = (0 fby occ + 1) when rdy ;
10
11   (dur, rdy) =
12                (exec signal cy in                                ESTEREL
13                         (trap T in
14                          [ let _ = collatz(a) in exit T
15                          || let _ = collatz(b) in exit T
16                          || loop emit cy (0 fby ?cy + 1) ; pause end ])
17                           ?cy
18                 default (0 fby dur)) ;;
```

```
6
7
8    node main (a, b, button) returns (dur, occ, rdy) =                LUSTRE
9        occ     = (0 fby occ + 1) when rdy ;
10
11    (dur, rdy) = if (a < 1) or (b < 1) then (0, false) else
12                 (exec signal cy in                                  ESTEREL
13                          (trap T in
14                            [ let _ = collatz(a) in exit T
15                            || let _ = collatz(b) in exit T
16                            || loop emit cy(0 fby ?cy + 1) ; pause end ])
17                             ?cy
18                  default (0 fby dur)) ;;
```

```
6   signal stop ;;
7
8   node main (a, b, button) returns (dur, occ, rdy) =                  LUSTRE
9        occ      = (0 fby occ + 1) when rdy ;
10       ()       = if button then emit stop else () ;
11    (dur, rdy)  = if (a < 1) or (b < 1) then (0, false) else
12                  (exec signal cy in                                  ESTEREL
13                        suspend (trap T in
14                                [ let _ = collatz(a) in exit T
15                                || let _ = collatz(b) in exit T
16                                || loop emit cy(0 fby ?cy + 1) ; pause end ])
17                        when stop; ?cy
18                   default (0 fby dur)) ;;
```

Conclusion

- ECLAT design choices inspired by ESTEREL and LUSTRE (e.g., the non-blocking ECLAT construct **exec** for periodic execution of multicycle computations)
- the hope of being as expressive as ESTEREL and LUSTRE for programming reactive behaviors, while meeting the needs for high-level design of compute-intensive, timing-predictable and composable hardware accelerators.
- Experimental answer: a conservative extension of ECLAT with ESTEREL and LUSTRE constructs (taking inspiration from a lot of existing work in the community)
- But questions remain:
  - **checking causality** – at source level ? at circuit level ?
    ECLAT is compiled to hierarchical mealy machines (keeping the control flow of the original program) rather than flat, easier to analyze netlists
  - recover a simple (ML-like) **clock calculus** while keeping dynamicity of signal broadcasting
  - **software simulation** and **executable semantics**
  - keep it simple, with respect to applicability and potential users