A TUTORIAL OF LUSTRE

Nicolas HALBWACHS, Pascal RAYMOND

January 24, 2002

Contents

1 Basic language 2
  1.1 Simple control devices ........................................... 2
  1.2 Numerical examples ............................................... 8
  1.3 Tuples ........................................................... 11

2 Clocks 11

3 Arrays and recursive nodes 13
  3.1 Warning .......................................................... 13
  3.2 A binary adder .................................................... 13
  3.3 The exclusive node .............................................. 16
  3.4 The delay node with arrays ..................................... 17
  3.5 The delay node with recursion .................................. 17
  3.6 Two recursive networks ......................................... 18

4 Verification 19
  4.1 Program comparison .............................................. 19
  4.2 Proof of safety properties ...................................... 20
  4.3 Numerical values ................................................ 21

Bibliography 23

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].
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 \(\text{EDGE}\)) will take \(X\) as an input parameter and return \(Y\) as an output parameter. The interface of the node is the following:

\[
\text{node EDGE } (X: \text{bool}) \text{ returns } (Y: \text{bool});
\]

The definition of the output \(Y\) is given by a single equation:

\[
Y = X \text{ and not pre}(X);
\]

This equation defines \(Y\) (its left-hand side) to be always equal to the right-hand side expression \(X \text{ and not pre}(X)\). 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 \(A \text{ and } B\) is the Boolean stream \((a_1 \land b_1, a_2 \land b_2, \ldots, a_n \land b_n, \ldots)\).

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.
The “\texttt{pre}” (for “\texttt{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, \texttt{pre}(A) is the stream \((\texttt{nil}, a_1,a_2,\ldots,a_{n-1},\ldots)\). Its first value is the undefined value \texttt{nil}, and for any \(n > 1\), its \(n\)th value is the \((n-1)\)th value of \(A\).

The “\texttt{->}” (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 “\(A\rightarrow B\)” 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 “\texttt{nil}” value introduced by the \texttt{pre} operator.

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

The complete definition of the node \texttt{EDGE} is the following:

\begin{verbatim}
node EDGE (X: bool) returns (Y: bool);
let
  Y = false -> X and not pre(X);
tel
\end{verbatim}

\section*{1.1.2 Simulating a node}

Let us write the node \texttt{EDGE} in a file \texttt{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):

\tiny

\begin{verbatim}
and or xor not # if...then...else... + - * / div mod = <> < <= > >=
int real
\end{verbatim}

\footnote{A Lustre constant denotes an infinite stream of a same value. Pre-defined constants are \texttt{false}, \texttt{true}, and immediate arithmetic values. For instance, the expression 3.14 denotes (3.14, 3.14, 3.14, ...).}
The graphical simulator is based on an interpreter 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:

```bash
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:

```bash
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:

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

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

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

```bash
######## 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.

```c
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
be the input, but, depending on the value of $X$, the next state will be either 1 (corresponding to $\text{pre } X = \text{false}$) or 2 (corresponding to $\text{pre } X = \text{true}$). 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:

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

  We get:

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

---

Footnote:

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:

\[
\text{node FALLING\_EDGE (X: bool) returns (Y: bool);} \\
\text{let Y = EDGE(not X);} \\
\text{tel}
\]

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:

\[
\text{node SWITCH1 (set, reset, initial: bool) returns (level: bool);} \\
\text{let level = initial -> if set then true} \\
\text{else if reset then false} \\
\text{else pre(level);} \\
\text{tel}
\]

which specifies that the “level” is initially equal to “initial”, and then for-
ever,

- 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
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 $C = 0 \rightarrow \text{pre } C + 1$; 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” occurrences in a boolean flow $X$:

$C = 0 \rightarrow \text{if } X \text{ then } (\text{pre } C + 1) \text{ else } (\text{pre } C)$;

This definition does not meet exactly the specification, since it ignores the initial value of $X$. A well initialized counter of $X$ occurrences 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 additional 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

\[ F_n = f(x_n) \text{ and } x_{n+1} = x_n + \text{STEP}_{n+1} \]

It computes a real-valued stream Y, such that

\[ Y_{n+1} = Y_n + (F_n + F_{n+1}) \times \text{STEP}_{n+1}/2 \]

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))\times\text{STEP})/2.0;
tel

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

```plaintext
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);
end
```

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 \( n \)th value of \( \sin \) needs the \( n \)th value of \( \cos \), and conversely. We have to cut the dependence loop, introducing a “\texttt{pre}” operator:

```plaintext
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);
end
```

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 three real variables, $(s, c) = \text{sincos}(\omega)$ 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 Pythagorean theorem:

```plaintext
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:

```plaintext
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:

```plaintext
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:

<table>
<thead>
<tr>
<th>(set,delay) when ck</th>
<th>(s1,d1)</th>
<th>(s2,d2)</th>
<th>(s3,d3)</th>
<th>(s4,d4)</th>
<th>(s5,d5)</th>
<th>(s6,d6)</th>
<th>(s7,d7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ck</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>(set,delay)</th>
<th>(ff,2)</th>
<th>(ff,2)</th>
<th>(ff,2)</th>
<th>(ff,2)</th>
<th>(ff,2)</th>
<th>(tt,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(second)</td>
<td>tt</td>
<td>ff</td>
<td>tt</td>
<td>ff</td>
<td>tt</td>
<td>ff</td>
</tr>
<tr>
<td>ck</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(set,delay)</td>
<td>(tt,2)</td>
<td>(ff,2)</td>
<td>(ff,2)</td>
<td>(ff,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>when ck</td>
<td>tt</td>
<td>tt</td>
<td>ff</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STABLE((set,delay)</td>
<td>tt</td>
<td>tt</td>
<td>ff</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>when ck</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>current(STABLE</td>
<td>tt</td>
<td>tt</td>
<td>tt</td>
<td>tt</td>
<td>ff</td>
<td>ff</td>
</tr>
<tr>
<td>(set,delay)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>when ck</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 instantiations. As a consequence, the array dimensions must be known at compile-time, and so do the parameters controlling 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):

```plaintext
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


- 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
the same size as the input arrays), obtained by applying ADD1 component-wise 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

\[
\begin{align*}
(S[1], C[1]) &= ADD1(A[1], B[1], C[0]); \\
(S[2], C[2]) &= ADD1(A[2], B[2], C[1]); \\
(S[3], C[3]) &= ADD1(A[3], B[3], C[2]);
\end{align*}
\]

The *expansion* of this node is the first task of the compiler. It consists, more or less, in translating \texttt{ADD4} into \texttt{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:

```plaintext
node ADD (const n: int; A, B: bool^n) returns (S: bool^n; carry: bool);
var C: bool^n;
let
\[
\begin{align*}
(S[0], C[0]) &= ADD1(A[0], B[0], false); \\
(S[1..n-1], C[1..n-1]) &= ADD1(A[1..n-1], B[1..n-1], C[0..n-2]); \\
carry &= C[n-1];
\end{align*}
\]
tel
```

Figure 7: 4-bits adder, with arrays
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:

```plaintext
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:

```plaintext
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 \( \text{EX} \) whose \( i \)th element is true if there is at most one true element in \( X[0..i] \), and an array \( \text{OR} \) to compute the cumulative disjunction of \( X \)’s elements:

\[
\text{EX}[i] = \{j \leq i \text{ s.t. } X[j] = \text{true} \} \leq 1
\]

\[
\text{OR}[i] = \bigvee_{j \leq i} X[j]
\]

In other words:

\[
\text{EX}[i+1] = \text{EX}[i] \land \neg(\text{OR}[i] \land X[i+1]) \text{ with } \text{EX}[0] = \text{true}
\]

\[
\text{OR}[i+1] = \text{OR}[i] \lor X[i+1] \quad \text{with } \text{OR}[0] = X[0]
\]

One can write the corresponding node as follows:

```plaintext
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 \( X:\tau^m \) and \( Y:\tau^n \) are two arrays, “\( X|Y \)” is their concatenation, of type \( \tau^{(m+n)} \).
- The concatenation "\"\": If E₀, E₁, ..., Eₙ are 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...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 ≥ 0) and a Boolean stream X, and returning a "delayed" version of X, i.e., a Boolean stream Y such that yₙ = xₙ₋d, for any n > d. Let us assume yₙ = false, for n ≤ d (initialization). We use an auxiliary array A of type bool^[d], such that A[i] = X[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) which will involve 2¹⁰ 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
```
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

```plaintext
def node verify_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:

```plaintext
lustre F.lus verify_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 verify_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 reachable states with a symbolic method.

- `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](image)
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):

```plaintext
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);
```

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

```plaintext
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]);
```

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:

```plaintext
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);
```

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 measure 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 detects 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 mechanism: the train becomes early, when diff
Figure 10: The hysteresis mechanism

becomes greater than 10, and it remains early while diff stays greater than 0. Conversely, the train becomes 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):

```plaintext
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:

```plaintext
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 wathether the employed method. Using leasar 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 $(x \leq -10) \lor (x \geq 10)$). This is quite disappointing, but just remember that Lesar is a Boolean tool, and does not know anything about numerical properties. A special algorithm has been 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

Unfortunately, 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


