A TUTORIAL OF LUSTRE

Nicolas HALBWACHS, Pascal RAYMOND

January 24, 2002

Contents

This document is an introduction to the language Lustre_V4 and its associated tools. We will not give a systematic presentation of the language, but a complete bibliography is added. The basic references are [8, 12]. The most recent features (arrays, recursive nodes) are described in [32].

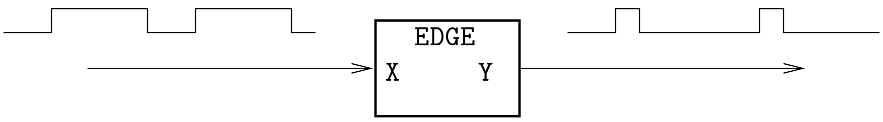


Figure 1: A Node

1 Basic language

A LUSTRE program or subprogram is called a **node**. LUSTRE is a functional language operating on **streams**. For the moment, let us consider that a stream is a finite or infinite sequence of values. All the values of a stream are of the same type, which is called the type of the stream. A program has a cyclic behavior. At the nth execution cycle of the program, all the involved streams take their nth value. A node defines one or several output parameters as functions of one or several input parameters. All these parameters are streams.

1.1 Simple control devices

1.1.1 The *raising edge* node

As a very first example, let us consider a Boolean stream $X = (x_1, x_2, \ldots, x_n, \ldots)$. We want to define another Boolean stream $Y = (y_1, y_2, \ldots, y_n, \ldots)$ corresponding to the rising edge of X, i.e., such that y_{n+1} is true if and only if x_n is false and x_{n+1} is true (X raised from false to true at cycle $n + 1$). The corresponding node (let us call it `EDGE`) will take X as an input parameter and return Y as an output parameter The **interface** of the node is the following:

```
node EDGE (X: bool) returns (Y: bool);
```

The definition of the output `Y` is given by a single **equation**:

```
Y = X and not pre(X);
```

This equation defines "`Y`" (its left-hand side) to be *always equal* to the right-hand side **expression** "`X and not pre(X)`". This expression involves the input parameter X and three **operators**:

- "`and`" and "`not`" are usual Boolean operators, extended to operate pointwise on streams: if $A = (a_1, a_2, \ldots, a_n, \ldots)$ and $B = (b_1, b_2, \ldots, b_n, \ldots)$ are two Boolean streams, then `A and B` is the Boolean stream $(a_1 \wedge b_1, a_2 \wedge b_2, \ldots, a_n \wedge b_n, \ldots)$. Most usual operators are available in that way, and are called "**data-operators**". Here is the list of built-in data operators in LUSTRE-V4[1]:

[1] Most of them have obvious meanings. "`xor`" is the exclusive "or", "`#`" takes any number of Boolean parameters, and returns true at cycle n if and only if at most one of its parameters is true. "`int`" and "`real`" are explicit conversion operators.

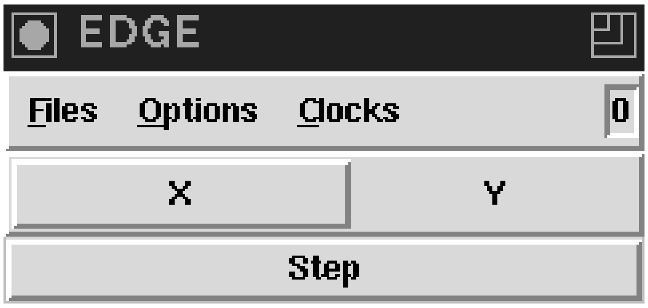


Figure 2: Simulating a node

`and`	`or`	`xor`	`not`	`#`
`if...then...else...`			`+`	`-`
`*`	`/`	`div`	`mod`	`=`
`<>`	`<`	`<=`	`>`	`>=`
`int`	`real`			

- The "`pre`" (for "**previous**") operator allows to refer at cycle n to the value of a stream at cycle $n-1$: if `A` $= (a_1, a_2, \ldots, a_n, \ldots)$ is a stream, `pre(A)` is the stream $(nil, a_1, a_2, \ldots, a_{n-1}, \ldots)$. Its first value is the undefined value *nil*, and for any $n > 1$, its nth value is the $(n-1)$th value of `A`.
- The "`->`" (followed by) operator allows to initialize streams. If `A` $= (a_1, a_2, \ldots, a_n, \ldots)$ and `B` $= (b_1, b_2, \ldots, b_n, \ldots)$ are two streams of the same type, then "`A->B`" is the stream $(a_1, b_2, \ldots, b_n, \ldots)$, equal to `A` at the first instant, and then forever equal to `B`. In particular, this operator allows to mask the "nil" value introduced by the `pre` operator.

As a consequence, if `X` $= (x_1, x_2, \ldots, x_n, \ldots)$ the expression "`X and not pre(X)`" represents the stream $(nil, x_2 \wedge \neg x_1, \ldots, x_n \wedge \neg x_{n-1}, \ldots)$. In order to avoid the "nil" value, let us use the `->` operator, and the built-in constant `false` [2].

The complete definition of the node `EDGE` is the following:

```
node EDGE (X: bool) returns (Y: bool);
let
  Y = false -> X and not pre(X);
tel
```

1.1.2 Simulating a node

Let us write the node `EDGE` in a file `F.lus` and call the graphical simulator, giving the name of the file and the name of the node representing the main program (Fig. 2):

[2]A lustre constant denotes an infinite stream of a same value. Pre-defined constants are `false`, `true`, and immediate arithmetic values. For instance, the expression `3.14` denotes $(3.14, 3.14, 3.14, \ldots)$.

```
luciole F.lus EDGE
```

The simulator opens a window containing:

- a label corresponding to the output of the node `Y`; this widget behaves as a "lamp", it is highligted when the output is "true".
- a widget corresponding to the input of the node `X`. This widget behaves differently depending on the *mode*: in the *auto-step mode*, inputs are supposed to be exclusive, so activating a input button provoques a single reaction of the program; in the *compose mode*, inputs behaves as "check-buttons", so several inputs can be selected, without provoquing a reaction. Whatever is the mode, the step button provoques a single reaction. The menu "Clocks" allows the user to switch between the "auto-step" and the "compose" mode.

In this example, the mode is not very important, since there is only one input. Try, for instance, the auto-step mode:

- pressing `X` provoques a reaction with `X` = true
- pressing `STEP` provoques a reaction with `X` = false

1.1.3 Compiling a node

The graphical simulator is based on an interpretor of LUSTRE programs. You can also simulate the program by compiling it into a C program. Let us call the compiler giving the name of the file and the name of the main node:

```
lustre F.lus EDGE
```

We get a file `EDGE.oc` which contains the object code written in the ESTEREL-LUSTRE common format OC [23]. We can simulate this program using the LUX simulator, by typing:

```
lux EDGE.oc
```

The OC code is translated into an instrumented program `EDGE.c`. A standard main loop program is also generated in a file `EDGE_loop.c`. Then the two files are compiled and linked into an executable program `EDGE`. Calling `EDGE` we get

```
##### STEP 1 ##################
X (true=1/false=0) ?
```

asking for a first value of `X`, of type `bool`. We type "1", and get

```
##### STEP 1 ##################
X (true=1/false=0) ? 1
Y = false
##### STEP 2 ##################
X (true=1/false=0) ? 1
```

The first value of Y is false, and a new value is wanted for X. We can then continue the simulation, and terminate it by "^C".

Let us have a look at the C code, in the file EDGE.c. The file contains some declarations, and the procedure EDGE_step, shown below, which implements the generated automaton. The procedure selects the code to be executed according to the value of the context variable "ctx->current_state", which is initialized to 0.

```
void EDGE_step(EDGE_ctx* ctx){
   switch(ctx->current_state){
   case 0:
      ctx->_V2 = _false;
      EDGE_O_Y(ctx->client_data, ctx->_V2);
      if(ctx->_V1){
         ctx->current_state = 1; break;
      } else {
         ctx->current_state = 2; break;
      }
   break;
   case 1:
      ctx->_V2 = _false;
      EDGE_O_Y(ctx->client_data, ctx->_V2);
      if(ctx->_V1){
         ctx->current_state = 1; break;
      } else {
         ctx->current_state = 2; break;
      }
   break;
   case 2:
      if(ctx->_V1){
         ctx->_V2 = _true;
         EDGE_O_Y(ctx->client_data, ctx->_V2);
         ctx->current_state = 1; break;
      } else {
         ctx->_V2 = _false;
         EDGE_O_Y(ctx->client_data, ctx->_V2);
         ctx->current_state = 2; break;
      }
   break;
   } /* END SWITCH */
   EDGE_reset_input(ctx);
}
```

1.1.4 Minimizing an automaton

The automaton corresponding to EDGE.oc is drawn in Fig. 3. The program is in the state 0 at the initial instant. In this state, the output is false whatever

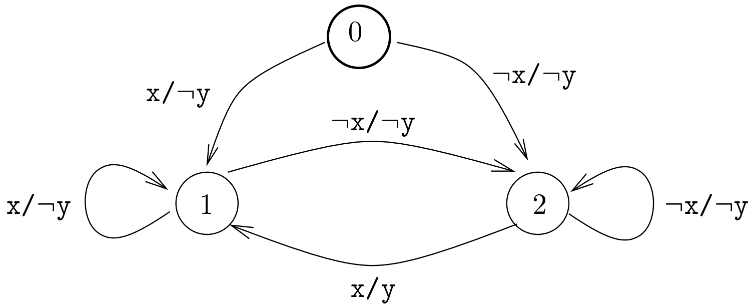


Figure 3: The automaton of the node `EDGE`

be the input, but, depending on the value of `X`, the next state will be either 1 (corresponding to `pre X` = false) or 2 (corresponding to `pre X` = true). The state 1 behaves like the initial state. In the state 2, the next state is computed like in the other ones, but the value of `Y` depends on the the value of `X`. One can note that this automaton is not "minimal" since states 0 and 1 are equivalent. There is two ways to obtain a minimal automaton:

- The oc code can be minimized by calling:

  ```
  ocmin EDGE.oc -v
  ```

  The `-v` option sets the verbose mode, and we get:

  ```
  Loading automaton ...
  => done : 3 states
  Minimizing (algo no 1) ...
  => done : 3 => 2  (2 steps)
  ```

  That means that the automaton was not minimal, and a minimal one, with only two states, was written in the file `EDGE_min.oc`.

- The Lustre compiler can directly produce a minimal automaton using the `-demand` option[3]:

  ```
  lustre F.lus EDGE -demand -v
  ```

  We get:

  ```
  DONE =>     2 states     4 transitions
  ```

[3]Two algorithms for the construction of automata are implemented in the compiler. The first one is called "data driven" (the default one), and the result is in general non minimal. The second is called "demand driven", it takes more time, and the result is minimal.

1.1.5 Re-using nodes

Once a node has been defined, it can be called from another node, using it as a new operator. For instance, let us define another node, computing the falling edge of its input parameter:

```
node FALLING_EDGE (X: bool) returns (Y: bool);
let
  Y = EDGE(not X);
tel
```

We can add this node declaration to our file `F.lus`, call the compiler with `FALLING_EDGE` as the main node:

```
lustre F.lus FALLING_EDGE
```

and simulate the resulting code:

```
lux FALLING_EDGE.oc
```

1.1.6 The *switch* node

The `EDGE` node is of very common usage for "deriving" a Boolean stream, i.e., transforming a "level" into a "signal". The converse operation is also very useful, it will be our second example: We want to implement a "switch", taking as input two signals "`set`" and "`reset`" and an initial value "`initial`", and returning a Boolean "`level`". Any occurrence of "`set`" rises the "`level`" to true, any occurrence of "`reset`" resets it to false. When neither "`set`" nor "`reset`" occurs, the "`level`" does not change. "`initial`" defines the initial value of "`level`". In LUSTRE, a signal is usually represented by a Boolean stream, whose value is true whenever the signal occurs. Below is a first version of the program:

```
node SWITCH1 (set, reset, initial: bool) returns (level: bool);
let
  level = initial -> if set then true
                     else if reset then false
                     else pre(level);
tel
```

which specifies that the "`level`" is initially equal to "`initial`", and then forever,

- if "`set`" occurs, then it becomes true
- if "`set`" does not occur but "`reset`" does, then "`level`" becomes false
- if neither "`set`" nor "`reset`" occur, "`level`" keeps its previous value (notice that "`level`" is **recursively defined**).

However, this program has a flaw: It cannot be used as a "one-button" switch, whose level changes whenever its unique button is pushed. Let "`change`" be a Boolean stream representing a signal, then the call

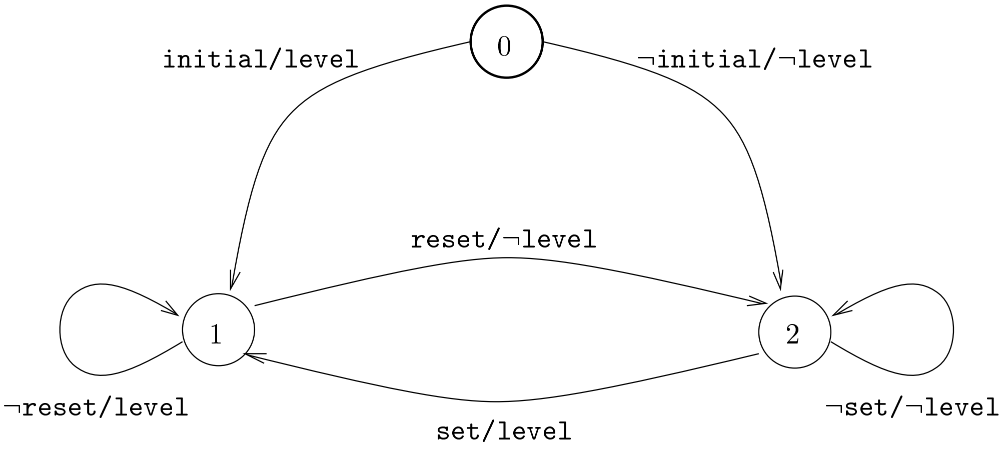


Figure 4: The automaton of the node SWITCH

```
state = SWITCH1(change,change,true);
```

will compute the always true stream: “`state`” is initialized to true, and never changes because the “`set`” formal parameter has been given priority (Try it). To get a node that can be used both as a “two-buttons” and a “one-button” switch, we have to make the program a bit more complex: the “`set`” signal must be considered only when the switch is turned off. We get the following program:

```
node SWITCH (set, reset, initial: bool) returns (level: bool);
let
  level = initial -> if set and not pre(level) then true
                     else if reset then false
                     else pre(level);
tel
```

Compiling this node, we get the automaton drawn in Fig. 4. The nodes `SWITCH` and `SWITCH1` behave the same as long as “`set`” and “`reset`”

1.2 Numerical examples

1.2.1 The *counter* node

It is very easy in LUSTRE to write a recursive sequence. For instance the definition `C = 0 -> pre C + 1;` defines the sequence of natural. Let us complicate this definition to build a integer sequence, whose value is, at each instant, the number of “true” occurences in a boolean flow `X`:

```
C = 0 -> if X then (pre C + 1) else (pre C);
```

This definition does not meet exactly the specification, since it ignores the initial value of `X`. A well initialized counter of `X` occurences is for instance:

```
PC = 0 -> pre C;
C = if X then (PC + 1) else PC;
```

Let us complicate this example to obtain a general counter with additionnal inputs:

- an integer `init` which is the initial value of the counter.
- an integer `incr` to add to counter each time `X` is true,
- a boolean `reset` which sets the counter to the value of `init`, whatever is the value of `X`.

The complete definition of the counter is:

```
node COUNTER(init, incr : int; X, reset : bool) returns (C : int);
var PC : int;
let
  PC = init -> pre C;
  C  = if reset then init
       else if X then (PC + incr)
       else PC;
tel
```

This node can be used to define the sequence of odd integers:

```
odds = COUNTER(0, 2, true, true->false);
```

Or the integers modulo 10:

```
reset = true -> pre mod10 = 9;
mod10 = COUNTER(0, 1, true, reset);
```

1.2.2 The *integrator* node

This example involves real values. Let f be a real function of time, that we want to integrate using the trapezoid method. The program receives two real-valued streams `F` and `STEP`, such that

$$\mathtt{F}_n = f(x_n) \text{ and } x_{n+1} = x_n + \mathtt{STEP}_{n+1}$$

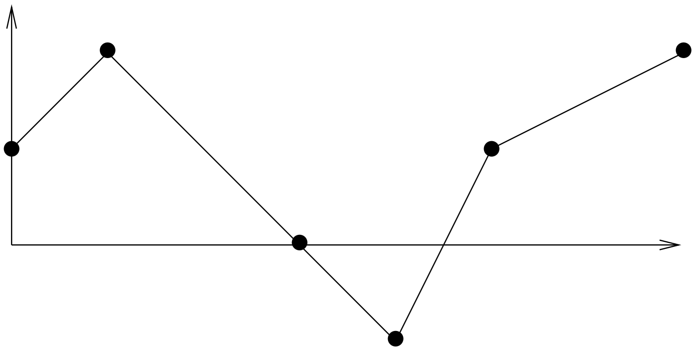
It computes a real-valued stream `Y`, such that

$$\mathtt{Y}_{n+1} = \mathtt{Y}_n + (\mathtt{F}_n + \mathtt{F}_{n+1}) * \mathtt{STEP}_{n+1}/2$$

The initial value of `Y` is also an input parameter:

```
node integrator(F,STEP,init: real) returns (Y: real);
let
  Y = init -> pre(Y) + ((F + pre(F))*STEP)/2.0;
tel
```

Try this program on the example shown in Fig. 5.



init	0					
STEP	-	1	2	1	1	2
F	1	2	0	-1	1	2
Y	0	1.5	3.5	3	3	6

Figure 5: Use of the integrator

1.2.3 The *sinus/cosinus* node

One can try to loop two such integrators to compute the functions $\sin(\omega t)$ and $\cos(\omega t)$ in a simple-minded way:

```
node sincos(omega:real) returns (sin, cos: real);
let
  sin = omega * integrator(cos, 0.1, 0.0);
  cos = 1 - omega * integrator(sin, 0.1, 0.0);
tel
```

Called on this program, the compiler complains that there is a **deadlock**. As a matter of fact, the variables `sin` and `cos` instantaneously depend on each other, i.e., the computation of the nth value of `sin` needs the nth value of `cos`, and conversely. We have to cut the dependence loop, introducing a "`pre`" operator:

```
node sincos(omega:real) returns (sin, cos: real);
let
  sin = omega * integrator(cos, 0.1, 0.0);
  cos = 1 - omega * integrator(0.0 -> pre(sin), 0.1, 0.0);
tel
```

Try this program, and observe the divergence (with omega = 1.0 for instance)!

1.3 Tuples

1.3.1 Nodes with several outputs

The node `sincos` 1.2.3 does not work very well, but it is interesting, because it returns more than one output. In order to call such nodes, LUSTRE syntax allows to write **tuples definition**. Let `s`, `c` and `omega` be tree real variables, `(s, c) = sincos(omega)` is a correct LUSTRE equation defining `s` and `c` to be respectively the first and the second result of the call. The following node compute how the node `sincos` (badly) meets the Pythagore theorem:

```
node pythagore(omega : real) returns (one : real);
var s, c : real;
let
(s, c) =  sincos(omega);
one = s*s + c*c;
tel
```

1.3.2 Tuple expressions

The left hand side of a tuple definition consists of a list of variables. The right hand side of a tuple definition must be an expression denoting a tuple of flows. A tuple expression is either:

- the call of a node returning more than one output,
- an explicit tuple of expressions (for instance `(true->false , 1.0)` is a tuple composed by a boolean flow and a real flow),
- a "`if ... then ... else`" whose two last operands are tuples. The "`if`" operator is the only built-in operator which is polymorphic.

Tuples can be used to "factorise" the definitions, like in the following node `minmax`:

```
node minmax(x, y : int) return (min, max : int);
let
  (min, max) = if (x < y) then (x, y) else (y, x);
tel
```

2 Clocks

Let us consider the following control device: it receives a signal "`set`", and returns a Boolean "`level`" that must be true during "`delay`" cycles after each reception of "`set`". The program is quite simple:

```
node STABLE (set: bool; delay: int) returns (level: bool);
var count: int;
let
   level = (count>0);
```

```
   count = if set then delay
           else if false->pre(level) then pre(count)-1
           else 0;
tel
```

Now, suppose we want the "`level`" to be high during "`delay`" seconds, instead of "`delay`" cycles. The second will be provided as a Boolean input "`second`", true whenever a second elapses. Of course, we can write a new program which freezes the counter whenever the "`second`" is not there:

```
node TIME_STABLE1(set,second:bool; delay:int) returns (level:bool);
var count: int;
let
   level = (count>0);
   count = if set then delay
           else if second then
              if false->pre(level) then pre(count)-1
              else 0
           else (0->pre(count));
tel
```

We can also reuse our node "`STABLE`", calling it at a suitable **clock**, by **filtering** its input parameters. It consists of changing the execution cycle of the node, activating it only at some cycles of the calling program. For the delay be counted in seconds, the node "`STABLE`" must be activated only when either a "set" signal or a "second" signal occurs. Moreover, it must be activated at the initial instant, for initialization purposes. So the activation clock is:

```
   ck = true -> set or second;
```

Now a call "`STABLE((set,delay) when ck)`" will feed an instance of "`STABLE`" with rarefied inputs, as shown by the following table:

`(set,delay)`	(s_1, d_1)	(s_2, d_2)	(s_3, d_3)	(s_4, d_4)	(s_5, d_5)	(s_6, d_6)	(s_7, d_7)
`ck`	true	false	false	true	true	false	true
`(set,delay) when ck`	(s_1, d_1)			(s_4, d_4)	(s_5, d_5)		(s_7, d_7)

According to the data-flow philosophy of the language, this instance of "`STABLE`" will have a cycle only when getting input values, i.e., when `ck` is true. As a consequence, the inside counter will have the desired behavior, but the output will also be delivered at this rarefied rate. In order to use the result, we have first to **project** it onto the clock of the calling program. The resulting node is

```
node TIME_STABLE(set, second: bool; delay: int) returns (level: bool);
var ck: bool;
let
   level = current(STABLE((set,delay) when ck));
   ck = true -> set or second;
tel
```

Here is a simulation of this node:

`(set,delay)`	(tt,2)	(ff,2)	(ff,2)	(ff,2)	(ff,2)	(ff,2)	(ff,2)	(tt,2)	(ff,2)
`(second)`	ff	ff	tt	ff	tt	ff	ff	ff	tt
`ck`	tt	ff	tt	ff	tt	ff	ff	tt	tt
`(set,delay) when ck`	(tt,2)		(ff,2)		(ff,2)			(tt,2)	(ff,2)
`STABLE((set,delay) when ck)`	tt		tt		ff			tt	tt
`current(STABLE (set,delay) when ck))`	tt	tt	tt	tt	ff	ff	ff	tt	tt

3 Arrays and recursive nodes

3.1 Warning

Arrays and recursive nodes have been introduced in LUSTRE as a *syntactic* facility. They do not increase the descriptive power of the language, and the user must be aware of the fact that the compiler LUSTRE-V4 *expands* arrays into as many variables as they have elements, and *unfolds* recursive nodes into regular node instanciations[4]. As a consequence, the array dimensions must be known at compile-time, and so do the parameters controling the recursivity. A **compile-time expression** is either an explicit constant (e.g., `true`, `45`) or an expression made of explicit constants and formal parameters whose actual counterparts are always explicit constants.

3.2 A binary adder

Assume we want to describe a binary adder, working on two 4-bits integers `A` and `B`. Using the basic language, we will have 8 input parameters (one for each bit), and we could write (see Fig. 6):

```
node FIRST_ADD4 (a0,a1,a2,a3: bool; b0,b1,b2,b3: bool)
returns (s0,s1,s2,s3:bool; carry: bool);
var c0,c1,c2,c3: bool;
let
   (s0,c0) = ADD1(a0,b0,false);
   (s1,c1) = ADD1(a1,b1,c0);
   (s2,c2) = ADD1(a2,b2,c1);
   (s3,c3) = ADD1(a3,b3,c2);
   carry = c3;
tel
```

[4]In particular, if the recursivity does not stop, neither does the *compilation* of the program!

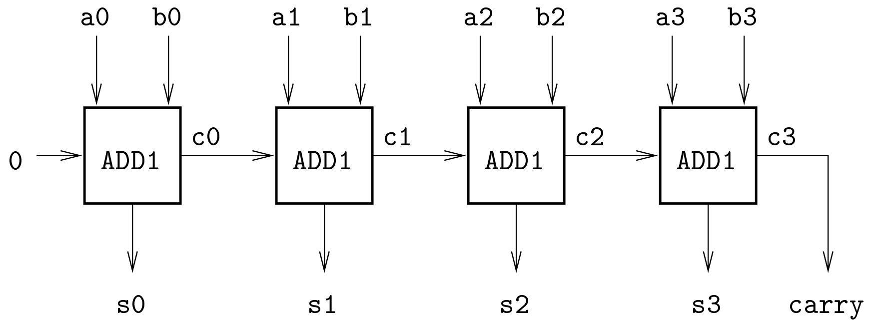


Figure 6: 4-bits adder

where the 1-bit adder ADD takes as input two bits and an input carry, and returns their sum and an output carry:

```
node ADD1(a,b,c_i: bool) returns (s,c_o: bool);
let
  s = a xor b xor c_i;
  c_o = (a and b) or (b and c_i) or (c_i and a);
tel
```

Instead, we can consider A and B as **arrays**, made of 4 Booleans. "bool^4" denotes the type of "arrays of 4 Booleans", indexed from 0 to 3 (the "^" operator here refers to Cartesian power). The adder node becomes (see Fig. 7):

```
node ADD4 (A,B: bool^4) returns (S:  bool^4; carry: bool);
var C: bool^4;
let
   (S[0],C[0]) =  ADD1(A[0],B[0],false);
   (S[1..3],C[1..3]) = ADD1(A[1..3],B[1..3],C[0..2]);
   carry = C[3];
tel
```

The first equation defines the first components of S and C using the standard indexation notation (notice that arrays can only be indexed by compile-time expressions). The second equation is less standard, and makes use of **slicing** and **polymorphism**:

- the notation "S[1..3]" refers to the "slice" of the array S, made of elements 1 to 3 of S, i.e., the array X of type bool^3 such that

  ```
  X[0] = S[1] , X[1] = S[2] , X[2] = S[3]
  ```

- From its declaration, the node ADD1 takes three Booleans as input parameters, and returns 2 Booleans. Here, it is called with three Boolean *arrays* (of the same size) as input parameters, and returns 2 Boolean arrays (of

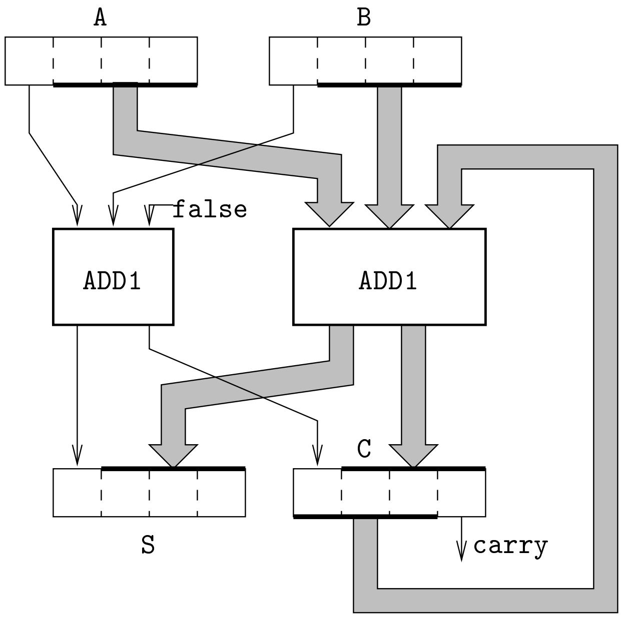


Figure 7: 4-bits adder, with arrays

the same size as the input arrays), obtained by applying `ADD1` componentwise to the input arrays. Such a *polymorphic extension* is available for any operator of the language.

So, the equation "`(S[1..3],C[1..3]) = ADD1(A[1..3],B[1..3],C[0..2])`" stands for

```
(S[1],C[1]) = ADD1(A[1],B[1],C[0]);
(S[2],C[2]) = ADD1(A[2],B[2],C[1]);
(S[3],C[3]) = ADD1(A[3],B[3],C[2]);
```

The *expansion* of this node is the first task of the compiler. It consists, more or less, in translating `ADD4` into `FIRST_ADD4`, by replacing any array element by a variable defined by its own equation.

Now, we can also define a general binary adder, taking the size of the arrays as a parameter:

```
node ADD (const n: int; A,B: bool^n)
returns (S:  bool^n; carry: bool);
var C: bool^n;
let
   (S[0],C[0]) =  ADD1(A[0],B[0],false);
   (S[1..n-1],C[1..n-1]) = ADD1(A[1..n-1],B[1..n-1],C[0..n-2]);
   carry = C[n-1];
tel
```

Such a node cannot be compiled alone. As a matter of fact, the compiler needs an actual value to be given to the parameter `n`, in order to be able to expand the program. A main node must be written, for instance:

```
node MAIN_ADD (A,B: bool^4) returns (S: bool^4);
var carry: bool;
let
  (S, carry) =  ADD(4,A,B);
tel
```

or, better, defining the size as a constant:

```
const size = 4;
node MAIN_ADD (A,B: bool^size) returns (S: bool^size);
var carry: bool;
let
  (S, carry) =  ADD(size,A,B);
tel
```

3.3 The *exclusive* node

Let us show another example making use of arrays: In §4.2 we will need an extension of the "`#`" (exclusion) operator to arrays, i.e., an operator taking a Boolean array `X` as input, and returning "true" if and only if at most one of `X`'s element is true. We use two auxiliary Boolean arrays: An array `EX` whose ith element is true if there is at most one true element in `X[0..i]`, and an array `OR` to compute the cumulative disjunction of `X`'s elements:

$$\begin{aligned} \texttt{EX[i]} &= |\{j \leq i \text{ s.t. } \texttt{X[j]} = \mathit{true}\}| \leq 1 \\ \texttt{OR[i]} &= \bigvee_{j \leq i} \texttt{X[j]} \end{aligned}$$

In other words:

$$\begin{aligned} \texttt{EX[i+1]} &= \texttt{EX[i]} \wedge \neg(\texttt{OR[i]} \wedge \texttt{X[i+1]}) && \text{with } \texttt{EX[0] = true} \\ \texttt{OR[i+1]} &= \texttt{OR[i]} \vee \texttt{X[i+1]} && \text{with } \texttt{OR[0] = X[0]} \end{aligned}$$

One can write the corresponding node as follows:

```
node exclusive (const n: int; X: bool^n) returns (exclusive: bool);
var  EX, OR: bool^n;
let
  exclusive = EX[n-1];
  EX = [true] | (EX[0..n-2] and not(OR[0..n-2] and X[1..n-1]));
  OR = [X[0]] | (OR[0..n-2] or X[1..n-1]);
tel
```

In this program we used two new operators on arrays:

- The constructor "`[.]`": If `X:`τ`^m` and `Y:`τ`^n` are two arrays, "`X|Y`" is their concatenation, of type τ`^(m+n)`.

- The concatenation "|": If `E0, E1, ..., En` are n expressions of the same type τ, then "`[E0, E1, ...,En]`" is the array of type τ`^(n+1)` whose ith element is `Ei` $(i = 0 \ldots n)$.

In the equation defining "`EX`", the Boolean "`true`" has been converted into the *array of one Boolean* "`[true]`" to be given to the concatenation operator.

3.4 The *delay* node with arrays

As a last example with arrays, we will build a general "delay" operator, taking as (static) parameter an integer d ($d \geq 0$) and a Boolean stream `X`, and returning a "delayed" version of `X`, i.e., a Boolean stream `Y` such that $y_n = x_{n-d}$, for any $n > d$. Let us assume y_n = false, for $n \leq d$ (initialization). We use an auxiliary array `A` of type `bool^d`, such that `A[i]`$_n$=`X`$_{n-i}$. The resulting node is:

```
node DELAY (const d: int; X: bool) returns (Y: bool);
var A: bool^(d+1);
let
  A[0] = X;
  A[1..d] = (false^(d)) -> pre(A[0..d-1]);
  Y = A[d];
tel
```

The expression "`false^(d)`" denotes an array of size `d`, all the elements of which are false. It is the initial value of the slice `A[1..d]`. Notice the polymorphic extensions of the operators `->` and `pre`. To compile this program, we have again to call it from a main node:

```
node MAIN_DELAY (A: bool) returns (A_delayed: bool);
let
  A_delayed = DELAY(10, A);
tel
```

However, compiling such a program into an automaton is not a good idea (Try it): The call "`DELAY(10,A)`" creates 10 Boolean memories (instances of a `pre` operator) wich will involve 2^{10} states in the automaton. Instead, one can call the compiler with the option "`-0`",

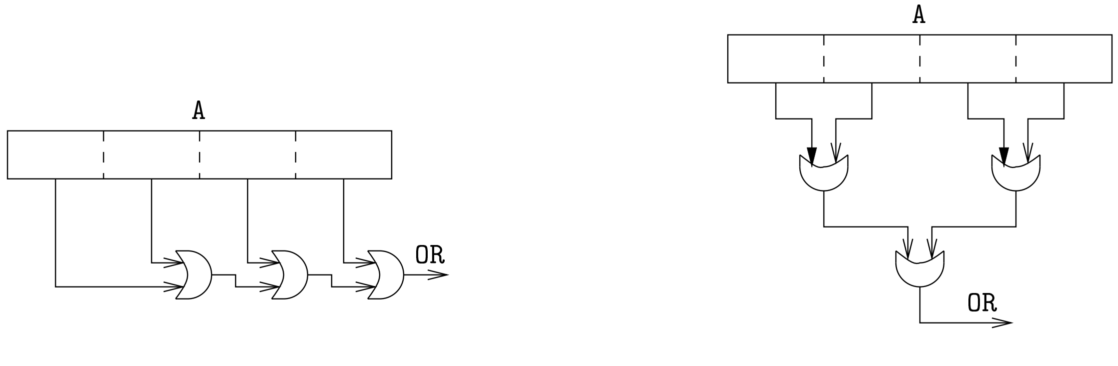
```
    lustre F.lus  MAIN_DELAY -0
```

which produces a single-loop code: The resulting automaton has only one state and one (complicated) transition.

3.5 The *delay* node with recursion

Another solution for the delay operator is to write a **recursive node**:

```
node REC_DELAY (const d: int; X: bool) returns (Y: bool)
let
  Y = with d=0 then X
      else false -> pre(REC_DELAY(d-1,X));
tel
```



(a) (b)

Figure 8: Two nets for array disjunction

The recursivity is controlled by a **static conditional** operator "`with...then...else...`", which is executed at compile-time to unfold the recursivity: The call "`REC_DELAY(3,X)`" will be expanded into something like:

```
Y_3 = false -> pre(Y_2);
Y_2 = false -> pre(Y_1);
Y_1 = false -> pre(Y_0);
Y_0 = X;
```

3.6 Two recursive networks

Recursive nodes can be used to describe complex regular networks. For instance, if we want to compute the disjunction of all the elements of a Boolean array, we can use a linear network (Fig. 8.a):

```
node LIN_OR (const n: int; A: bool^n) returns (OR: bool);
let
  OR = with n=1 then A[0]
       else A[0] or LIN_OR(n-1,A[1..n-1]);
tel
```

or a tree structure (Fig. 8.b):

```
node TREE_OR (const n: int; A: bool^n) returns (OR: bool);
let
  OR = with n=1 then A[0]
       else TREE_OR(n div 2, A[0..(n div 2 -1)]) or
                   TREE_OR((n+1)div 2, A[n div 2 .. n-1]);
tel
```

4 Verification

4.1 Program comparison

4.1.1 Building a comparison node

The simplest case of program verification is the comparison of two purely Boolean programs under some assumption about their environment. For instance, in §1.1.6 we built a first version of the switch, named `SWITCH1`, and we noticed that it worked properly only if its input parameters "`set`" and "`reset`" are never simultaneously true. Then we wrote the more general version `SWITCH`. Now, let us verify that, when "`set`" and "`reset`" are never simultaneously true, the two programs behave the same. For that, we build a **verification program**

```
node verif_switch(set, reset, initial: bool) returns (ok: bool);
var level, level1: bool;
let
   level = SWITCH (set, reset, initial);
   level1 = SWITCH1 (set, reset, initial);
   ok = (level = level1);
   assert not(set and reset);
tel
```

which consists of

- the parallel activation of the two nodes, fed with the same input parameters;
- the definition of a unique Boolean output, "`ok`", comparing the outputs of the nodes
- an **assertion** that the input parameters "`set`" and "`reset`" are never simultaneously true

4.1.2 Verifying with the Lustre compiler

Now, let us compile this program, first using the "data-driven" code generator:

```
lustre F.lus verif_switch -data -v
```

This generator produces the automaton in a straightforward, enumerative, way. The result is drawn in Fig. 9.a. On this automaton (as in the C code `verif_switch.c`) it is clear that the output "`ok`" is always true, and thus that the results of the two nodes are always equal, whatever be the input parameters satisfying the assertion. The result is even more obvious if we use the code generator with the "`-demand`" option, which produces a *minimal* automaton [5, 19]. The result is the one-state automaton shown in Fig. 9.b.

4.1.3 Verifying with the Lesar tool

For more complex verification problems, the verification tool LESAR is more efficient than the compiler. It only traverses the automaton without generating it explicitly. Several algorithms are available:

- `lesar F.lus verif_switch -enum` performs a traversal of the automaton using an enumerative strategy similar to the "data-driven" generation in the compiler; it is the default algorithm.
- `lesar F.lus verif_switch -forward` computes the set of reachables states with a symbolic methods.
- `lesar F.lus verif_switch -backward` computes (in a symbolic way) the set of states violating the property.

One can also check that, without the assertion, the verification fails: The minimal automaton has 4 states, and assigns false to "`ok`" whenever "`set`" and "`reset`" are both true (except in the initial state). The verifier also complains, and when called with the `-diag` option, it also outputs a (shortest) path to a transition where "`ok`" is false.

4.2 Proof of safety properties

Let us consider an extremely simple mutual exclusion algorithm: n processes $p_0, p_1, \ldots, p_{n-1}$ compete for an exclusive resource. The arbiter receives a Boolean array `REQ`, where `REQ[i]` is true whenever the process p_i requests the resource — and returns an array `GRANT`, such that `GRANT[i]` is true whenever the resource is granted to p_i. The arbiter proceeds by letting a token travel around the processes. When the process which has the token is requesting the resource, it takes the resource and keeps the token until it releases the resource.

Let us describe the behavior of the arbitration part attached to one process: it receives the requests and the token as Boolean inputs, and returns the granting and the passed token as Boolean outputs. The token is passed either

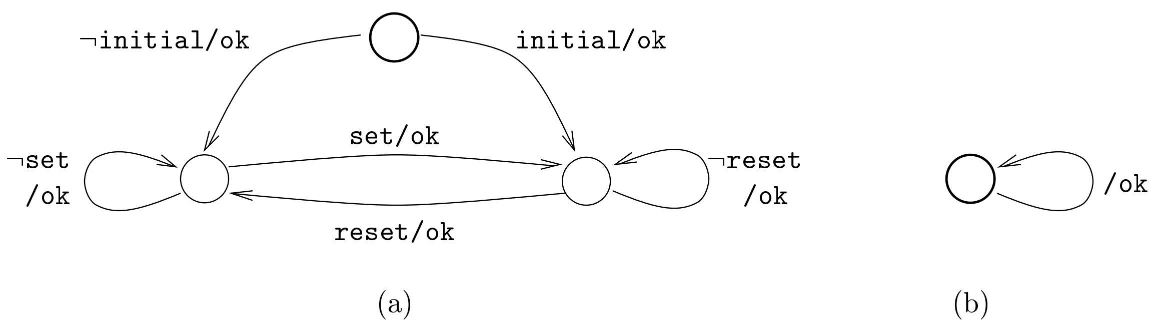


Figure 9: Verification automata

if it was received at the previous step, and the process is not requesting the resource, or if the process stops using the resource. The resource is granted to the process if it requests it when receiving the token, and remains granted to it until it stops requesting (remember the definition of nodes SWITCH and FALLING_EDGE, §1.1.6):

```
node process(request,token:bool) returns (grant,new_token:bool);
let
  grant = SWITCH(token and request,not request,token and request);
  new_token = false ->
              pre(token and not request) or FALLING_EDGE(grant);
tel
```

The whole arbiter is made of a ring of such processes, one of them owning the token at the initial instant:

```
node mutex(const n: int; REQ: bool^n) returns (GRANT: bool^n);
var TOK, NTOK: bool^n;
let
  (GRANT, NTOK) = process(REQ, TOK);
  TOK[0] = true -> pre(NTOK[n-1]);
  TOK[1..n-1] = false^(n-1) -> pre(NTOK[0..n-2]);
tel
```

Now, let us verify that the mutual exclusion is satisfied, i.e, that at most one element of GRANT is true. We write the following verification program, using the node "exclusive" we wrote in §3.3:

```
const nb_proc = 4;
node verif_mutex(REQ: bool^nb_proc) returns (ok: bool);
var GRANT: bool^nb_proc;
let
   GRANT = mutex(nb_proc, REQ);
   ok = exclusive(nb_proc, GRANT);
tel
```

Try the compiler and the verifier on this program for various values of "nb_proc". On this example, the best way to verify the program is to use the "forward symbolic" algorithm.

4.3 Numerical values

Let us consider a program which is supposed to mesure the "speed" of a train. This program has two inputs: the flow "second" comes from some real-time clock, the flow "beacon" each time the train detect a beacon along the way. Normally, the train is supposed to detect a beacon each second. Let diff be the difference between the number of beacons and the number of seconds; if diff is positive, the train is *early*, otherwise it is *late*. In order to avoid oscillation, the program has an hysteresis mecanism: the train becames early, when diff

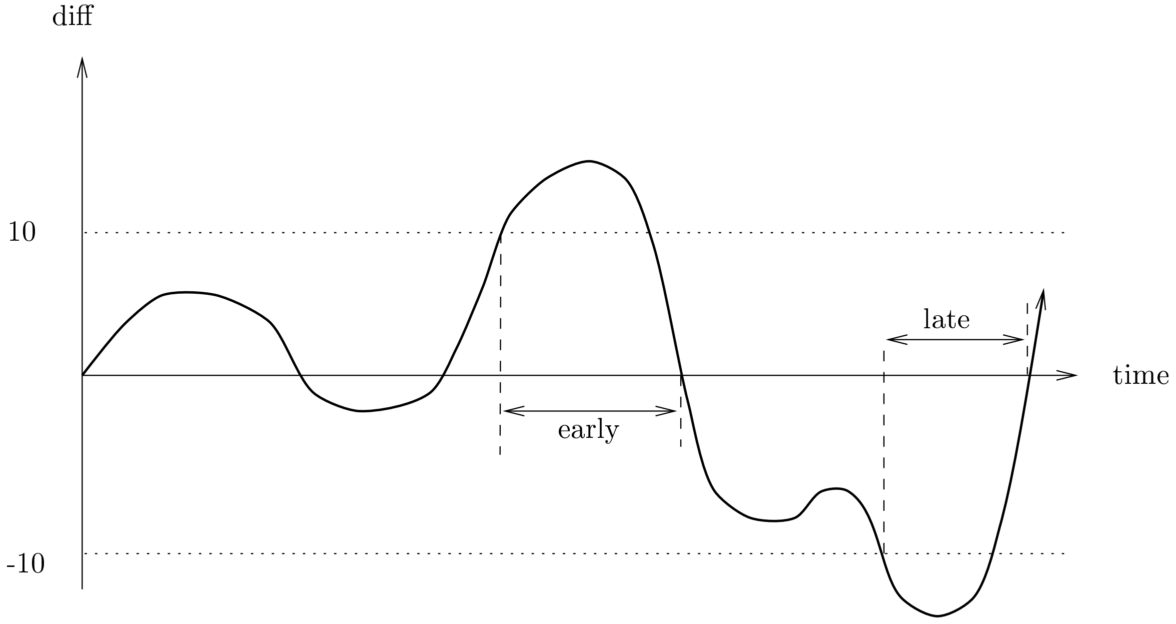


Figure 10: The hysteresis mecanism

becomes greater than 10, and it remains early while `diff` stays greater than 0. Conversely, the train becames late when `diff` becomes less than -10, and remains late while `diff` stays less than 0 (Fig. 10).

The variable `diff` can be defined using the general counter1.2.1: the counter is modified each time either `second` or `beacon` are true, but the increment dynamically depends on those inputs (it can be 1, -1 or 0):

```
node speed(beacon, second : bool) returns (late, early : bool);
var
  diff, incr : int;
let
  incr = if (beacon and not second) then 1
         else if(second and not beacon) then -1
         else 0;
  diff = COUNTER(0, incr, (beacon or second), false);
  early = false -> if pre early then (diff > 0)
                   else (diff >= 10);
  late = false -> if pre late then (diff < 0)
                  else (diff <= -10);
tel
```

A simple property (expected) for this program is that the train cannot be both late and early. The corresponding verification node is, for instance:

```
node verif_speed(beacon, second : bool) returns (ok : bool);
```

```
var late, early : bool;
let
  (late, early) = speed(beacon, second);
  ok = #(late, early);
tel
```

The verification of this program fails wathever the employed method. Using `lesar F.lus verif_speed -v -diag` you get a (complicated) diagnosis involving linear constraints. With a little patience, one can find that this diagnosis only shows unrealisable constraints on numerical values (for instance $(x <= -10) \vee (x >= 10))$. This is quite disapointing, but just remenber that LESAR is a Boolean tool, and does not know anything about numerical properties. A special algorithm has be added into LESAR in order to treat this problem:

```
lesar F.lus verif_speed -poly
```

This algorithm is based on the *enumerative* one, but it use a *polyedra* library to check whether each linear constraints appearing in the automaton is realisable or not.

Another property is that the train cannot pass directly form early to late:

```
node verif_speed2(beacon, second : bool) returns (ok : bool);
var late, early : bool;
let
  (late, early) = speed(beacon, second);
  ok = true -> not late and pre early;
tel
```

Unfortunatly, this property cannot be verified, even with the `-poly` option. This property involves the dynamic behavior of numerical variables, and this problem is much more complicated than the previous one (in fact this problem is undecidable in general). This example shows the limits of the LESAR tool!

References

[1] J-L. Bergerand. LUSTRE: un langage déclaratif pour le temps réel. Thesis, Institut National Polytechnique de Grenoble, 1986.

[2] J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real-time data-flow language. In *1985 Real-Time Symposium, San Diego*, December 1985.

[3] J-L. Bergerand, P. Caspi, N. Halbwachs, and J. Plaice. Automatic control systems programming using a real-time declarative language. In *IFAC/IFIP Symp. 'SOCOCO 86, Graz*, May 1986.

[4] B. Berkane. Vérification des systèmes matériels numériques séquentiels synchrones : Application du langage LUSTRE et de l'outil de vérification Lesar. Thesis, Institut Polytechnique de Grenoble, October 1992.

[5] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[6] C. Buors. Sémantique opérationnelle du langage LUSTRE. DEA Report, University of Grenoble, June 1986.

[7] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[8] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages, POPL'87*, Munchen, January 1987.

[9] A. Girault and P. Caspi. An algorithm for distributing a finite transition system on a shared/distributed memory system. In *PARLE'92, Paris*, July 1992.

[10] A-C. Glory. Vérification de propriétés de programmes flots de données synchrones. Thesis, Université Joseph Fourier, Grenoble, France, December 1989.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification des systèmes réactifs à l'aide du langage flot de données synchrone LUSTRE. *Technique et Science Informatique*, 10(2), 1991.

[12] 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.

[13] N. Halbwachs and F. Lagnier. Sémantique statique du langage lustre - version 3. Technical Report SPECTRE L15, IMAG, Grenoble, February 1991.

[14] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7):523–543, 1992.

[15] 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.

[16] N. Halbwachs, A. Lonchampt, and D. Pilaud. Describing and designing circuits by means of a synchronous declarative language. In *IFIP Working Conference "From HDL Descriptions To Guaranteed Correct Circuit Designs"*, Grenoble, September 1986.

[17] N. Halbwachs and D. Pilaud. Use of a real-time declarative language for systolic array design and simulation. In *International Workshop on Systolic Arrays*, Oxford, July 1986.

[18] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.C. Glory. Specifying, programming and verifying real-time systems, using a synchronous declarative language. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, June 1989.

[19] 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. LNCS 528, Springer Verlag.

[20] D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In M. Joseph, editor, *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Warwick, September 1988. LNCS 331, Springer Verlag.

[21] J. A. Plaice. Sémantique et compilation de LUSTRE, un langage déclaratif synchrone. Thesis, Institut National Polytechnique de Grenoble, 1988.

[22] J. A. Plaice and N. Halbwachs. LUSTRE-V2 user's guide and reference manual. Technical Report SPECTRE L2, IMAG, Grenoble, October 1987.

[23] J. A. Plaice and J-B. Saint. The LUSTRE-ESTEREL portable format. Unpublished report, INRIA, Sophia Antipolis, 1987.

[24] C. Ratel. Etude de la conformité d'un programme LUSTRE et de ses spécifications en logique temporelle arborescente. DEA Report, Institut National Polytechnique de Grenoble, June 1988.

[25] C. Ratel. Définition et réalisation d'un outil de vérification formelle de programmes Lustre: Le système Lesar. Thesis, Université Joseph Fourier, Grenoble, June 1992.

[26] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New Orleans, December 1991.

[27] P. Raymond. Compilation séparée de programmes LUSTRE. Technical Report SPECTRE L5, IMAG, Grenoble, June 1988.

[28] P. Raymond. Compilation efficace d'un langage déclaratif synchrone : Le générateur de code LUSTRE-v3. Thesis, Institut National Polytechnique de Grenoble, November 1991.

[29] F. Rocheteau. Programmation d'un circuit massivement parallèle à l'aide d'un langage déclaratif synchrone. Technical Report SPECTRE L10, IMAG, Grenoble, June 1989.

[30] F. Rocheteau. Extension du langage Lustre et application à la conception de circuits: Le langage Lustre-V4 et le système Pollux. Thesis, Institut National Polytechnique de Grenoble, June 1992.

[31] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits, a hardware implementation of LUSTRE. In REX *Workshop on Real-Time: Theory in Practice, DePlasmolen (Netherlands)*, pages 195–208. LNCS 600, Springer Verlag, June 1991.

[32] F. Rocheteau and N. Halbwachs. POLLUX, a LUSTRE-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II, Chateau de Bonas*, June 1991.

[33] G. Thuau and B. Berkane. Using the language LUSTRE for sequential circuit verification. In *International Workshop on Designing Correct Circuits, Lingby (Denmark)*, January 1992.

[34] G. Thuau and D. Pilaud. Using the declarative language LUSTRE for circuit verification. In *Workshop on Designing Correct Circuits*, Oxford, September 1990.