An Introduction to Lustre

Marc Pouzet

École normale supérieure
Marc.Pouzet@ens.fr

May 4, 2017

The language Lustre

- Invented by Paul Caspi and Nicolas Halbwachs arround 1984, in Grenoble (France).
- A program is a set of equations. An equation defines a infinite sequence of values.
- Boolean operators applied point-wise, a unit-delay, and sampling operators.
- Equivalent graphical representation by block diagrams.
- Feedback loops must cross a unit delay.
- Time is synchronous: at every tick of a global clock, every operation does a step.
- Code generation to sequential code and formal verification techniques.
- An industrial success: SCADE (Esterel-Technologies company) is used for programming critical control sofware (e.g., planes, nuclear plants).

Lustre

Program by writting stream equations.

X	1	2	1	4	5	6	...
Y	2	4	2	1	1	2	...
1	1	1	1	1	1	1	...
$X + Y$	3	6	3	5	6	8	...
$X + 1$	2	3	2	5	6	7	...

Equation $Z = X + Y$ means that at any instant $n \in \mathbb{N}$, $Z_n = X_n + Y_n$.

Time is logical: inputs X and Y arrive **at the same time**; the output Z is produced at the same time.

Synchrony means that at instant n, all streams take their n-th value.

In practice, check that the current output is produced before the next input arrives.

Example: 1-bit adder

```
node full_add(a, b, c:bool) returns (s, co:bool);
  let
    s = (a xor b) xor c;
    co = (a and b) or (b and c) or (a and c);
  tel;
```

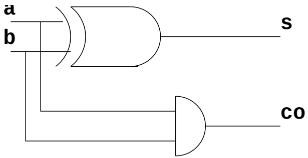
or:

```
node full_add(a, b, c:bool) returns (s, co:bool);
  let
    co = if a then b or c else b and c;
    s = (a xor b) xor c;
  tel;
```

Full Adder

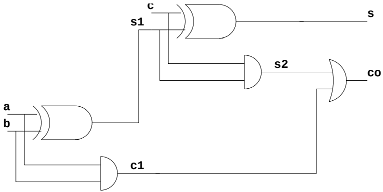
Compose two "half adder"

```
node half_add(a,b:bool)
returns (s, co:bool);
  let s = a xor b;
      co = a and b;
  tel;
```



Instanciate it twice:

```
node full_add_h(a,b,c:bool)
returns (s, co:bool);
  var s1,c1,c2:bool;
  let
    (s1, c1) = half_add(a,b);
    (s, c2) = half_add(c, s1);
    co = c1 or c2;
  tel;
```



Verify properties

How to be sure that `full_add` and `full_add_h` are equivalent?

$$\forall a, b, c : \texttt{bool}.\, \texttt{full\_add}(a, b, c) = \texttt{full\_add\_h}(a, b, c)$$

Write the following program and prove that it returns `true` at every instant!

```
-- file prog.lus
node equivalence(a,b,c:bool) returns (ok:bool);
  var o1, c1, o2, c2: bool;
  let
     (o1, c1) = full_add(a,b,c);
     (o2, c2) = full_add_h(a,b,c);
     ok = (o1 = o2) and (c1 = c2);
  tel;
```

Then, use the model-checking tool `lesar`:

```
% lesar prog.lus equivalence
--Pollux Version 2.2

TRUE PROPERTY
```

The Unit Delay

One can refer to the value of a input at the "previous" step.

X	1	2	1	4	5	6	...
pre X	*nil*	1	2	1	4	5	...
Y	2	4	2	1	1	2	...
Y -> *pre* X	2	1	2	1	4	5	...
S	1	3	4	8	13	19	...

The stream $(S_n)_{n \in \mathbb{N}}$ with $S_0 = X_0$ and $S_n = S_{n-1} + X_n$, for $n > 0$ is written:

```
S = X -> pre S + X
```

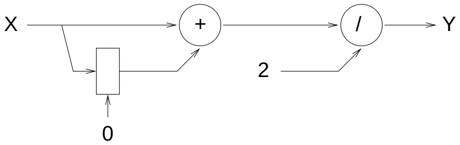
Introducing intermediate equations does not change the meaning of programs:

```
S = X -> I; I = pre S + X
```

Example: convolution

Define the sequence:

$$Y_0 = X_0/2 \quad \wedge \quad \forall n > 0. Y_n = (X_n + X_{n-1})/2$$




```
node convolution(X:real) returns (Y:real);
let Y = (X + (0 -> pre X)) / 2.0;
tel;
```

or:

```
node convolution(X:real) returns (Y:real);
var pY:int;
let Y = (X + pY) / 2;
    pY = 0 -> pre X;
tel;
```

Linear filters

FIR (Finite Impulse Response)

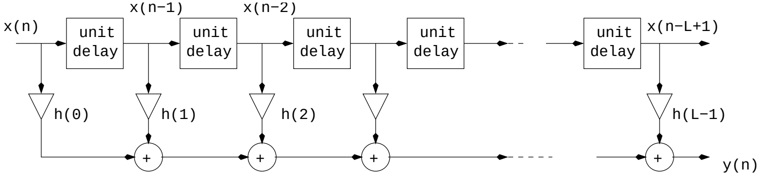
$$y(n) = \sum_{m=0}^{L-1} x(n-m)b(m)$$

IIR (Infinite Impulse Response) or recursive filter

$$y(n) = \sum_{m=0}^{L-1} x(n-m)b(m) + \sum_{m=1}^{M-1} y(n-m)a(m)$$

FIR

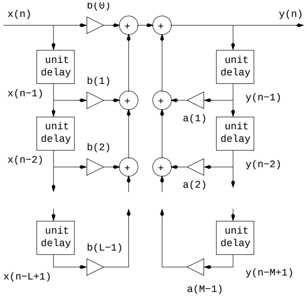
Build a block-diagram with three operators: a gain (multiplication by a constant), a sum and a unit delay (register).



Previous example

$\forall n \geq 0. y(n) = 1/2\,(x(n) + x(n-1))$

IIR



Example: follow x with a 20% gain.

$$\forall n \geq 0. y(n) = 0.2(x(n) - y(n-1)) + y(n-1)$$

```
node filter(x: real) returns (y:real);
  let y = 0.0 -> 0.2 * (x - pre y) + pre y; tel;
```

Retiming:

Optimise by moving unit delays arround combinatorial operators.

DEMO: type `luciole filter.lus filter`

Counting events

Count the number of instants where the input signal tick is true between two top.

```
node counter(tick, top:bool) returns (cpt:int);
let
    cpt = if top then 0
          else if tick then (0 -> pre cpt) + 1 else pre cpt;
tel;
```

Is this program well defined? Is-it deterministic? No: initialization issue.

tick	*f*	*f*	*t*	*t*	*t*	*t*	*t* ...
top	*f*	*f*	*f*	*f*	*t*	*f*	*f* ...
cpt	*nil*	*nil*	*nil*	*nil*	0	1	2 ...

Write instead:

```
    cpt = if top then 0
          else if tick then (0 -> pre cpt) + 1 else 0 -> pre cpt;
```

An explicit Euler integrator

```
node integrator(const step: real; x0, x':real) returns (x:real);
let
    x = x0 -> pre(x) + pre(x') * step;
tel;
```

step is a constant stream computed at compile-time.

Sinus/cosine functions

```
node sinus_cosinus(theta:real)
returns (sin,cos:real);
let sin = theta * integrator(0.01, 0.0, cos);
    cos = theta * integrator(0.01, 1.0, 0.0 -> pre sin);
tel;
```

Initial Value Problem (IVP)

f is a combinatorial function with y of type ty. t is the current time. $x(t)$ be defined by the IVP:

$$\dot{x} = f(y, t, x) \quad \text{with} \quad x(0) = x0$$

```
node ivp(const step: real; y: ty; read) returns (x: real)
  var t: real;
  let
    x = integr(step, x0, f(y, t, x));
    t = 0.0 -> pre t + step;
  tel;
```

Exercice

- Program a classical explicit Runge Kutta method (e.g., order 4).
- More difficult: program a variable step Runge Kutta method (RK45). Hint: use a control bit `error_too_large` to shrink the step dynamically.

Counting Beacons [1]

Counting becons and seconds to decide whether a train is on time.

Use an hysteresis with a low and high threshold to reduce oscillations.

```
node beacon(sec, bea: bool) returns (ontime, late, early: bool);
var diff, pdiff: int; pontime: bool;
let
  pdiff = 0 -> pre diff;
  diff = pdiff + (if bea then 1 else 0) +
         (if sec then -1 else 0);
  early = pontime and (diff > 3) or
          (false -> pre early) and (diff > 1);
  late = pontime and (diff < -3) or
         (false -> pre late) and (diff < -1);
  ontime = not (early or late);
  pontime = true -> pre ontime;
tel;
```

[1] This example is due to Pascal Raymond

Two types of properties

Safety property

"Something wrong never happen", i.e., a property is invariant and true in any accessible state. E.g.:

- "The train is never both early and late", it is either on time, late or early;
- "The train never passes immediately from late to early"; "It is impossible to stay late only a single instant".

Liveness property

"Something good with eventually happen.", i.e., any execution will reach a state verifying the property. E.g., "If the trains stop, it will eventually be late."

Remark:

"If the train is on time and stops for ten seconds, it will be eventually late" is a safety property.
Safety properties are critical ones in practice.

Formal verification and modeling of systems

A safety property ("something bad will never happen") is a boolean proved to be true at every instant.

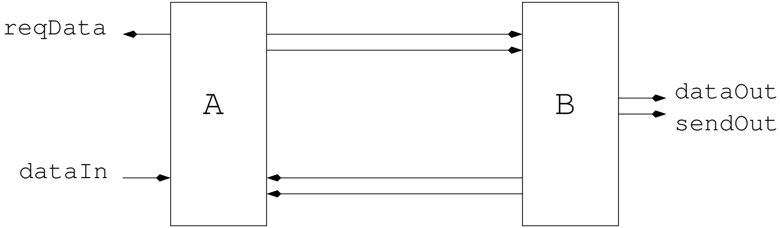
Example: the alternating bit protocol

A transmitter A; a receiver B. Two unreliable lines $A2B$ and $B2A$ that may loose messages.

- A asks for one input. It re-emits the data with $bit = true$ until it receives $ack = true$.
- It asks for an other input and emits the data with $bit = false$ until it receives $ack = false$.
- B sends $ack = false$ until it receives $bit = true$; it sends $ack = true$ until it receives $bit = false$;
- initialization: send anything with $bit = true$. The first message arriving with $bit = false$ is valid.

Objective:

Model and prove the protocol is correct, i.e., the network is the identity function (input sequence = output sequence) with two unreliable lines.



Model the asynchronous communication by adding a "presence" bit to every data: a pair $(\mathit{data}, \mathit{enable})$ is meaningful when $\mathit{enable} = \mathit{true}$.

The Sender

- *A* asks for one input. It re-emits the data with $bit = true$ until it receives $ack = true$.
- It asks for an other input and emits the data with $bit = false$ until it receives $ack = false$.

```
node A(dataIn: int; recB: bool; ack: bool)
returns (reqData: bool; send: bool; data: int; bit: bool);

var
  buff: int; chstate : bool;

let
  buff = if reqData then dataIn else (0 -> pre buff);
  chstate = recB and (bit = ack);
  reqData, send, bit =
    (false, true, true) ->
              pre (if chstate then (true, true, not bit)
                   else (false, send, bit));
  data = buff;
tel
```

The Receiver

- *B* sends *ack* = *false* until it receives *bit* = *true*; it sends *ack* = *true* until it receives *bit* = *false*;

```
node B(recA : bool; data: int; bit: bool;)
returns (sendOut: bool; dataOut: int; send2A: bool; ack: bool);

var chstate : bool;

let
  chstate = recA and (ack xor bit);

  sendOut, send2A, ack =
   (false, true, true) ->
        pre (if chstate then (true, true, not ack)
                else (false, true, ack));
  dataOut = data;
tel
```

Modeling the channel and the main property

```
node unreliable(loose: bool; presIn: bool)
returns (presOut: bool);
let
  presOut = presIn and not loose;
tel

-- The property that two signals [r] and [s] alternate.
node altern(r,s: bool) returns (ok: bool);
var
 s0, s1 : bool;
 ps0, ps1 : bool;
let
  ps0 = true -> pre s0;
  ps1 = false -> pre s1;
  s0 = ps0 and (r = s) or ps1 and s and not r;
  s1 = ps0 and r and not s or ps1 and not r and not s;
  ok = (true -> pre ok) and (s0 or s1);
tel
```

The main system

```
node obs(dataIn: int; looseA2B, looseB2A : bool;)
returns (ok : bool; reqData: bool; sendOut: bool);
var
  dataOut: int;
  sendA2B: bool; data: int; bit: bool;
  recA2B, recB2A : bool;
  sendB2A: bool; ack: bool;

let
 ok = altern(reqData, sendOut);

 recA2B = unreliable(looseA2B, sendA2B);
 recB2A = unreliable(looseB2A, sendB2A);

 reqData, sendA2B, data, bit = A(dataIn, recB2A, ack);
 sendOut, dataOut, sendB2A, ack = B(recA2B, data, bit);
tel

%aneto.local: lesar ba.lus obs
TRUE PROPERTY
```

Clocks: mixing slow and fast processes

A slow process is made by sampling its inputs; a fast one by oversampling its inputs.

The operators `when`, `current` and `merge`

`B`	*false*	*true*	*false*	*true*	*false*	*false*
`X`	x_0	x_1	x_2	x_3	x_4	x_5
`Y`	y_0	y_1	y_2	y_3	y_4	y_5
`Z = X when B`		x_1		x_3		
`K = Y when not B`	y_0		y_2		y_4	y_5
`T = current Z`	*nil*	x_1	x_1	x_3	x_3	x_3
`O = merge B T K`	y_0	x_1	y_2	x_3	y_4	y_5

The operator `merge` is not part of Lustre. It was introduced later in Lucid Synchrone and SCADE 6.

The Gilbreath trick in SCADE 6 [2]

Take a card deck where card alternate; split it in two; shuffle them arbitrarily. Then, if you take two successives cards, their colors are different (provided bottom cards have diff. colors).

```
node Gilbreath_stream (clock c:bool) returns (prop: bool; o:bool);
var
  s1 : bool when c;
  s2 : bool when not c;
  half : bool;

let
  s1 = (false when c) -> not (pre s1);
  s2 = (true  when not c) -> not (pre s2);
  o = merge (c; s1; s2);
  half = false -> (not pre half);

  prop = true -> not (half and (o = pre o));
tel;
```

[2] This code is due to Jean-Louis Colaco. The trick is exposed and proved in Coq in: "The Gilbreath Trick: A case study in Axiomatisation and Proof Development in the Coq Proof Assistant." Gerard Huet. May 1991.

The Gilbreath trick in Lustre

```
node Gilbreath_stream (c:bool) returns (OK: bool; o:bool);
var
  ps1, s1 : bool; ps2, s2 : bool; half : bool;
let

  s1 = if c then not ps1 else ps1;
  ps1 = false -> pre s1;
  s2 = if not c then not ps2 else ps2;
  ps2 = true  -> pre s2;

  o = if c then s1 else s2;

  half = false -> (not pre half);

  OK = true -> not (half and (o = pre o));
tel;
```

Proved automatically using Lesar (Pascal Raymond) and KIND2 (Cesare Tinelli).

A classical use of clock: the activation condition

Run a process on a slower by sub-sampling its inputs; hold outputs.

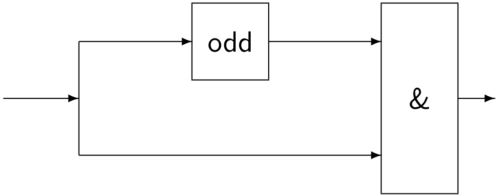
```
node sum(i:int) returns (s:int);
 let
   s = i -> pre s + i;
 tel;
```

1	1	1	1	1	1	1
`cond`	*true*	*false*	*true*	*true*	*false*	*true*
`sum(1)`	1	2	3	4	5	6
`sum(1 when cond)`	1		2	3		4
`(sum 1) when cond`	1		3	4		6

Sampling inputs *vs* sampling outputs

- `current (f(x when c))` is called an "activation condition"
- `f(x when c)` $\neq$ `(f x) when c`
- `current(x when c)` $\neq$ `x`

Why synchrony?




```
let half = true -> not (pre half);
    o = x & (x when half);
tel
```

It defines the sequence: $\forall n \in \mathbb{N}. o_n = x_n \,\&\, x_{2n}$

- It cannot be computed in bounded memory.
- Its corresponding Kahn networks has unbounded buffering.
- This is forbidden, a dedicated analysis for that: **clock calculus**

(Intuitive) Clocking rules in Lustre

Clocks must be declared and visible from the interface of a node.

```
node stables(i:int) ← base clock (true)
returns (s:int; ncond:bool;
         (ns:int) when ncond); ← clock declaration
var cond:bool;
    (l:int) when cond; ← clock declaration
let
    cond = true -> i <> pre i;
    ncond = not cond;
    l = somme(i when cond);
    s = current(l);
    ns = somme(i when ncond);
tel;
```

Constraints

Rules

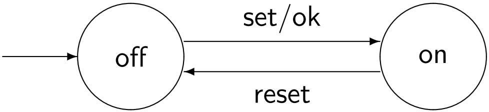
- Constants are on the base clock of the node.
- By default, variables are on the base clock of the node.
- Unless a clock is associated to the variable definition.
- $\mathit{clock}(e_1 \ \mathit{op} \ e_2) = \mathit{clock}(e_1) = \mathit{clock}(e_2)$
- $\mathit{clock}(e \ \mathtt{when} \ c) = c$
- $\mathit{clock}(\mathtt{current}; e) = \mathit{clock}(\mathit{clock}(e))$

Implementation choices

- Clocks are declared and verified. No automatic inference.
- Two clocks are equal if expressions that define them are syntactically equal.

One hot coding of Mealy machines

Represent a state by a Boolean variable.




```
node switch(set,reset:bool) returns (ok :bool);
 var on: bool;
 let
  on = false ->
         if set and not (pre on) then true
         else if reset and (pre on) then false
         else (pre on);
  ok = on;
 tel;
```

Think in term of an invariant: what is the expression defining the current value of on at every instant?

Verification with assertions

Consider a second version.

```
node switch2(set, reset:bool) returns (ok:bool);
  var s1, s2: bool;
let
    s1 = true -> if reset and pre s2 then true
                 else if pre s1 and set then false else pre s1;
    s2 = false -> if set and pre s1 then true
                  else if pre s2 and reset then false else pre s2;
    ok = s2;
tel;

node compare(set,  reset: bool) returns (ok: bool);
  let  ok = switch(set,  reset) = switch2(set, reset); tel;
```

We get:

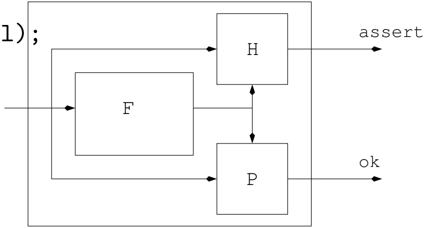
```
% lesar prog.lus compare
--Pollux Version 2.2

TRUE PROPERTY
```

Synchronous observers

Comparison is a particular case of a synchronous observer.

- Let $y = F(x)$, and $ok = P(x, y)$ for the property relating x and y
- assert$(H(x, y))$ is an hypothesis on the environment.

```
node check(x:t) returns (ok:bool);
  let
    assert H(x,y);
    y = F(x);
    ok = P(x,y);
  tel;
```



If *assert* is (infinitely) true, then *ok* stay infinitely true
(always(*assert*)) $\Rightarrow$ (always(*ok*)).

Any safety temporal property can be expressed as a Lustre program [6, 5]. No temporal logic/language is necessary.

Safety temporal properties are regular Lustre programs

Array and slices

Array are manipulated by slices with implicit point-wise extension of operations. `t[0..N]` defines a slice of t from index 0 to N.

```
const N = 10;

node plus(const N: int; a1, a2: int^N) returns (o: int^N);
  let
    o[1..N] = a1[1..N] + a2[1..N];
  tel;
```

Arrays

```
-- serial adder

node add(a1: bool^N; a2: bool^N; carry: bool)
returns (a: bool^N; new_carry: bool);
  var c: bool^N;
  let
    (a[0..N-1], c[0..N-1]) =
      bit_add(a1[0..N-1], a2[0..N-1], ([carry] | c[0..N-2]));
    new_carry = c[N-1];
  tel;

node add_short(a1: bool^N; a2: bool^N; carry: bool)
returns (a: bool^N; new_carry: bool);
  var c: bool^N;
  let
    (a, c) = bit_add(a1, a2, ([carry] | c[0..N-2]));
    new_carry = c[N-1];
  tel;
```

Conclusion

Compilation

- Static, compile-time checking to ensure the absence of deadlock, that the code behave deterministically.
- Execution in bounded memory and time.
- Code generation into sequential "single loop" code. More advanced methods into automata and/or modular.

Verification by Model-checking

- Synchronous observer: a safety property is a Lustre program
- Avoid to introduce an ad-hoc temporal logic.
- Tool Lesar (BDD technique) by Pascal Raymond (VERIMAG, France).
- KIND and KIND2 (k-induction, PDR based on SMT techniques) by Cesare Tinelli (Iowa State Univ., USA).
- Plug-in (k-induction based on SAT techniques) by Prover-Technologies (associated to SCADE 6).

Related languages and verification tools

Various teams have done their own variant of Lustre.

Language embedding in Haskell

- Copilot (Nasa, USA), an Embedding of Lustre.
- FRAN (images, animation), Functional Reactive Programming (FRP), Hawk (architecture), Lava (synchronous circuits).
- Based on a compilation-by-evaluation technique.

Language extensions, formal verification

- Heptagon: Extended Lustre (automata, arrays) with controller synthesis (Gwenael Delaval, Univ. Grenoble)
- Prelude: Lustre with periodic clocks and a compiler that generates tasks for a real-time OS (Julien Forget, Univ. Lille).
- Lustre compiler at Onera for verification purposes (Pierre-Loic Garoche)

References

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.

Jr Edmund M Clarke, Orna Grumberg, and Doron A Peled.
Model Checking.
The MIT Press, 1999.

N. Halbwachs.
Synchronous programming of reactive systems.
Kluwer Academic Pub., 1993.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud.
The synchronous dataflow programming language LUSTRE.
Proceedings of the IEEE, 79(9):1305–1320, September 1991.

N. Halbwachs, F. Lagnier, and C. Ratel.
Programming and verifying real-time systems by means of the synchronous data-flow programming language lustre.
IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems, September 1992.

N. Halbwachs, F. Lagnier, and P. Raymond.
Synchronous observers and the verification of reactive systems.
In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

N. Halbwachs, P. Raymond, and C. Ratel.
Generating efficient code from data-flow programs.
In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.