Synchronous Modeling and Verification with the Language Lustre

Marc Pouzet

École normale supérieure
Marc.Pouzet@ens.fr

April 30, 2020
Synchronous modeling with Lustre

- A domain specific language to model reactive systems.
- A reactive system = a system with *continuous interaction* with an external environment which *imposes timing* [1].
- Lustre: invented by Paul Caspi and Nicolas Halbwachs in 1984, in Grenoble (France) [3].
- A ideal *synchronous model of time*: at design time, make as if all process advance synchronously and share a global discrete time scale.
- Then check that the implementation is fast enough, i.e., the time to react is less than the next arrival of input.
- The idea of synchronous circuits but applied to software.
The language Lustre

• A time evolving value (called a *signal*) is an infinite sequence \((x_n)_{n \in \mathbb{N}}\); a *system* is a function from signals to signals.

• Given a Lustre model, the compiler generate sequential code or a representation for formal verification by Model-checking techniques [7].

• Lustre had a tremendous impact on *formal methods* and software engineering, both scientifically and industrially.

• The synchronous language Scade \(^1\) deeply rely on the principles of Lustre.

  Scade is the *de facto standard* for critical software development: fly-by-wire command in planes, braking system, engine control, train tracking, etc.

---

\(^1\) https://www.college-de-france.fr/media/gerard-berry/UPL9185028255611736393_BP_CollegeDeFrance_23_avril_2013.pdf
Why should we care about Lustre in a OS class?

• First, see that there are several possible interpretation of time and concurrency: read [1].

• Time can be logical of physical. E.g., in a synchronous circuit, a circuit is first described ideally in *logical time* — time is $\mathbb{N}$ — while the actual implementation takes *physical time* to compute.

• Time can be global (e.g., circuit) or local (e.g., thing of a distributed system).

• Time can be imposed by an external environment or by the OS itself.

• In a real-time system, time is imposed by the environment (e.g., some physical system); in a general purpose OS, the OS does its best to ensure timing in a best effort manner.

• Parallel composition can be ideal, synchronous or by interleaving; it can preserve determinacy or not.

• A deep observation (R. Milner [15]): it is possible to model an asynchronous system synchronously (and not the converse). We exploit it to model various aspects of an OS using Lustre.
An introduction to Lustre
Lustre

Program by writing stream equations.

\[
\begin{array}{cccccccc}
X & 1 & 2 & 1 & 4 & 5 & 6 & \ldots \\
Y & 2 & 4 & 2 & 1 & 1 & 2 & \ldots \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots \\
X + Y & 3 & 6 & 3 & 5 & 6 & 8 & \ldots \\
X + 1 & 2 & 3 & 2 & 5 & 6 & 7 & \ldots \\
\end{array}
\]

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”

```javascript
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?

∀a, b, c : bool. full_add(a, b, c) = 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 an input at the “previous” step.

<table>
<thead>
<tr>
<th>$X$</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pre\ X$</td>
<td>nil</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>$Y$</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>...</td>
</tr>
<tr>
<td>$Y \rightarrow pre\ X$</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>$S$</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>13</td>
<td>19</td>
<td>...</td>
</tr>
</tbody>
</table>

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:

$$S = X \rightarrow pre\ S + X$$

Introducing intermediate equations does not change the meaning of programs:

$$S = X \rightarrow I; I = pre\ S + X$$
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.

Write instead:
cpt = if top then 0
   else if tick then (0 -> pre cpt) + 1 else 0 -> pre cpt;
Modeling a watchdog [8]

A watchdog

- Continuously observe a device and checks that it is alive. Reset it when some event (e.g., timeout) happens.
- Three input arguments: set, reset and deadline
- One output alarm when the watchdog is armed and deadline is true.

```plaintext
node WATCHDOG1(set,reset,deadline:bool) returns (alarm:bool);
var watchdog_is_on:bool;
let
    alarm = deadline and watchdog_is_on;
    watchdog_is_on = false -> if set then true
    else if reset then false
    else pre(watchdog_is_on);

-- set and reset are never true at the same time
assert not(set and reset);
tel;
```
A Watchdog timer

The processor emits periodically a time delay and the watchdog decrements a counter from the received value. If the counter reaches zero before being reset, the processor is supposed to be failed and it must be reset.

- receive set, reset and delay
- emits an alarm when set is absent (false) for a delay delay.
- real-time is that of the implementation.

```plaintext
node WATCHDOG_timer(set, reset: bool; delay: int)
  returns (alarm: bool);
var remaining_delay: int; deadline: bool;
let
  alarm = WATCHDOG1(set, reset, deadline);
  deadline = EDGE(remaining_delay = 0);
  remaining_delay = if set then delay else
                   (0 -> pre(remaining_delay)-1);
  tel;

node EDGE (b: bool) returns (edge: bool);
let edge = false -> b and not pre(b); tel;
```
A word on causality loops

All recursive feedback loop must cross an explicit unit delay. E.g., the following equation is rejected:

\[ x = x + 1; \]

All sequence must be computable sequentially, that is, a variable \( x \) must not depend instantaneously on itself.

Even in the case where the equation that defines \( x \) has a unique solution. E.g. the following definitions do have a unique solution but are rejected.

\[ x = (2 * x - 1) / x; \]
\[ \text{tobe} = \text{tobe or not tobe} /* The Berry example */ \]

/ * A combinatorial cyclic circuit which returns */
/ * either \( y = f \circ g (x) \) or \( y = g \circ f (x) \) */

\[ x1 = \text{if} \ c \ \text{then} \ x \ \text{else} \ y2; \]
\[ x2 = \text{if} \ c \ \text{then} \ y1 \ \text{else} \ x; \]
\[ y1 = f(x1); \]
\[ y2 = g(x2); \]
\[ y = \text{if} \ c \ \text{then} \ y2 \ \text{else} \ y1 \]
Example: convolution

Define the sequence:

\[ Y_0 = \frac{X_0}{2} \quad \land \quad \forall n > 0. \ Y_n = \frac{X_n + X_{n-1}}{2} \]

\[
\begin{align*}
X & \quad \rightarrow \quad + \quad \rightarrow \quad / \quad \rightarrow \quad Y \\
\quad & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 0 \\
& \quad 2
\end{align*}
\]

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

\[ \forall n \geq 0. y(n) = \frac{1}{2} (x(n) + x(n - 1)) \]
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
An Euler integrator

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

node forward(const step: real; x0, x’:real) returns (x:real);
let
    x = x0 -> pre(x + 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, pre sin);
tel;
Initial Value Problem (IVP)

A differential equation:

\[ \dot{x} = f(y, t, x) \quad \text{with} \quad x(0) = x_0 \]

can be approximated by computing a sequence \((x_n)_{n \in \mathbb{N}}\) a successive time steps \(n \cdot h\) with \(n \in \mathbb{N}\). The derivative \(\dot{x}\) is approximated by \((x(n + 1) - x(n))/h\).

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

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

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

```python
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;
```

²This example is due to Pascal Raymond [17]
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 \((\text{data}, \text{enable})\) is meaningful when \(\text{enable} = \text{true}\).
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 $ack = false$.

```ml
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

<table>
<thead>
<tr>
<th>B</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>x₀</td>
<td>x₁</td>
<td>x₂</td>
<td>x₃</td>
<td>x₄</td>
<td>x₅</td>
</tr>
<tr>
<td>Y</td>
<td>y₀</td>
<td>y₁</td>
<td>y₂</td>
<td>y₃</td>
<td>y₄</td>
<td>y₅</td>
</tr>
<tr>
<td>Z = X when B</td>
<td>x₁</td>
<td></td>
<td>x₃</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K = Y when not B</td>
<td>y₀</td>
<td>y₁</td>
<td>y₂</td>
<td>y₃</td>
<td>y₄</td>
<td>y₅</td>
</tr>
<tr>
<td>T = current Z</td>
<td>nil</td>
<td>x₁</td>
<td>x₁</td>
<td>x₃</td>
<td>x₃</td>
<td>x₃</td>
</tr>
<tr>
<td>O = merge B Z K</td>
<td>y₀</td>
<td>x₁</td>
<td>y₂</td>
<td>x₃</td>
<td>y₄</td>
<td>y₅</td>
</tr>
</tbody>
</table>

The operator merge is not part of Lustre. It was introduced in 1996 in Lucid Synchrone [16] (paper [5]) \(^3\) and Scade 6 [6].

\(^3\)The language is no more maintained but the documentation is still available at: https://www.di.ens.fr/~pouzet/lucid-synchrone/
A new version of the Watchdog

The watchdog now receives an extra signal that models its clock. Hence, the time delay is counted according to the number of occurrences of a time_unit

```plaintext
define WATCHDOG3(set,reset,time_unit:bool;delay:int) returns (alarm:bool):
  var clk:bool;
  let
    -- the watchdog is activated when either time_unit paces or
    -- signal set or reset is active.
    alarm = current(WATCHDOG2((set,reset,delay) when clk));
  clk = true -> set or reset or time_unit;
  tel;
```

The composition: \( \text{current}(f(x \text{ when } c)) \) is called an “activation condition”
The Gilbreath trick

The Gilbreath shuffle (from Wikipedia):

• Deal off any number of the cards from the top of a deck onto a new pile.
• Riffle the new pile with the remainder of the deck.

A trick based on the resulting Gilbreath permutations was formalized and verified in Coq by G. Huet

Input: two decks of alternating colours (red, black, red, black, ...) whose bottom cards have different colours.

Output: one deck of alternating red/black pairs.

The property is implied by the following one on Boolean streams:

\textit{if }s_1 \textit{ and } s_2 \textit{ be two alternating streams starting with different values; let } o \textit{ be a stream built by “riffle shuffling” } s_1 \textit{ and } s_2, \textit{ then } o \textit{ is such that it is the succession of pairs of different values.}
The Gilbreath trick in Scade 6 [6]

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;

Proved automatically with the verification engine of Scade (by Prover).
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 or Kind 2.
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;

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>sum(1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>sum(1 when cond)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sum 1) when cond</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sampling inputs vs sampling outputs

- current \( (f(x \text{ when } c)) \) is called an “activation condition”
- \( f(x \text{ when } c) \neq (f x) \text{ when } c \)
- current\( (x \text{ when } c) \neq x \)
Why synchrony?

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.
- $\text{clock}(e_1 \text{ op } e_2) = \text{clock}(e_1) = \text{clock}(e_2)$
- $\text{clock}(e \text{ when } c) = c$
- $\text{clock}(\text{current}; e) = \text{clock}(\text{clock}(e))$

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.

```plaintext
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 \( y \)
- \( \text{assert}(H(x, y)) \) is an hypothesis on the environment.

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

If \( \text{assert} \) is (infinitely) true, then \( ok \) stay infinitely true
\((\text{always}(\text{assert})) \Rightarrow (\text{always}(ok)))\).

Any safety temporal property can be expressed as a Lustre
program [13, 12]. 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 \( N \).

\[
\text{const N} = 10;
\]

\[
\text{node plus(const N: int; a1, a2: int^N) returns (o: int^N);} \\
\text{let} \\
\quad o[1..N] = a1[1..N] + a2[1..N]; \\
\text{tel;}
\]
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;
A boolean implementation of bounded arithmetic

const sz=3;

type count = bool^sz;

const zero = [true, false, false];

node succ(x: count) returns (y: count);
let
    y = ([false] | x[0..1]);
tel

node inf2(x: count) returns (y:bool);
let
    y = x[0] or x[1] or x[2];
tel
We do not go further on the use of Lustre for circuit description.

Let us go back to our initial intend: exploit the idea of a synchronous modeling of asynchronous features.

Read the beautiful paper by Halbwachs [10]
Modeling asynchronous communication


Imagine two processors that execute periodically with same period. But they are not physically synchronised hence clocks can have some jittering that is bounded.

They communicate through black board. Such a distributed implementation is used for fly-by-wire command.

“Quasi-synchrony” is a discrete-time abstraction of it. Two process execute in quasi-synchrony if there is at most 0, 1 or two step of one between two steps of the other.

\(^5\)https://guillaume.baudart.eu/thesis/
node is_qs (const d: int; x, y: bool) returns (ok : bool);
var xav, yav, pxav, pyav : int;
let
    pxav = 0 -> pre xav;
    pyav = 0 -> pre yav;
    xav = if y then 0 else if x then pxav + 1 else pxav;
    yav = if x then 0 else if y then pyav + 1 else pyav;
    ok = (xav <= d) and (yav <= d);
tel
node is_qs2 (x, y: bool) returns (ok : bool);
var xav, yav, pxav, pyav : count;
let
  pxav = zero -> pre xav;
  pyav = zero -> pre yav;
  xav = if y^3 then zero else if x^3 then succ(pxav) else pxav;
  yav = if x^3 then zero else if y^3 then succ(pyav) else pyav;

  ok = (inf2(xav) and inf2(yav)) and (true -> pre ok);
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

G. Berry.
Real time programming: Special purpose or general purpose languages.  

Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet.
Clock-directed Modular Code Generation of Synchronous Data-flow Languages.  

P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice.
Lustre: a declarative language for programming synchronous systems.  

Paul Caspi.
*The Quasi-Synchronous Approach to Distributed Control System Design, Crisys "Cooking Book"*.  

Paul Caspi and Marc Pouzet.
Synchronous Kahn Networks.  

Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet.

Jr Edmund M Clarke, Orna Grumberg, Daniel Kroening, Doron A Peled, and Helmut Veith.  
*Model Checking*.  
second edition.

N. Halbwachs.
*Synchronous programming of reactive systems*.  