LUSTRE LANGUAGE

REFERENCE MANUAL

Versions 3, 3+, 4, 5

September 18, 1997

Contents

Foreword

This document describes the LUSTRE language, as it is implemented in the industrial environment SCADE, developped by Verilog, and in the academic freeware LUSTRE/LESAR, developped by Verimag/CNRS, from Paul Caspi's ideas.

The design of the base of the language took place in 1984-86 [Ber86, BCH+85, CPHP87] at Vérimag (formerly IMAG/LGI-Spectre). The first two versions of the academic compiler were written in Le-Lisp by John Plaice [Pla88].

At the same time, LUSTRE became the core language of the SAGA programming environment [BP88], developped and used in industrial applications by Schneider-Electric (formerly Merlin-Gerin). Then, in cooperation with Vérimag, Schneider-Electric, and Aérospatiale, Verilog designed a commercial version of SAGA, made of the graphical environment SAO+, and the code generator SAGA-CG. At the beginning of this industrial design, the LUSTRE language was stabilized in a version called V3 [HL91].

Now, in order to satisfy users'demand, the industrial compiler incorporated some extensions, like type refinements. We call V3+, the version of the language actually implemented by SAGA-CG [Ver96].

On the academic side, important extensions were added, to cope with a project of silicon compiling [RH91], in particular array types. The corresponding compiler, called LUSTRE-V4, was developped by Frédéric Rocheteau [Roc92], for the front-end, and by Pascal Raymond [Ray91], for the back-end. Finally, a new front-end is under development by Fabienne Lagnier and Nicolas Halbwachs, which will be called V5.

One goal of this document is to describe the "live" versions of the language (V3+, V4, and V5), starting from V3, to highlight the differences, and to be used as a basis for the design of a new stabilized version.

Chapter 1

Overview of the Language

Contents of the chapter

1.1 Introduction

This manual presents the LUSTRE language, a synchronous language based on the dataflow model and designed for the description and verification of real-time systems. In this chapter, we present the general framework that forms the basis of the language: the synchronous model, the dataflow model, and the synchronous dataflow model. Then we introduce the main features of the language through some simple examples.

The end of the chapter gives some basic elements for reading the rest of the document: it makes precise the metalanguage used to describe the syntax throughout the document and describes the lexical rules of the language.

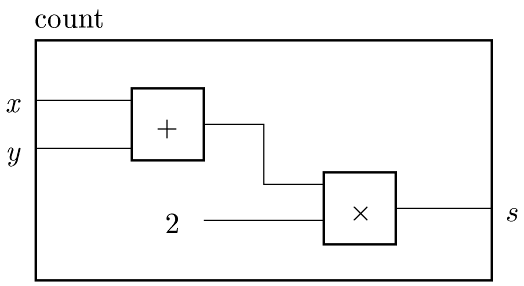
1.1.1 Synchronous Model

The synchronous model was introduced to provide abstract primitives assuming that a program reacts instantaneously to external events. Each output of the program is assigned a precise date in relation to the flow of input events.

A discrete time scale is introduced. The time granularity is considered to be adapted a priori to the time constraints imposed by the dynamics of the environment on which the system is to react. It is verified a posteriori. Each instant on the time scale corresponds to a computation cycle, i.e., in the case of LUSTRE, to the arrival of new inputs. The synchrony hypothesis presumes that the means of computation are powerful enough for the level of granularity to be respected. In other words, the time to compute outputs in function of their inputs is less than the level of granularity on the discrete time scale. Consequently, outputs are computed and inputs are taken into account "at the same time" (with respect to the discrete time scale).

1.1.2 Dataflow Model

The dataflow model is based on a block diagram description. A block diagram can be described either graphically, or by a system of equations. A system is made up of a network of operators acting in parallel and in time with their input rate.



Graphic view of a network

```
node count (x,y: int) returns (s: int);
let
    s = 2*(x+y);
tel
```

Textual view of the same network

This model provides the following advantages:

- maximal use made of parallelism (the only constraints are dependencies between data),
- mathematical formalization (formal verification methods),
- program construction and modification,
- ability to describe a system graphically.

1.1.3 Synchronous Dataflow Model

The synchronous dataflow approach consists in adding a time dimension to the dataflow model. A natural way of doing this is to associate time with the rate of dataflow. The entities manipulated can naturally be interpreted as functions of time. A basic entity (or flow) is a couple made up of:

- a sequence of values of a given type,
- a clock representing a suite of graduations (on the discrete time scale).

A flow takes the t^{th} value in its sequence at the t^{th} instant of its clock. For instance, the description given by the previous diagram expresses the following relation:

for any instant $t, s_t = 2 * (x_t + y_t)$

The time dimension is therefore an underlying feature in any description of this type of model. LUSTRE is a synchronous language based on the dataflow model. The synchronous aspect introduces constraints on the type of input/output relations that can be expressed: the output of a program at a given instant cannot depend on future inputs (causality) and can depend on only a bounded number of inputs (each cycle can memorize the value of the previous input).

1.1.4 Building a Description

A LUSTRE program describes the relations between the outputs and inputs of a system. These relations are expressed using operators, auxiliary variables, and constants. The operators can be:

- basic operators,
- more complex, user-defined, operators, called nodes.

Each description written in LUSTRE is built up of a network of nodes. A node describes the relation between its input and output parameters using a system of equations. Nodes correspond to the functions of the system and allow complex networks to be built simply by passing parameters.

The synchrony hypothesis presumes that each operator in the network responds to its inputs instantaneously.

A LUSTRE description is a list of type, constant and node declarations. The declarations can occur in any order.

The *functional behavior* of an application described in LUSTRE does not depend on the clock cycle. It is therefore possible to perform a functional validation of the application (ignoring the time validation) by testing it on a machine different from the target machine (on the development machine in particular).

Time validation is performed on the target machine. If the computation time is less than the time interval between two instants on the discrete time scale, it can be considered to be zero, and the synchrony hypothesis is satisfied. The interval between two instants on the scale is imposed by the requirements report. Computation time depends on software and hardware performance. LUSTRE is a language describing systems with a deterministic behavior from both a functional and a time point of view.

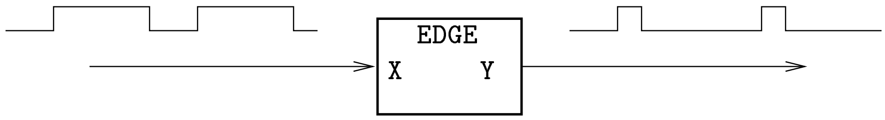


Figure 1.1: A Node

1.2 Basic Features

In this section, we present informally the main basic features of the language, through several simple examples.

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

1.2.1 Simple control devices

As a very first example, let us consider a Boolean flow $X = (x_1, x_2, \ldots, x_n, \ldots)$. We want to define another Boolean flow $Y = (y_1, y_2, \ldots, y_n, \ldots)$ 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 (see Fig. 1.1). 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 flows: if $\mathsf{A} = (a_1, a_2, \ldots, a_n, \ldots)$ and $\mathsf{B} = (b_1, b_2, \ldots, b_n, \ldots)$ are two Boolean flows, then "A **and** B" is the Boolean flow $(a_1 \wedge b_1, a_2 \wedge b_2, \ldots, a_n \wedge b_n, \ldots)$. Most usual operators are available in that way, and are called "*data-operators*".

- The "**pre**" (for "*previous*") operator allows one to refer at cycle n to the value of a flow at cycle $n-1$: if $\mathsf{A} = (a_1, a_2, \ldots, a_n, \ldots)$ is a flow, **pre**(A) is the flow $(nil, a_1, a_2, \ldots, a_{n-1}, \ldots)$. Its first value is the undefined value *nil*, and for any $n > 1$, its nth value is the $(n-1)$th value of A

As a consequence, if $\mathsf{X} = (x_1, x_2, \ldots, x_n, \ldots)$, the expression "X **and not pre**(X)" represents the flow $(nil, x_2 \wedge \neg x_1, \ldots, x_n \wedge \neg x_{n-1}, \ldots)$. Now, since its value at the first cycle is *nil* the program would be rejected[1] by the compiler: it indicates that the output lacks an initialization. A correct equation could be:

```
Y = false -> X and not pre(X);
```

Here, "**false**" denotes the *constant* flow, always equal to false. We have used the second specific LUSTRE operator, "->" (read "*followed by*") which defines initial values. If $\mathsf{A} = (a_1, a_2, \ldots, a_n, \ldots)$ and $\mathsf{B} = (b_1, b_2, \ldots, b_n, \ldots)$ are two flows of the same type, then "A -> B" is the flow $(a_1, b_2, \ldots, b_n, \ldots)$, equal to A at the first instant, and then forever equal to B.

So, 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
```

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

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

The EDGE node is of very common usage for "deriving" a Boolean flow, 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 flow, 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,

[1] Or, at least, a warning would be returned.

- 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*: its current value is defined by means of its previous value).

Moreover, if this node is intended to be used only in contexts where inputs set and reset are never true together, such an *assertion* can be specified:

```
assert not (set and reset);
```

Otherwise, 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 flow representing a signal, then the call

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

will compute the always true flow: “state” is initialized to true, and never changes because the “set” formal parameter has been given priority. 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
```

1.2.2 Numerical examples

Recursive sequences are very easy to define in LUSTRE. For instance, the equation “N = 0 -> **pre** N + 1;” defines the sequence of natural numbers. Let us complexify this definition to build an integer sequence, whose value is, at each instant, the number of occurrences of the “true” value of a Boolean flow X:

```
N = 0 -> if X then pre N + 1 else pre N;
```

This definition does not exactly meet the specification, since it ignores the initial value of X. A well-initialized counter could be:

```
PN = 0 -> pre N;
N = if X then PN + 1 else PN;
```

or, simply

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

or even

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

Let us write a more general operator, with additional inputs:

- an integer init, which is the initial value of the counter;
- an integer incr, which must be added to the counter when X is true;
- a Boolean reset, which reset the counter to the value init, whatever be the value of X.

The complete definition of this operator is the following:

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

This node can be used to define, e.g., the sequence of odd integers:

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

or the sequence of integers modulo 10:

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

Our next 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 flows F and STEP, such that

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

It computes a real-valued flow Y, such that

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

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
```

One can try to connect two such integrators in loop 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 = omega * integrator(-sin,0.1,1.0);
tel
```

Called on this program, the compiler would complain 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 = omega * integrator(0.0 -> -pre(sin),0.1,1.0);
tel
```

1.2.3 Multiple Equation

The node sincos above does not work very well, but it is interesting since it returns more than one output. To call such a node, Lustre allows *multiple definitions* to be written. Let s, c, omega be three real variables, then

```
(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.

So, the left-hand side of an equation can be a list of variables. The right hand side of such a multiple definition must denote a corresponding list of expressions, of suitable types. It can be

- a call to a node returning several outputs
- an explicit list
- the application of a *polymorphic* operator to a list

For instance, the equation

```
(min, max) = if a<b then (a,b) else (b,a);
```

directly defines min and max to be, respectively, the least and greatest value of a and b.

1.2.4 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 to 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

1.3 Syntax Description Metalanguage

The syntax rules of the language will be given in EBNF. In this section, we make precise the used notation.

- Syntax rules have the following form

  NON-TERMINAL ::= TERM_1 | ... | TERM_n

  meaning that the syntactic unit "NON-TERMINAL" can derive on each of the TERM_i, $i = 1 \ldots n$.
- Syntactic units are noted $\langle$unit$\rangle$.
- Syntactic terms are regular expressions made of

- Syntactic units,
- Keywords, noted **keyword**,
- Special characters, noted $\boxed{\text{char}}$,
- Operators like sequence (simple concatenation), optional presence (noted [...]) and 0 or more iteration (noted {...}).

For instance, the following (simplified) rule

⟨number⟩ ::= ⟨digit⟩ { ⟨digit⟩ } [$\boxed{.}$ { ⟨digit⟩ }]

expresses that a number is an arbitrary, non-empty, sequence of digits, optionally followed by a period and an arbitrary sequence of digits.

1.4 Lexical Aspects

This section describes the way the basic objects of the language must be written.

1.4.1 Numbers

There a two kinds of numbers in LUSTRE: integers and real. Their syntax follows usual rules:

⟨int_value⟩ ::= ⟨digit⟩ { ⟨digit⟩ }

⟨real_value⟩ ::= ⟨int_part⟩ ⟨decimal_part⟩ [⟨exponent⟩]
| ⟨int_part⟩ ⟨exponent⟩

⟨int_part⟩ ::= ⟨int_value⟩
⟨decimal_part⟩ ::= $\boxed{.}$ [⟨int_value⟩]
⟨exponent⟩ ::= $\boxed{\text{E}}$ [⟨sign⟩] ⟨int_value⟩
⟨sign⟩ ::= $\boxed{+}$ | $\boxed{-}$
⟨digit⟩ ::= $\boxed{0}$ | $\boxed{1}$ | $\boxed{2}$ | $\boxed{3}$ | $\boxed{4}$ | $\boxed{5}$ | $\boxed{6}$ | $\boxed{7}$ | $\boxed{8}$ | $\boxed{9}$

1.4.2 Identifiers

Identifiers follow the syntax of C: they are arbitrary sequences of alphanumeric characters (including the "underscore"), starting with a letter or and "underscore". The language is case-sensitive (i.e., `A_1` is not the same as `a_1`). There is no restriction on the length of identifiers.

⟨ident⟩ ::= ⟨C_letter⟩ ⟨alpha_num⟩
⟨C_letter⟩ ::= $\boxed{\_}$ | $\boxed{\text{a}}\ldots\boxed{\text{z}}$ | $\boxed{\text{A}}\ldots\boxed{\text{Z}}$
⟨alpha_num⟩ ::= ⟨C_letter⟩ | ⟨digit⟩

In V4 and V5, there is no limitation on the length of identifiers. In V3+, the maximum length is a parameter of the compiler.

In V4 and V5, the same identifier may be used for objects of different kinds (for instance, a type and a variable). In V3+, it may raise problems with the C code produced by the compiler, since identifiers are reported in C as they are in the source.

It is forbidden to use any LUSTRE keyword as an identifier.

In V4 and V5, **true**, **false**, **bool**, **int**, and **real** are not keywords, but predefined identifiers. They are treaten as keywords in V3+.

1.4.3 Keywords and Special Characters

Here is the list of LUSTRE keywords. Some of them are not used in the current versions, but are reserved to make possible up-compatibility with ongoing versions:

and	**as**	**assert**	**case**	**condact**	**const**	**current**
default	**deferred**	**div**	**else**	**external**	**function**	**if**
implementation	**imported**	**include**	**let**	**mod**	**node**	**not**
of	**or**	**pre**	**refinement**	**refines**	**returns**	**step**
tel	**type**	**then**	**var**	**when**	**with**	**xor**

The following special characters are used in (some versions of) the language:

`|` `[` `]` `{` `}` `(` `)` `;` `:` `.` `,` `+`

`-` `*` `/` `^` `#` `=` `<` `>` `->` `>=` `=>` `<=`

`<>` `..` `%` `--` `/*` `*/` `(*` `*)`

1.4.4 Comments

There are two kinds of comments in LUSTRE, the syntax of which sometimes depend on the version:

- Each text between `--` and the end of the line, is a comment
- In V3+, each text between `/*` and `*/`, is a comment
- in V4 and V5, each text between `(*` and `*)`, is a comment

Comments of the first kind can be nested into ones of the second kind.

1.4.5 Including files

A LUSTRE description may spread over several files. A file containing any declarable LUSTRE entity can be included in the current work file using the **include** directive. The following syntax must be respected to include a file:

⟨include_line⟩ ::= **include** `"` ⟨file_name⟩ `"`

This line will be replaced by the contents of the file ⟨file_name⟩.

Example
File `exple.lus`:

```
include "exple_types.lus" ;
node N (e : T) returns (s : T) ;
let
...........
tel
```

File `exple_types.lus`:

```
type T = .... ;
...........
```

Chapter 2

Programs

Contents of the chapter

2.1 Programs

A LUSTRE program is a list of declarations. Recall that any part of this list can be imported from another file, using an "**include**" command (cf. §1.4.5).

Four kinds of objects can be declared:

- Constants: A constant declaration either associates an identifier to an immediate value, or declares a constant imported from the host language (see § 2.2).
- Types: A type declaration either associates an identifier to a type, or declares an external type, imported from the host language (see § 2.2).
- Functions: A function declaration gives the parameter types of a function described in the host language (see § 2.2).
- Nodes: Nodes are the active entities of a LUSTRE program. Chapter 5 describes the header of a node, while its body is described in Chapters 6 and 7.

The order of these declarations is meaningless. All the declared objects are global to the program.

2.1.1 Syntax

⟨program⟩ ::= { ⟨declaration⟩ }
⟨declaration⟩ ::= ⟨constant_declarations⟩
| ⟨type_declarations⟩
| ⟨function_decl⟩
| ⟨node_decl⟩

A constant (resp., type) declaration starts with the keyword **const** (resp., **type**), followed by one or several declarations. Each node (resp., function) declaration has its own starting keyword **node** (resp., **function**)

⟨constant_declarations⟩ ::= **const** ⟨const_decl⟩ { ⟨const_decl⟩ }
⟨const_decl⟩ ::= (see Chapter 4)
⟨type_declarations⟩ ::= **type** ⟨type_decl⟩ { ⟨type_decl⟩ }
⟨type_decl⟩ ::= (see Chapter 3)
⟨function_decl⟩ ::= (see Chapter 5)
⟨node_decl⟩ ::= (see Chapter 5)

[V3+]
In V3+, there is an other kind of declaration: the declaration of refinements:

⟨declaration⟩ ::= ⟨constant_declarations⟩
| ⟨type_declarations⟩
| ⟨refinement_declarations⟩
| ⟨function_decl⟩
| ⟨node_decl⟩

⟨refinement_declarations⟩ ::= **refinement** ⟨refinement_decl⟩ { ⟨refinement_decl⟩ }

⟨refinement_decl⟩ ::= (see Chapter 3)

[end V3+]

2.1.2 An Example

We give below a small, but complete, example of a LUSTRE program, showing all kinds of objects that can be declared at the upper level[1].

[1] There are some syntactic differences in V3+, where, for instance, the declaration of imported types would contain the keyword **imported**.

```
-- three type declarations
type TIME, DAY;
type DISPLAY = [TIME, DAY];
-- some constant declaration
const initial_time: TIME;
const Mo, Tu, We, Th, Fr, Sa, Su: DAY;
-- two function declarations
function Increment_Time (time: TIME) returns (new_time: TIME; change_day: bool);
function ToMorrow (today: DAY) returns (tomorrow: DAY);
-- a node declaration
node Timer(second: bool) returns (display: DISPLAY);
var time: TIME; day: DAY; change_day: bool;
let
    display = [time, day];
    (time, change_day) =
            (initial_time, false) -> if second then Increment_Time(pre time)
                                     else pre(time,change_day);
    day = Mo ->  if change_day then ToMorrow(pre day)
                 else pre day;
tel
```

2.2 External Objects

Lustre is not a general purpose language. A Lustre program is intended to be embedded into a more general program, which is in charge, in particular, of sampling inputs and displaying outputs of the Lustre program. The embedding program is generally written in another language (e.g., C), called the *host language*. Some objects must be shared between the Lustre program and the embedding program. Moreover, some objects, like complex data types or functions, are easier to describe in the general-purpose host language. This is why Lustre allows constants, types, and functions, to be imported from the host language.

Chapter 3

Types

Contents of the chapter

3.1 Introduction

LUSTRE is a strongly typed language, meaning that all variables and constants have a fixed type. Variables and constants are given a type when they are declared. A type is then associated with each expression in the language. In this way semantic rules can be checked by the compiler.

Types appear in constant and variable declarations (see Chapter 4). Types can also be declared.

The type mechanism makes the main differences between the various versions of the language.

LUSTRE provides:

- predefined types: integer, boolean, real,
- tuple types (see §3.3),
- types declared by the user, and known by their identifier.

- Expressions can be of type "list" (see §3.4, §6.2.4), but such a type cannot be defined, or used to type a variable nor a constant.

$$\langle\text{type}\rangle ::= \langle\text{predefined\_type}\rangle \mid \langle\text{tuple\_type}\rangle \mid \langle\text{ident}\rangle$$

[V4 and V5]
V4 and V5 also provide arrays (see §3.5)

$$\langle\text{type}\rangle ::= \langle\text{predefined\_type}\rangle \mid \langle\text{tuple\_type}\rangle \mid \langle\text{ident}\rangle \mid \langle\text{array\_type}\rangle$$

[end V4 and V5]

3.2 Predefined Types

Predefined types are boolean (**bool**), integer (**int**), and real (**real**). They cannot be redefined.

Example

- x : **int** ; defines the variable x to be of integer type;
- the expression x+1 is also of integer type.

3.3 Tuple Types

The concept of tuple types was introduced to allow the data hierarchy to be decomposed.

A tuple of length n is an element of the cartesian product of n types. Tuples are therefore type expressions. They are written between brackets.

[V5]
In V5, they are written between braces, to distinguish them from arrays.

[end V5]

Example
[**int**, **int**, **bool**] is a tuple type of flows, each value of which is made of 3 fields, the first two being of type **int**, and the third one being a boolean.

3.3.1 Nesting Tuples

Tuple types can be nested.

Example
[**int**, [**bool**, **real**]] is a tuple of the length 2, made up of 2 type expressions:

- a predefined type (**int**),
- a tuple of 2 predefined types ([**bool**, **real**]).

3.3.2 Variables and Expressions of Tuple Type

Variables can be declared as being of tuple type:

Example

x : [**int**, **bool**, [**int**, **real**]] ;

Apart from such variables, expressions may be of tuple type. A node can take tuples as input or output parameter.

Example

- If x, y, and z are variables of integer type, the tuple [x, y, z] is an expression of type [**int**, **int**, **int**]. (see §7.3.3, for a description of the tuple operator [.,.,]).
- If x is of type **int**, y of type **real**, z of type **bool**, then the expression [x,[y,z]][2] is of the type [**real**, **bool**] (see §7.3.3, for a description of the selection operator [.]).
- If x is of type **int**, if N returns a unique output parameter, of type [**real**, **bool**], then [x,N(a,b)] is an expression of type [**int**, [**real**, **bool**]].

3.4 List Types

Some expressions in LUSTRE can represent lists: The basic need for such expressions is the fact that nodes can have several outputs, and that a node call is always an expression. For convenience, we consider that an expression can be of type "list", even if it is impossible to declare a variable of type "list"[1].

Throughout this document, a type "list" will be noted as a list of types between parentheses.

[1]In some terminology, "list" types would be said to be "expressible" but not "denotable".

List nesting is ignored: If a list contains a type "list", it is considered as flatenned:

$$(\tau_1, (\tau_2, \tau_3)) \;=\; (\tau_1, \tau_2, \tau_3)$$

The actual parameters of a node are often considered to be a single expression of type "list". This way, a node can always be considered as taking an expression as input and returning an expression.

Only nodes, functions, and the following polymorphic operators of the LUSTRE language, can be applied to expression of type "list":

if ... **then** ... **else** ..., **pre**, **->**, **when**, **current**

Example
If a, b are of type **int**, if c is of type **bool**, if N is a node taking two integers and a boolean as inputs, and returning an integer and a boolean — i.e., N : (**int**,**int**,**bool**) $\rightarrow$ (**int**,**bool**) —, then

- N(a,b,c) is an expression of type (**int**,**bool**).
- **if** a< b **then** (a,b) **else** (b,a) is a correct expression, of type (**int**,**int**).
- **pre**((a,b) **when** c) is a correct expression, of type (**int**,**int**).
- **if** c **then** (a, **true**) **else** N(a,b,c) is a correct expression[2], of type (**int**,**bool**).
- N(a,N(a,b,c)) is a correct call to N, since the innermost call to N is an expression of type (**int**,**bool**), so the outermost call is applied to an expression of type (**int**,(**int**,**bool**)), which is flatenned into (**int**,**int**,**bool**), the expected input type of N.

[V4 and V5]

3.5 Array Types in V4 and V5

Arrays have been introduced in V4, to make easier the description of circuits [RH91]. In V4, arrays are tuples, all the elements of which are of the same type; specific operators are provided to deal with arrays. In V5, tuples and arrays have nothing to do with each other; this is why tuple are noted between braces instead of brackets.

3.5.1 Simple arrays

An array type is considered to be a cartesian power of its element type. This is why the type "array of n elements of type τ" is noted τ^n.

[2] This is not true in V3+, where calls to nodes with several outputs are not allowed to be embedded in expressions.

⟨array_type⟩ ::= ⟨type⟩ $\boxed{\hat{}}$ ⟨int_constant⟩

Here, the size ⟨int_constant⟩ must be an integer constant *known at compile time*: it can be

1. an integer, or
2. a defined constant (see §4.2.1), or
3. an input constant parameter to a node, which will be instanciated with a formal parameter of one of the preceding kinds, or
4. a constant expression made of the previous kinds of objects, and data operators.

Example

```
const
    size = 32;
    unknown: int
type T;
...
node N (const n: int) returns (x: int);
var V: T ^(2*n);
let
    ...
tel ...
...
X = N(size+1); --  is correct
X = N(unknown); --  is rejected
...
```

In the above example, the node N takes as input parameter, a constant n, which is used to define the size of a local array V. This node may only be called with an actual parameter which is known at compile time: the first call, where the value of the parameter size is known to be 32, is correct; the second call makes the program to be rejected by the compiler.

3.5.2 Arrays of arrays

Elements of an array can be of array type.

Example
`int^3^5` is the type of arrays of 5 elements of type array of 3 integers.

[end V4 and V5]

3.6 Type Declarations

Types can be declared by the user. Such a declaration can

- Associate an identifier with a type
- Declare a type imported from a host language (C).

Type declarations appear at the upper level of the program (see Chapter 2). They are global to the whole program. Types can be declared in any order.

⟨type_declarations⟩ ::= **type** ⟨type_decl⟩ { ⟨type_decl⟩ }

⟨type_decl⟩ ::= ⟨type_definition⟩
| ⟨type_importation⟩

[V3+]

V3+ provide a way of declaring a type — called *deferred* —, while postponing its exact definition, and to write several, compatible, definitions — called *refinements* — of a deferred type. See §3.6.3.

⟨type_decl⟩ ::= ⟨type_definition⟩
| ⟨type_importation⟩
| ⟨deferred_type_definition⟩

[end V3+]

3.6.1 Defined Types

A type definition gives an identifier to a type expression.

type_definition ::= ⟨ident⟩ $\boxed{=}$ ⟨type⟩ $\boxed{;}$

Example
type complex = [**real**, **real**];

Provided such a definition appears in the program, the type identifier may be used anywhere in place of a type expression.

Example
var x, y : complex;

All type identifiers must be different. **bool**, **int**, and **real** are predefined type identifiers, which cannot be redefined.

3.6.2 Imported Types

Imported types are just names of types that are defined in another language. They are handled in the same way as the predefined types.

⟨type_importation⟩ ::= ⟨ident⟩ { [,] ⟨ident⟩ } [;]

[V3+]
In V3+, an imported type declaration is indicated by the keyword **imported**.

⟨type_importation⟩ ::= **imported** ⟨ident⟩ { [,] ⟨ident⟩ } [;]

[end V3+]

Example
type time;

Constant and variables of these types can be declared and used.

The only operators that can be applied to objects of imported types, are nodes, functions, and polymorphic operators.

if ... **then** ... **else** ..., **pre**, **->**, **when**, **current**

[V3+]
In V3+, [=] and [<>] are polymorphic, too.

[end V3+]

All other operations must be performed using an external function (see §5.1) provided by the user in another programming language.

Example

```
const initial_time : time;
...
function increment (t: time) returns (new_t: time);
...
node ...
var time : time;
let
    ...
    time = initial_time -> if second then increment(pre(time));
                           else pre(time);
    ...
tel
```

[V3+]

3.6.3 Deferred Types and Refinements

In V3+, the concept of refinement completes the concept of defined types. Defined types structure the data in a LUSTRE application in one particular way. By using type refinements, data can be structured in several different ways, i.e., a type can be decomposed in several different ways provided that it is possible to decompose all the refinements associated with the type into the same basic predefined or imported types. Refined types are extensions of LUSTRE types. Refined types may use defined types but the opposite is not true. Tuples are forbidden in refined types.

Example

```
type
    deferred T, T2, T3 ;
    T1 = int ;
refinement
    r1 : [T1, T2] refines T ;
    r2 : [T3, T1] refines T ;
    r3 : [bool,int] refines T2;
    r4 : [int,bool] refines T3;
```

In the example above, the deferred type T can be "flattened" into [**int**,**bool**,**int**] according to both of its refinements r1 and r2.

⟨deferred_type_definition⟩ ::= **deferred** ⟨ident⟩ { $\boxed{,}$ ⟨ident⟩ } $\boxed{;}$

⟨refinement_declarations⟩ ::= **refinement** ⟨refinement_decl⟩ { ⟨refinement_decl⟩ }

⟨refinement_decl⟩ ::= ⟨ident⟩ $\boxed{:}$ ⟨type⟩ **refines** ⟨ident⟩ $\boxed{;}$

In a type definition, the expression defining the type is exclusively made up of predefined types, imported types or types also defined by a type expression.

Example

```
type
    imported pressure ;
    deferred T1 ;
    T = [int, pressure] ; -- is correct
    T = [int, T1] ; -- is not allowed
```

A refinement can be associated only with a deferred type.

Example
refinement r : [**int**, **bool**] **refines** T

A refinement is an expression of simple types (no nesting).

Example

```
type
    deferred T1, T2 ;
refinement
    r1: [int, bool] refines T1 ; -- is correct
    r2 : [int, [bool, real]] refines T2 ; -- is not allowed
```

The types appearing in the refinement can be:

- predefined types,
- imported types,
- defined types, provided they are not tuples
- other types declared deferred.

Example

```
type
    deferred T ;
    T1 = [int, bool] ;
refinement
    r1 : [int, int, bool] refines T ;
    r2 : [int, T1] refines T ; -- is not allowed
```

It must be possible to decompose all the refinements associated with a given type into the same basic predefined or imported types.

Example

```
type
    deferred T, T1, T2 ;
    imported T3 ;
refinement
    r1 : [int, T2, T3] refines T ;
    r2 : [T1, bool, T3] refines T ;
    r3 : [int, real] refines T1 ;
    r4 : [real, bool] refines T2 ;
```

The type T can be decomposed into the types [**int**, **real** , **bool**, T3] whatever be the refinement.

[end V3+]

There is no implicit type casting in LUSTRE. The language provides two operators for explicit casting between integers and real: **int** converts a real into its integral part, and **real** converts an integer into a real.

[V3+]
In V3+, two casting functions are associated with each defined type and each refinement:

- the first function casts the type into the list of its component types.
- the second function performs the opposite operation: it recombines the component types to obtain the considered type.

These functions are respectively named

- _FROM_⟨defined_type_name⟩,and _TO_⟨defined_type_name⟩ for the defined types,
- and _FROM_⟨refinement_name⟩, and _TO_⟨refinement_name⟩ for the refinements .

Example

```
type
    T = [int, bool] ;
    deferred T1, T2, T3 ;
refinement r : [T1, T2] refines T3 ;
```

The associated conversion functions are:

_FROM_T :	T → **int**×**bool**
_TO_T :	**int**×**bool** → T
_FROM_r :	T3→ T1 × T2
_TO_r :	T1×T2 → T3

These functions could be used as follows:

```
x, t : T ;
y : int ;
z : bool ;
v1 : T1 ;
v2 : T2 ;
v3, v4 : T3 ;
x = _TO_T (y,z) ;
y,z = _FROM_T (t) ;
v3 = _TO_r (v1, v2) ;
v1, v2 = _FROM_T (v4) ;
```

[end V3+]

Chapter 4

Constants, and Variables

Contents of the chapter

4.1 Introduction

Declared flows can be

- variables, which are local to a node;
- global constants;
- formal parameters of nodes and functions.

This chapter deals with the declarations of flows of the first two kinds. Parameter declarations are described in Chapter 5.

As said in Chapter 1, a flow represents a sequence of typed values, appearing according to a clock. So, a type and a clock are associated with each flow. A constant flow is a sequence of constant values.

4.2 Constant Declarations

Constant can be declared by the user. Such a declaration can

- Associate an identifier with a constant flow
- Declare a constant imported from a host language (C).

Constant declarations appear at the upper level of the program (see Chapter 2). They are global to the whole program. Constants can be declared in any order.

$$\langle\text{constant\_declarations}\rangle ::= \textbf{const}\ \langle\text{const\_decl}\rangle\ \{\ \langle\text{const\_decl}\rangle\ \}$$
$$\begin{aligned}\langle\text{const\_decl}\rangle ::=\ & \langle\text{const\_definition}\rangle \\ |\ & \langle\text{const\_importation}\rangle\end{aligned}$$

4.2.1 Constant Definition

A constant definition gives an identifier to a constant expression.

$$\langle\text{const\_definition}\rangle ::= \langle\text{ident}\rangle\ \boxed{=}\ \langle\text{expression}\rangle\ \boxed{;}$$

[V3+]
In V3+, one can give both the type and the value of the defined constant.

$$\begin{aligned}\langle\text{const\_definition}\rangle ::=\ & \langle\text{ident}\rangle\ \boxed{=}\ \langle\text{expression}\rangle\ \boxed{;} \\ |\ & \langle\text{ident}\rangle : \langle\text{type}\rangle\ \boxed{=}\ \langle\text{expression}\rangle\ \boxed{;}\end{aligned}$$

[end V3+]

Example

```
const
    PI = 3.1416;
    RightAngle = PI/2.;
```

Provided such a definition appears in the program, the constant identifier may be used anywhere in place of a type expression.

The expression defining a declared constant must represent a constant flow. It may involve any data operator but no time operator.

The expression defining a declared constant must be well-typed, and its type is the type of the constant.

Such a constant is always considered to be on the basic clock of the node where it is used.

All constant identifiers must be different. **true** and **false** are predefined constant identifiers, which cannot be redefined.

[V4 and V5]
In V4 and V5, some integer parameters must be known at compile time (e.g., those defining array sizes). Only immediate values or defined constants may appear at these places.

Example

const size = 64;
type vector = **int**^size;

[end V4 and V5]

4.2.2 Imported Constants

Imported constants are just names of constants that are defined in another language. They are declared with their type. Several constants of the same type can be declared in the same declaration.

⟨const_importation⟩ ::= ⟨ident⟩ { $\boxed{,}$ ⟨ident⟩ } $\boxed{:}$ ⟨type⟩ $\boxed{;}$

Example

const
initial_time : TIME ;
Mo, Tu, We, Th, Fr, Sa, Su : DAY ;
Low_Bound, Up_Bound : real;

4.3 Variable Declarations

Inside a node, it is possible to declare local variables (see Chapter 5). These variables are only visible inside the node where they are declared. They are given with their type, and possibly their clocks.

⟨local_declarations⟩ ::= **var** ⟨var_decl⟩ { ⟨var_decl⟩ }
⟨var_decl⟩ ::= ⟨ident⟩ { $\boxed{,}$ ⟨ident⟩ } ⟨type⟩ [**when** ⟨clock⟩] $\boxed{;}$
⟨clock⟩ ::= ⟨ident⟩ | **true**

Several variables with the same type and clock may be declared together.

A variable identifier may not be the same as the one of a global constant or a formal parameter. **true** and **false** are predefined flows, which may not be redefined.

The clock must be either **true** (meaning the basic clock of the node), or the identifier of a visible boolean variable (a parameter or another local variable).

If the clock is not specified,

- in V3, V3+, V4, the clock of the variable will be synthesized from its definition.
- in V5, the variable is supposed to be on the basic clock of the node.

Example

```
var
    state : bool;
    coordinates : [ real , real ] when state ;
```

Chapter 5

Functions and Nodes

Contents of the chapter

5.1 Functions

Not all operators can be easily expressed in LUSTRE. It is sometimes necessary to write them in another programming language. Also, operators on external types must be described in another language.

The language allows functions to be imported from another language, and to be used as other operators in a LUSTRE program.

The function name and formal header must be specified in the LUSTRE program. These declarations appear at the upper level of the program (see Chapter 2). Declared functions are global to the whole program. Functions can be declared in any order.

$$\begin{array}{lcl} \langle\text{function\_decl}\rangle & ::= & \textbf{function}\ \langle\text{ident}\rangle\ \boxed{(}\ \langle\text{function\_parameters}\rangle\ \boxed{)} \\ & & \textbf{returns}\ \boxed{(}\ \langle\text{function\_parameters}\rangle\ \boxed{)}\ \boxed{;} \end{array}$$

A function cannot have the same identifier as another function or node. Recall that **int** and **real** are predefined functions (explicit casting), which cannot be redefined.

The only constraint in using a function is that it has no memory.

Example

function increment (time: time)[1]**returns** (newtime: time);
function decompose (time: time) **returns** (hour, minute, second: **int**);
...
(H,M,S) = **if** update
then decompose(updated_time)
else decompose(increment(actual_time));

5.1.1 Function Parameters

Input and output function parameters follow the same syntax:

⟨function_parameters⟩ ::= ⟨func_param⟩ { ⟨;⟩ ⟨func_param⟩ }
⟨func_param⟩ ::= ⟨ident⟩ { $\boxed{,}$ ⟨ident⟩ } $\boxed{:}$ ⟨type⟩

All the parameters of a function must have different identifiers. These identifiers must also be different from the ones of global constants and from predefined identifiers **true** and **false**.

[V4 and V5]
In V4 and V5, a function has at least one input and one output parameter.

[end V4 and V5]

Parameters of a function cannot be clocked.

5.2 Nodes

A node is a network of operators described in LUSTRE for which formal input and output parameters have been defined.

A node is instantiated by replacing the formal parameters with actual parameters.

Example

node N (x : **int**) **returns** (y : **int** ; z : **real**) ;
let
...
tel
node M (e : **int**) **returns** (s : **int** ; v : **real**) ;
let
s, v = N (e) ;
tel

[0]Recall that such a synonymy between a variable and a type is forbidden in V3+.

5.2.1 Node Declaration

Node declarations appear at the upper level of the program (see Chapter 2). Declared nodes are global to the whole program. Nodes can be declared in any order.

The syntax of a node declaration is the following:

⟨node_decl⟩ ::= ⟨header⟩ [⟨local_declarations⟩] ⟨node_body⟩

⟨header⟩ ::= **node** ⟨ident⟩ $\boxed{(}$ ⟨node_in_parameters⟩ $\boxed{)}$
returns $\boxed{(}$ ⟨node_out_parameters⟩ $\boxed{)}$ $\boxed{;}$

The syntax of parameter declarations is given in the next sections.

Local variables can be declared.

⟨local_declarations⟩ ::= (see §4.3)

The body of a node is a system of equations and assertions:

⟨node_body⟩ ::= (see Chapter 6)

Two nodes cannot have the same identifier. A node and a function cannot have the same identifier. Recall that **int** and **real** are predefined functions (explicit casting), which cannot be redefined.

[V4 and V5]
In V4 and V5, a node has at least one input and one output parameter.

[end V4 and V5]

5.2.2 Node Input Parameters

Input parameters of a node may be clocked, and may be declared to be constant:

⟨node_in_parameters⟩ ::= ⟨in_par_decl⟩ { $\boxed{;}$ ⟨in_par_decl⟩ }
⟨in_par_decl⟩ ::= ⟨in_param⟩ [**when** ⟨clock⟩]
| $\boxed{(}$ ⟨in_param⟩ { $\boxed{;}$ ⟨in_param⟩ } $\boxed{)}$ **when** ⟨clock⟩
⟨in_param⟩ ::= ⟨ident_decl⟩
| **const** ident_decl
⟨ident_decl⟩ ::= ⟨ident⟩ { $\boxed{,}$ ⟨ident⟩ } $\boxed{:}$ ⟨type⟩
⟨clock⟩ ::= ⟨ident⟩ | **true**

All the parameters of a node must have different identifiers. These identifiers must also be different from the ones of global constants and from the predefined identifiers **true** and **false**.

Several parameters with the same type and clock may be declared together (rule ⟨ident_decl⟩), provided they are either all constant or all variable (rule ⟨in_param⟩). A clock association may spread over many parameter declarations (rule ⟨in_par_decl⟩).

The clock must be either **true** (meaning the basic clock of the node), or the identifier of another Boolean input parameter. If the clock is not specified, the parameter is on the base clock of the node.

If an input parameter is specified to be constant, the compiler will check that the corresponding actual parameter will be a constant[1]. These parameters must not be confused with the constants that are global to the whole description.

Example

node N (x, y : bool; **const** low, high : int) **returns**...
node M (x, y : bool; a: int **when** x; (b:int; r: real) **when** y) **returns**...

5.2.3 Node Output Parameters

The syntax of output parameters is the same, except that they cannot be declared to be constant:

⟨node_out_parameters⟩ ::= ⟨out_par_decl⟩ { $\boxed{;}$ ⟨out_par_decl⟩ }
⟨out_param⟩ ::= ⟨ident_decl⟩ [**when** ⟨clock⟩]
| $\boxed{(}$ ⟨ident_decl⟩ { $\boxed{;}$ ⟨ident_decl⟩ } $\boxed{)}$ **when** ⟨clock⟩
⟨ident_decl⟩ ::= ⟨ident⟩ { $\boxed{,}$ ⟨ident⟩ } $\boxed{:}$ ⟨type⟩
⟨clock⟩ ::= ⟨ident⟩ | **true**

All the parameters of a node must have different identifiers. These identifiers must also be different from the ones of global constants and from the predefined identifiers **true** and **false**.

The clock must be either **true** (meaning the basic clock of the node), or the identifier of another Boolean input or output parameter. If the clock is not specified,

- in V3, V3+, V4, the clock of the parameter will be synthesized from its definition;
- in V5, the parameter is on the basic clock of the node.

5.3 Calling a Function or a Node

A function or node call is an expression (see Chapter 7). Its type is deduced from the types of output formal parameters (see §3.4).

A function or node is called by means of its identifier, with a list of actual input parameters, which are expressions.

[1]This is not true in V3+.

⟨call⟩ ::= ⟨ident⟩ $\boxed{(}$ ⟨expression⟩ { $\boxed{,}$ ⟨expression⟩ } $\boxed{)}$

A node call defines an instance of the node. Each instance has its own memory, and its own clock.

The types of actual input parameters must match those of formal input parameters.

The clocks of actual input parameters of a function should be the same. This common clock is also the clock of the actual outputs.

For a node call:

- The clocks of actual input parameters corresponding to formal parameters on the base clock should be the same. This common clock defines the basic clock of the instance of the node corresponding to the call.

  [V3+]
  In V3+, a node is allowed to have no input. In that case, the basic clock of an instance of the node is the one of the calling node.

  [end V3+]

- The clocks of other actual input parameters should be consistent with those of corresponding formal parameters, after substitution of actuals to formals.

- The clocks of actual outputs is determined from those of formal output parameters, after substitution of actuals to formals.

Example
Consider the node declaration

```
node N (c: bool; a: int when c)
    returns ( d: bool when c ; b: int when d);
```

The call "N(E1,E2)" is correct provided

- E1 is boolean,
- E2 is integer, on clock E1

It returns two flows F1 and F2, such that

- F1 is boolean, on clock E1
- F2 is integer, on clock F1

The basic clock of this call will be the clock of E1.

Chapter 6

Equations and Assertions

Contents of the chapter

6.1 Node Body

The body of a node is a system of equations and assertions. Equations define the output parameters and possibly the local variables of the node. Each output parameter and each local variable must be defined once and only once in the body. Assertions specify Boolean expressions which must be always true; they are used to express properties of the input parameters (preconditions).

$\langle$node_body$\rangle$ ::= **let** $\langle$eq_or_assert$\rangle$ { $\langle$eq_or_assert$\rangle$ } **tel**

$\langle$eq_or_assert$\rangle$ ::= $\langle$equation$\rangle$
| $\langle$assertion$\rangle$

6.2 Equations

Equations define output parameters and local variables. In the simplest case, an equation defines one variable. However, in some cases, an equation may define several variables. So, we distinguish between *simple* and *multiple* equations. In V4 and V5, because of the presence of structured objects (tuples and arrays), more complex *structured equations* are used.

6.2.1 Simple equations

A variable that is not an input for the node is defined by one and only one equation. The expression defining a variable must be of the same type as the variable.

Example

```
X = Y + Z ;
```

defines the variable X to be the sequence made of the pointwise sum of the sequences of Y and Z. Here X, Y, and Z must be of the same type (**int** or **real**).

6.2.2 Multiple equations

LUSTRE allows several variables to be defined in one equation.

Example

```
(min,max) = if a<b then (a,b) else (b,a) ;
```

is an abbreviated way of writing:

```
min = if a<b then a else b ;
max = if a<b then b else a ;
```

The right-hand part of such a multiple equation can be:

- an expression built from polymorphic operators, which are the following:
  - the conditional (**if...then...else...**)
  - the time operators: **pre**, **->**, **when**, **current**
- a comma separated list of expressions
- a node or function call

In this case, the types of both sides of the equation are "list" types (see §6.2.4), which should match.

Example

```
(X, Y) = if C then (E, F) else (E1, F1) ;
(X, Y) = (E, F) -> pre (X, Y) ;
(X, Y) = (E, F) when C ;
(X, Y) = current (E, F) ;
(X, Y) = N(U, V, W) ;
```

are correct equations provided E, E1 and the first output parameter of N have the same type as X, and F, F1 and the second output parameter of N have the same type as Y.

6.2.3 Syntax of Equations

⟨equation⟩ ::= ⟨left_part⟩ $\boxed{=}$ ⟨expression⟩

⟨left_part⟩ ::= ⟨ident⟩
| $\boxed{(}$ ⟨ident⟩ { $\boxed{,}$ ⟨ident⟩ } $\boxed{)}$

[V3+]
In V3+, parentheses are ommitted around a left part that is a list.

[end V3+]

[V4 and V5]
In V4 and V5, one can have access to components of structured objects (records and arrays) in place of ⟨ident⟩ in left parts (see §6.2.5).

[end V4 and V5]

6.2.4 Usage Rules

- A variable and the expression defining it must be of the same type and of the same clock.

- A right part expression can be a list of expressions. Its type is a list of types, which must match the type of the corresponding component of the left part. While an expression can be of type "list", it is impossible to declare a variable of such type (see §3.4).

- notice that if a list expression contains a subexpression that is also a list, the resulting type is a simple list (in other words, lists are always flatenned)

  Example
  Given the node header

  ```
  node N (a,b: int) returns (c,d: int);
  ```

and assuming that all the variables A, B, C, D, E, F, G, H, are declared of type **int** the following equation is correct

```
(A, B, C, D) = (N(E,F) , N(G,H)) ;
```

[V3+]
Such a nesting of lists is forbidden in V3+, which imposes the following restrictions about lists:

- A call to a node with several outputs must appear alone in the right-hand side of an equation.
- A list can only appear as the argument of a polymorphic operator.

Example

```
(A,B) = (E,F) ; -- is forbidden
(A,B) = pre(E,F) ; -- is allowed
```

[end V3+]

[V4 and V5]

6.2.5 Structured Equations in V4 and V5

V4 and V5 provide an extension of the notion of multiple equation. When the right part of the equation is of structured type (i.e., a list, a tuple, or an array), the left part may be a *filter*, i.e., a structured construction whose components are defined by the corresponding components of the right part.

Example
Assume e is an expression of type [bool, [int, int]], c is a boolean variable, a, b are integer variables, and A is a variable of type [int, int]. Then

```
[c,[a,b]] = e;
 [c,A] = e;
```

are two legal equations, which are equivalent, respectively, to

```
c = e[0]; a = e[1][0]; b = e[1][1];
```

and

```
c = e[0]; A = e[1];
```

Another form equivalent to the second equation is

```
[c,[A[0],A[1]]] = e;
```

So, in V4 and V5, the syntax of the left part of an equation is

⟨left_part⟩	::=	⟨struct_item⟩
	|	$\boxed{(}$ ⟨struct_item⟩ { $\boxed{,}$ ⟨struct_item⟩ } $\boxed{)}$

i.e., either a structured item or a list of structured items. A structured item is either an item, or an array or a tuple of structured items.

⟨struct_item⟩	::=	⟨item⟩
	|	⟨tuple_struct_item⟩
	|	⟨array_struct_item⟩

Recall that, in V5, the syntax of tuples differ from that of arrays:

[V4]

⟨tuple_struct_item⟩	::=	$\boxed{[}$ ⟨struct_item⟩ { $\boxed{,}$ ⟨struct_item⟩ } $\boxed{]}$
⟨array_struct_item⟩	::=	⟨tuple_struct_item⟩

[end V4]

[V5]

⟨tuple_struct_item⟩	::=	$\boxed{\{}$ ⟨struct_item⟩ { $\boxed{,}$ ⟨struct_item⟩ } $\boxed{\}}$
⟨array_struct_item⟩	::=	$\boxed{[}$ ⟨struct_item⟩ { $\boxed{,}$ ⟨struct_item⟩ } $\boxed{]}$

[end V5]

Finally, an ⟨item⟩ is either a variable identifier, or a selection in a array or tuple (see §7.4)

⟨item⟩	::=	⟨ident⟩
	|	⟨selection⟩

[end V4 and V5]

6.3 Assertions

A Lustre program is intended to tightly react to an environment. It is very common that some properties of this environment are known, which allows a better code to be generated, or which are necessary to prove most critical properties of the program. The language allows such *assertion* to be specified in the program, by stating that arbitrary boolean expressions are always true.

⟨assertion⟩ ::= **assert** ⟨expression⟩

The expression must be of boolean type. It may involve any (declared, visible) variable or operator.

Chapter 7

Expressions

Contents of the chapter

7.1 Syntax

Expressions in LUSTRE are built from variables, constants, basic and user-defined operators. The complete syntax of expressions is the following:

$$
\begin{array}{lll}
\langle\text{expression}\rangle & ::= & \langle\text{ident}\rangle \\
& | & \langle\text{constant}\rangle \\
& | & \boxed{(}\ \langle\text{expression}\rangle\ \{\ \boxed{,}\ \langle\text{expression}\rangle\ \}\ \boxed{)} \\
& | & \boxed{[}\ \langle\text{expression}\rangle\ \{\ \boxed{,}\ \langle\text{expression}\rangle\ \}\ \boxed{]} \\
& | & \langle\text{expression}\rangle\ \boxed{[}\ \langle\text{int\_value}\rangle\ \boxed{]} \\
& | & \langle\text{unary}\rangle\ \langle\text{expression}\rangle \\
& | & \langle\text{expression}\rangle\ \langle\text{binary}\rangle\ \langle\text{expression}\rangle \\
& | & \textbf{if}\ \langle\text{expression}\rangle\ \textbf{then}\ \langle\text{expression}\rangle\ \textbf{else}\ \langle\text{expression}\rangle \\
& | & \langle\text{call}\rangle \\
& | & \boxed{\#}\ \boxed{(}\ \langle\text{expression}\rangle\ \{\ \boxed{,}\ \langle\text{expression}\rangle\ \}\ \boxed{)}
\end{array}
$$

constant ::= ⟨bool_value⟩
| ⟨int_value⟩
| ⟨real_value⟩

⟨unary⟩ ::= $\boxed{-}$ | **not** | **int** | **real** | **pre** | **current**

binary ::= $\boxed{+}$ | $\boxed{-}$ | $\boxed{*}$ | $\boxed{/}$ | **div** | **mod** | $\boxed{<}$ | $\boxed{<=}$ | $\boxed{=}$
| $\boxed{<>}$ | $\boxed{>=}$ | $\boxed{>}$ | **or** | **and** | **->** | **when**

⟨call⟩ ::= ⟨ident⟩ $\boxed{(}$ ⟨expression⟩ { $\boxed{,}$ ⟨expression⟩ } $\boxed{)}$

⟨bool_value⟩ ::= **true** | **false**

[V3+]
V3+ offers also a "**case**" construction, and a way to associate an "activation condition" to a node call:

⟨expression⟩ ::= ⟨case⟩
| ⟨condact⟩

⟨case⟩ ::= **case** ⟨expression⟩ **of** ⟨caselist⟩

⟨caselist⟩ ::= { ⟨int_value⟩ $\boxed{:}$ $\boxed{(}$ ⟨expression⟩ $\boxed{)}$ $\boxed{,}$ } **default** $\boxed{:}$ $\boxed{(}$ ⟨expression⟩ $\boxed{)}$

⟨condact⟩ ::= **condact** $\boxed{(}$ ⟨expression⟩ $\boxed{,}$ ⟨call⟩ $\boxed{,}$ ⟨expression⟩ { $\boxed{,}$ ⟨expression⟩ } $\boxed{)}$

The first argument of a "**case**" expression is an integer expression, the value of which selects an expression in the ⟨caselist⟩. The "activation condition" is described in §7.3.2.

[end V3+]

7.2 Constants

A constant is considered as a constant flow (i.e., a sequence of constant values), at the basic clock of the node it appears in. For example, the constant 1 expresses the suite of values (1, 1..., 1...). Constants are either built-in (like **true**, **false**, integer and real numbers, see §1.4.1) or external. External constants are declared with their types. In the same way, it is possible to give a name to a constant (see §4.2).

7.3 Operators

Lustre offers standard arithmetic, logical, relational, and conditional operators — called *data operators* —, together with four specific *time operators*, and data-structuring operators.

7.3.1 Data Operators

Table 7.3.1 gives the name, arity, and signature of each data operator: an operator with signature $\tau_1 \times \tau_2 \times \ldots \times \tau_n \to \tau$ takes n arguments, of respective types $\tau_1, \tau_2, \ldots, \tau_n$, and returns a result of type τ. A "$\star$" in place of a type means "any type" (polymorphic operator); for a given operator, all the types noted "$\star$" should be the same. "τ^+" means "any positive number of arguments of type τ. Notice that an operator can have several signatures (overloaded operators).

[V3+]
In V3+, the operator $\boxed{/}$ has also the following signatures:

$$\textbf{int} \times \textbf{int} \to \textbf{real} \quad , \quad \textbf{int} \times \textbf{real} \to \textbf{real} \quad , \quad \textbf{real} \times \textbf{int} \to \textbf{real}$$

The signature of the "**case**" operator is

$$\textbf{int} \times (\textbf{int} \times \star)^+ \to \star$$

[end V3+]

- All data operators apply pointwise on flows.
- All data operators should be applied to operands on the same clock, which is also the clock of the result.
- The `#` operator applies to a list of boolean expressions and evaluates to **true** if and only if at most one of its operand is true. This operator should not be confused with the exclusive or operator (which is `<>` applied to boolean arguments).
- The **div** operator means integer division.
- The **mod** operator means the modulo operation.

7.3.2 Time Operators

Table 7.3.2 gives the arity and signature of time operators.

- The **pre** operator returns a result of the same type and the same clock as its argument. The value returned at the nth cycle of this clock is the value of the argument at the $(n-1)$th cycle. The returned value at the first cycle is *nil*, the undefined value.
- The `->` operator should be applied on arguments of the same types and clocks, which are also the type and the clock of the result. It is used to specify initial values: At the first cycle, it returns the current value of its first argument; at any other cycle, it returns the current value of its second argument.
- The **when** operator should be applied to arguments on the same clock, the second argument being boolean. It returns a result of the same type as its first argument, and on the

Arithmetic operators

+ , - , *	binary	**int**×**int**→**int** **real**×**real**→**real**
-	unary	**int**→**int** **real**→**real**
div , **mod**	binary	**int**×**int**→**int**
/	binary	**real**×**real**→**real**

Boolean operators

not	unary	**bool**→**bool**
or , **and**	binary	**bool**×**bool**→**bool**
#	n-ary	**bool**$^+$×**bool**

Relational operators

= , <>	binary	**bool**×**bool**→**bool** **int**×**int**→**bool** **real**×**real**→**bool**
< , <= , >= , >	binary	**int**×**int**→**bool** **real**×**real**→**bool**

Conditional operator

if ... **then** ... **else**...	ternary	**bool**$\times \star \times \star \to \star$

Conversion operators

int	unary	**real**→**int**
real	unary	**int**→**real**

Table 7.1: Data operators

pre , **current**	unary	$\star \to \star$
->	binary	$\star \times \star \to \star$
when	binary	$\star\times$**bool**$\to \star$

Table 7.2: Time operators

clock defined by its second argument (i.e., the sequence of cycles where it is true). It is used to sample a flow on a slower clock: The result is the sequence of values of the first argument when the second argument is true. Notice that, when the second argument is false, the result *has no value*.

- The **current** operator is used to extend a flow on a faster clock. Applied to an argument which is on a clock defined by a boolean flow, say H, it returns a flow of the same type as the argument, and on the same clock as H, whose value is the value taken by the argument the last time H was true (the result is *nil* as long as H has never been true).

The following table illustrates the behavior of these time operators: each column corresponds to a cycle of the basic clock, and each line gives the sequence of values of a variable or expression. An empty space in the table indicates that the variable has no value for the cycle. Notice that the undefined value *nil* differs from the absence of value.

basic clock	0	1	2	3	4	5	6	7	8	9
X	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
Y	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9
pre(X)	*nil*	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
X->Y	x_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9
Y->**pre**(X)	y_0	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
H	false	true	false	true	true	false	false	true	false	true
clock on H		0		1	2			3		4
X **when** H		x_1		x_3	x_4			x_7		x_9
pre(X **when** H)		*nil*		x_1	x_3			x_4		x_7
current(X **when** H)	*nil*	x_1	x_1	x_3	x_4	x_4	x_4	x_7	x_7	x_9

[V3+]

Activation condition

An activation condition can be associated with a node call, in V3+, as a shorthand for a more complicated construction based on clocks. Basically, associating an activation condition C (i.e., a boolean expression) to a node call N(x,y,...) means that

- the node is only activated when C is true
- when C is false, the result is the last result returned by the node.

In other word, the intuitive meaning is

current(N(x **when** C, y **when** C, ...))

Now, the actual construct is more complicated, as it allows default initial values to be specified, which are taken as long as C has never been true. Let N be a node with n input parameters, and m output parameters, then a call to N activated with C will be written

condact(C, N(x_1,x_2,...x_n), v_1, v_2,..., v_m)

where v_i is the default value of the ith output parameter. The result of this expression is equivalent to the list (y_1, y_2,..., y_m) defined as follows in pure LUSTRE:

(y_1, y_2,..., y_m) = **if** C **then** current(N((x_1,x_2,...x_n) **when** C))
else (v_1, v_2,..., v_m) -> **pre**(y_1, y_2,..., y_m)

[end V3+]

7.3.3 Data Structuring Operators

Tuple Operator

The tuple operator builds a tuple (for more details see Chapter 3). If e_0, e_1, ..., e_k are expressions of respective types $\tau_0, \tau_1, \ldots \tau_k$, on the same clock, and such that, for each $i = 0 \ldots k$, e_i defines the sequence $(e_0^{(i)}, e_1^{(i)}, \ldots, e_n^{(i)}, \ldots)$, then [$e_0$, e_1, ..., e_k] is an expression of tuple type $[\tau_0, \tau_1, \ldots \tau_k]$, whose value at the nth instant is $[e_n^{(0)}, e_n^{(1)}, \ldots, e_n^{(k)}]$

Selection Operator

The components of an expression of tuple type can be accessed via their position in the tuple using the selection operator. If e is an expression of some tuple type $[\tau_0, \tau_1, \ldots \tau_k]$, if i is an integer, with $0 \leq \mathsf{i} \leq k$, then e[i] is an expression of type τ_i referring to the ith component of e. Notice that tuple fields are always numbered from 0.

[V5]
In V5, in order to prepare the introduction of records with named fields, one writes e.%i instead of e[i].

[end V5]

[V4 and V5]
V4 and V5 provide more sophisticated selection operators for arrays (see §7.4).

[end V4 and V5]

7.3.4 Operator Priorities

While any subexpression can be parenthesized to make explicit the evaluation order, there are implicit priorities over operators. There are 10 levels of priority (level 1 being the most prioritary):

Level 1: [ ] (selection)

Level 2: **int**, **real**, - (unary), **pre**, **current**, **not**, #

Level 3: when

Level 4: `*`, **div**, **mod**, `/`

Level 5: `+`, `-` (binary)

Level 6: `<`, `<=`, `=`, `<>`, `>=`, `>`

Level 7: and

Level 8: or

Level 9: `->`

Level 10: if then else

$\boxed{\text{La doc donne des priorités différentes. A contrôler sur V3+.}}$

Inside each priority level, priority is given left to right.

Example

a + 2 * b	means	a + (2 * b)
a + b - c	means	(a+b) -c
not a = b	means	(**not** a) = b
if c **then** a **else** e ->f	means	**if** c **then** a **else** (e ->f)

7.4 Arrays in V4 and V5

[V4 and V5]

In V4 and V5, several specific operators are provided for handling arrays.

7.4.1 Array Constructors

As said before, in V4, arrays are special cases of tuples. So the tupling operator (see §7.3.3) is also an array constructor: if $e_0, e_1, \ldots, e_{n-1}$ are expressions of the same type T,

$$[\ e_0,\ e_1,\ \ldots,\ e_{n-1}\ ]$$

is an expression of type `T^n`, denoting the array whose ith component is e_i (Recall that all arrays, like tuples, are indexed from 0). In V5, tuples and arrays are different:

- $[\ e_0, e_1, \ldots, e_{n-1}\ ]$ is an array, as above;
- $\{\ e_0, e_1, \ldots, e_{n-1}\ \}$ is a tuple, whatever be the types of $e_0, e_1, \ldots, e_{n-1}$.

Another array constructor is provided in both versions: if `e` is an expression of type `T`, then `e^n` is an expression of type `T^n`, denoting the array all the components of which are equal to `e`.

7.4.2 Array Selectors

Component Selection

A component of an array may be selected by its index, provided this index is known at compile time. If A is of type T^n, if i is a constant integer expression whose value is known at compile time and belongs to the interval $[0..n-1]$, then A[i] is an expression of type T, equal to the ith component of A (Recall the array components are indexed from 0).

Slice Selection

A whole, contiguous, part of an array can be selected. It is called a *slice*. Let A be of type T^n, let i and j be two constant integer expressions, whose values are known at compile time and belong to the interval $[0..n-1]$, then A[i..j] is the array

- [A[i], A[i+1], ..., A[j]], if i≤j
- [A[i], A[i-1], ..., A[j]], if i>j

Notice that a slice is an array, indexed from 0.

Example
Let A = [1, 2, 3, 4, 5, 6].
A[1] is 2;
A[2..4] is [3, 4, 5];
A[4..2] is [5, 4, 3];
A[2..2] is[1] [3];
A[2..4][0] is 3.

[V4]
V4 provides a **step** operator to select non-contiguous slices. A selector of the form [i..j **step** k] selects all the components with indexes i+ $\pm\ \ell$k belonging to the interval [i,j].

Example
If A = [1, 2, 3, 4, 5, 6], A[0..3 **step** 2] is [1,3].

This feature has been given up in V5, because of compiling problems.

[end V4]

Multidimensional Slices

Slicing shows counter-intuitive behavior when applied to "arrays of arrays".

Example
For instance, let A be of type T^10 ^5, (i.e., an array of 5 components, each of which is an array of 10 components of type T). Now, the slice A[1..4][1..2] seems to select a rectangular sub-matrix

$$\mathsf{B} = \begin{bmatrix} \mathsf{A[1][1]} & \mathsf{A[1][2]} \\ \mathsf{A[2][1]} & \mathsf{A[2][2]} \\ \mathsf{A[3][1]} & \mathsf{A[3][2]} \\ \mathsf{A[4][1]} & \mathsf{A[4][2]} \end{bmatrix}$$

of A. However, coming back to the definition, we get that A[1..4] is the array [A[1], A[2], A[3], A[4]], from which the slice selects components 1, and 2, so the slice is in fact [A[1], A[2]].

This is why the language offers a specific slicing mechanism to select slices through several dimensions: if A is of type T^n ^m, if i1, i2 (resp., j1, j2) are constant integer expressions, known at compile-time, and belonging to $[0, n-1]$ (resp., to $[0, m-1]$), then A[i1..i2, j1..j2] is the rectangular sub-matrix of A, made of components A[i][j] such that i∈[i1,i2] and j∈[j1,j2].

Example
Coming back to the previous example, the correct selection of B is A[1..4,1..2].

7.4.3 Operating on Arrays

Polymorphic operators

Of course, polymorphic operators may be applied to arrays.

Example
Let A and B be arrays of the same type (i.e., same size and same component type), let C be a boolean expression, then A -> B, **if** C **then** A **else** B, **current** (A **when** C) are correct expressions.

Homomorphic extension of operators

Any operator, node, or function, is extended to be applied to arrays, in the following way (called *homomorphic extension*): If **op** is an operator, a node, or a function, of signature

$$(\tau_1, \ldots, \tau_n) \rightarrow (\tau'_1, \ldots, \tau'_m)$$

it can be applied to arguments of types $(\tau_1\text{\^{}}k, \ldots, \tau_n\text{\^{}}k)$, then returning a result of type $(\tau'_1\text{\^{}}k, \ldots, \tau'_m\text{\^{}}k)$, computed by pointwise application to the components of its arguments.

Example
Let A and B be two boolean arrays of the same size. Then A **or** B is a boolean array of the same size, whose ith component is equal to A[i] **or** B[i].

As a consequence, if A and B are arrays of the same type, the expression **if** C **then** A **else** B is correct, be C be a boolean, or a boolean array of the same size as A and B.

- In the first case, the polymorphism of **if-then-else** is used, and the ith component of the result is equal to **if** C **then** A[i] **else** B[i];

- In the second case, the homomorphic extension of **if-then-else** is considered, and the ith component of the result is equal to **if** C[i] **then** A[i] **else** B[i].

Remark: The $\boxed{=}$ and $\boxed{\texttt{<>}}$ operators are **not** polymorphic: if A and B are arrays of the same type, the only meaning that can be given to the expression A=B is the homomorphic extension. The result is an array, the ith component of which is true if and only if A[i]=B[i]. We give below an example of a node which returns **true** if and only if A and B are equal.

Concatenation operator

Finally, an operator is provided to concatenate two arrays. If A and B are arrays, whose components are of the same type, then A | B is the array obtained by concatenating A and B

Example
[1, 2, 3] | [4, 5] is [1, 2, 3, 4, 5]

7.4.4 Array Equations

Let us give some examples of programs using arrays, to show the style of equations that can be written.

Array equality

As announced before, our first example is a node taking two integer arrays A and B of the same size, and returning a boolean which is true if and only if they are componentwise equal. The proposed solution uses two auxiliary arrays: the first one, E, has its ith component true if and only if A[i] and B[i] are equal; the second one, C has its ith component true if and only if E[j] is true for all j, $0 \leq j \leq i$. As a consequence, A and B are equal if and only if the last component of C is true.

```
node EQ (const n: int; A, B: int^n) returns (equal: bool;
var C, E : bool^n;
let
    equal = C[n-1];
    C = [E[0]] | C[0..n-2] and E[1..n-1];
    E = (A=B);
tel
```

Comments:

- the first equation defines the output equal to be the last component of C.
- the second equation defines the array C as the concatenation of
  - a one-component array [E[0]], whose only component is E[0] (meaning C[0] = E[0])
  - an array of size n-1, whose ith component is C[i] **and** E[i+1], meaning that, for $1 \leq i \leq n-1$, C[i] = C[i-1] **and** E[i].
- the third equation defines E[i] to be A[i]=B[i].

7.4.5 An arbitrary delay operator

Our next example is a generalization of the **pre** operator. It takes an integer flow, say X, and returns its nth previous value. During the n first cycles, it returns a default value.

```
node DELAY (const n, default : int; X: int) returns (delayed: int);
var window : int ^n;
let
    delayed = window[n-1];
    window = [X] |( default^(n-1) -> pre(window[0..n-2]) );
tel
```

Here, the definition of the auxiliary array window means

$$\text{window[i]} \;=\; \begin{cases} \text{X} & \text{for } i = 0 \\ \text{default -> } \mathbf{pre}\text{(window[i-1])} & \text{for } 1 \leq i < n \end{cases}$$

[end V4 and V5]

Bibliography

[BCH+85] 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.

[Ber86] J-L. Bergerand. LUSTRE: un langage déclaratif pour le temps réel. Thesis, Institut National Polytechnique de Grenoble, 1986.

[BP88] J. L. Bergerand and E. Pilaud. SAGA, a software environment for dependable automatic control. In *IFAC SAFECOMP'88*, Fulda, 1988.

[CPHP87] 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.

[HL91] N. Halbwachs and F. Lagnier. Sémantique statique du langage lustre - version 3. Technical Report SPECTRE L15, IMAG, Grenoble, February 1991.

[Pla88] J. A. Plaice. Sémantique et compilation de LUSTRE, un langage déclaratif synchrone. Thesis, Institut National Polytechnique de Grenoble, 1988.

[Ray91] P. Raymond. Compilation efficace d'un langage déclaratif synchrone : Le générateur de code LUSTRE-v3. Thesis, Institut National Polytechnique de Grenoble, November 1991.

[RH91] 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.

[Roc92] 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.

[Ver96] SAO+/SAOCG - Langage Lustre, manuel de référence. Technical report, Vérilog S.A., 1996.

Index