Automatic control systems programming using a real-time declarative language

J.-L. Bergerand P. Caspi N. Halbwachs J. A. Plaice

Abstract

LUSTRE is a declarative programming language based on the same principle as LUCID, i.e., programs operate over infinite sequences of values. The difference between LUSTRE and LUCID comes from the fact that LUSTRE is intended to be interpreted synchronously, i.e., each variable is intended to possess the n-th value of its associated sequence at the n-th execution step of the program. This interpretation provides LUSTRE with real-time capabilities, since the execution steps of a program may be viewed as a physical time scale. This paper briefly presents the language and illustrates its use on several examples taken from the field of logical and numerical process control.

Keywords: Programming languages, Parallel processing, Numerical control, Discrete-time systems, Transfer functions, \( z \)-transforms.

1 Introduction

This paper presents the programming language LUSTRE (a French acronym for “synchronous real-time LUCID”). Here, “real-time” refers to the intended application field of the language: systems which present “hard” real-time constraints such as compulsory response times or frequencies. This distinguishes our domain from those where real-time is interpreted in a looser sense, such as in terms of average performances.

Our domain is primarily concerned with the programming of logical and/or numerical automatic control systems. In these systems, computers and programs interact with physical environments (plants). These environments have their own dynamics and cannot wait for synchronization with programs; thus, timing constraints appear as a synchronization tool between the programs and their environments. Beside this main feature, the domain presents two other characteristics, which have played a great rôle in the design choices of our language:

- First, these systems are intrinsically parallel. Automatic control systems are usually designed and specified in terms of Boolean equations, gates or switch networks, and block diagrams of transfer functions. In addition, in former discrete and analog implementations, the corresponding behaviours were implicitly parallel. The generalized use of digital computers has obliged designers to spend a lot of effort in sequentializing these parallel behaviours, so as to meet the concepts of traditional programming languages. This is surprising, since, at the same time, computer scientists were evolving towards a better accounting for parallelism. However, this evolution towards parallelism is a very slow one, because in classical programming fields, sequential concepts are very natural and people are used to them. This is not the case in automatic control programming in which, as noted above, parallelism is a natural concept and meets the traditional way of thinking of its users. This has led us to design LUSTRE as a highly parallel programming language.

- Second, these systems often require a high degree of reliability, for in such systems as aircrafts, nuclear plants, etc., programming errors can lead to catastrophic consequences. Of
course, wrong programs can be written in any language, but more than 30 years of programming experience have shown that some features of programming languages are likely to enhance the reliability of programs:

- High-level languages shorten the bridge between a system specification and its implementation. From this point of view, LUSTRE can be seen as a high-level language for automatic control. In particular, its implicit parallelism will allow easy programming of parallel automatic control specifications.
- Syntactic redundancies, such as strong typing and declarations, allow consistency checks at compile time, and may help in detecting errors before they have serious consequences. A well-known bug in a spacecraft control system was due to the lack of such a feature in FORTRAN. Thus, unlike functional languages such as LISP, ML or FP, which are prone to type synthesis and where declarations are optimal, LUSTRE has been given a rigid PASCAL-like syntax.
- Finally, it has been recognized that the existence of simple formal semantics for a given language is a factor that enhances its reliability; formal semantics are a prerequisite of program verification; of course, one can argue that this still remains a difficult exercise, even for simple sequential programs, and that proofs are seldom undertaken. However, it appears that programs that can be verified are also easier to understand, test and debug. A well-known example is the one of structured programs, that have simpler semantics than unstructured ones. That is why programming languages for automatic control should be given formal semantics. Furthermore, these semantics cannot be conventional ones: usual semantics intend to answer the question “What does the program do?” while in real-time programming, another equally important question is “When does it do it?” Concerning LUSTRE programs, simple answers will be given to both questions.

Let us now consider the concept of synchrony, which is essential when comparing LUSTRE to other real-time languages: The basic features of most real-time languages come from operating systems concepts: in this field, people wanted to express activities that could run in time-sharing, and could communicate and synchronize with each other. Several concepts were designed, such as sequential processes, communication through shared memory, synchronization primitives (fork-join, semaphores, monitors, ...). To render simpler the semantics of such languages, synchronization and communication were unified into higher-order concepts: rendez-vous (in CSP or ADA), and communication queues (in “data-flow” languages [8, 9, 12]). Now, the activity of most real-time language designers has consisted of adding timing primitives to such languages. For instance, many languages possess some mechanisms that force an active process to wait until a given condition becomes true. Using such a mechanism and a real-time clock, people were able to design a waiting primitive such as “delay 5s”, meaning that a process reaching this statement at time t will be forced to a waiting state until t + 5s. The problem which arises is that the programmer cannot know the time at which the process reaches the delay statement, due to the asynchronous semantics of such languages. When several processes are active, the execution sequence is one of all the possible interleavings of statements of these processes. Therefore, as pointed out in [5], the sequence “delay 1s; delay 5s” is not equivalent to “delay 8s”, unless there is but one active process. Solutions have been proposed to this problem, by adding primitives such as process priorities, but these are very operational, low-level primitives that lead to very complex semantics which still do not provide a complete solution to the problem, since the nondeterminacy is not resolved.

Very recently, better solutions have been given, which consist in constraining asynchronous processes to perform given actions within given intervals of time: constrained Petri nets [6], real-time LUCID [7].

Another approach, which is the one taken in LUSTRE, is based on synchrony. As pointed out by Milner [10], synchrony differs from asynchrony in that several actions can take place “at the same time”. Clearly, this synchronous point of view is closer than the asynchronous one to what actually
happens in an electronic network of operators, since, in that case, there is neither time-sharing nor nondeterministic interleaving of actions. It seems also that the synchronous point of view will render easier the answer to the question “What happens at a given time?” but the problem is now the implementation on a digital computer that performs only one action at a time. The solution, which is well-known by real-time programmers, consists of splitting up processes into steps. When several processes are active, the computer, instead of choosing statements nondeterministically, executes, within a time unit, exactly one step of each active process, and then waits for the next time unit: all the process steps executed within a time unit are then seen as being executed simultaneously. However, this programming approach is usually very long and tedious, the more so as low-level languages are used. It is therefore appealing to design high-level languages that could discharge the programmer from this burden. Attempts in this direction have been made in ESTEREL [5], which is based on synchronous communicating processes, SIGNAL [11], which is based on data-flow and is very close to LUSTRE but for syntactical differences, LTS [2], which is based on ML and aims at hardware description and simulation, and finally LUSTRE, based on synchronous LUCID.

After this (rather long) explanation of the leading principles of LUSTRE, we shall describe the language in a rather informal way, based on examples. More formal descriptions can be found in [4, 3]. We shall first recall the basic notion of LUCID (§1) and the derivation from LUCID to LUSTRE (§2). Structuring tools and examples will be given in §3, and multisampling primitives will be presented in §4.

2 Basic LUCID

It seems that the origin of LUCID comes from an interrogation about statements such as “$X := X + 1$” in conventional languages. Several advantages could be gained if a mathematical meaning, in terms of actual equality, could be given to such statements: easier formal reasoning about programs, proofs and transformations. The solution proposed in LUCID was to consider a variable not as a value at a given program point but, rather, as the sequence of values taken by that variable during the life of the program.

2.1 Variables and constants

Thus, the semantics of a variable $X$ in a LUCID program is an infinite sequence

$$(x_0, x_1, \ldots, x_n, \ldots)$$

of values belonging to the domain specified by the type of $X$. Constants correspond to constant sequences: for instance: `true` means $(true, true, true, \ldots)$ and `1` means $(1, 1, 1, \ldots)$.

2.2 Data operators and equations

Now, expressions on variables and constants may be built using classical operators

- $+ , -, \ast , /$, for real numbers,
- $+ , -, \ast , \text{div}, \text{mod}$, for integers.
- $< , \leq , > , \geq , = , \neq$, for numbers.
- not, or, and, for Booleans.
- conditional operators if ··· then ··· else ···, case ··· of ··· else ···.

considered as operating pointwisely over sequences. New variables can be defined by means of equations. For instance, the LUCID statement

$$X = \text{if } Y > Z \text{ then } Y - Z \text{ else } Z - Y$$

defines $X$ to be the sequence whose $n$-th term is

$$x_n = \text{if } y_n > z_n \text{ then } y_n - z_n \text{ else } z_n - y_n$$

2.3 Sequence operators

Up to now, we cannot write a statement that would be an equivalent of “$X := X + 1$”. This requires sequence operators, and LUCID designers have identified a set of such operators: first, next and followed-by (abridged fby):

- $\text{first}(X)$ means the constant sequence $(x_0, x_0, \ldots, x_0, \ldots)$.
- $\text{next}(X)$ means the sequence $(x_1, x_2, \ldots, x_{n+1}, \ldots)$.
- $X \text{ fby } Y$ means the sequence $(x_0, y_0, y_1, \ldots, y_{n-1}, \ldots)$.

(There is some analogy, but also subtle differences, with the list operators car, cdr and cons of LISP). We can now write statements such as

- $X = \text{first}(Y)$ or
- $\text{next}(X) = X + 1$ or
- $X = 0 \text{ fby } (X + 1)$

This last statement is a recursive equation which defines $X$ to be the sequence of integers $(0, 1, 2, \ldots, n, \ldots)$.

2.4 Further derivations

Other constructs have been defined so as to provide LUCID with the power of a full algorithmic language. These will not be given here, as we do not use them in LUSTRE. In short, LUCID programs appear as sets of equations over sequences, and compilers have been studied that produce sequential code whose result is the unique solution of LUCID equations, if there exists such a solution. There can be several reasons for the lack of such a solution:

- Lack of definition; the statement $X = \text{next}(X) - 1$ cannot define $X$ because the initial value of $X$ is unknown.
- Deadlock: The system $X = \sqrt{Y}; Y = \sqrt{X}$ defines neither $X$ nor $Y$ because the compiler will refuse to look for solutions of the implicit equation $X = \sqrt{\sqrt{X}}$.

However, these problems can be detected at compile time, and the corresponding programs will be rejected.

2.5 The interest of LUCID approach

Let us note two appealing features of the LUCID style:

First, programs consist of sets of equations whose ordering is an arbitrary one and has no special meaning concerning the behaviour of programs. This is why LUCID is called a declarative or nonprocedural language, in contrast with procedural languages which describe sequences of actions.

Second, the sign $=$ in LUCID denotes the standard mathematical equality, thus yielding the so-called “substitution principle”: given the equation $X = Exp$, the identifier $X$ may be replaced by the expression $Exp$ in any context. This is a very useful property in program synthesis and transformation.

3 Basic LUSTRE

3.1 Synchronous interpretation

LUSTRE has been derived from LUCID by giving a temporal interpretation to the LUCID concept of sequence: a LUSTRE program implicitly runs under the control of an external real-time clock, called “the basic clock”, and it is assumed that the programmer (in fact, any people interested in the program behaviour) know the time at which the $n$-th tick of the basic clock takes place. The
temporal interpretation of a LUSTRE program consists of considering that the $n$-th value of the sequence associated with a variable $X$ is the value taken by $X$ between the $n$-th and $(n+1)$-st tick of the basic clock of the program (in fact, we shall see in §4 that this synchronous interpretation may be loosened by defining other clocks than the basic clock).

### 3.2 Sequence operators

As a consequence of the synchronous interpretation, the value of a variable at a given instant cannot depend upon the values of other variables in the future. We call this the causality principle. Instead of checking this principle, we have chosen to forbid the programmer from referencing the future of any variable. This is obtained by replacing the “next” operator by its right inverse, noted “pre”:

$$\text{pre}(X) \text{ means } (\text{nil}, x_0, x_1, \ldots, x_{n-1}, \ldots).$$

where nil is an undefined value which is added to the domain of each variable.

Now, the problem when discarding the operator “next” is that the LUCID construct “fby next” is causal and should be allowed. So, we have chosen to replace the LUCID operator “fby” by “fby next”, abridged $\Rightarrow$, such that

$$X \rightarrow Y \text{ means } (x_0, y_1, \ldots, y_n, \ldots).$$

Since $\text{next(pre}(X)) = X$, the operator fby is no longer primitive, and can be expressed by “$\Rightarrow$ pre”. Moreover, “first” does not seem useful in real-time systems, since $\Rightarrow$ easily allows the initialization of sequences. Therefore, our basic temporal primitives are just $\text{pre}$ and $\Rightarrow$.

Having introduced the undefined value nil, we had to decide what is the result of a data operator when one of its arguments happens to be nil. We decided that this result is nil, but for conditional operators: When the condition of an “if-then-else” construct evaluates to true, the returned value will be that of the “then” part, whatever be the value of the “else” part, and conversely.

### 3.3 First conclusions

Basic LUSTRE is therefore a very simple language, it uses variables and constants, data operators, the sequence operators “pre” and “$\Rightarrow$”, variable declarations and definitions (equations). On this basis, it can be shown that only causal programs can be written, and that these programs only require bounded memory (up to machine representation of integers and real numbers). However, causality does not exclude deadlocks such as illustrated in LUCID. But these deadlocks can be easily checked and rejected at compile time.

### 4 Examples and Structuring Tools

#### 4.1 Nodes and networks

Let us consider the sampled linear first-order filter

$$y_{n+1} = ay_n + bx_{n-1}.$$ 

The corresponding LUSTRE equation is

$$Y = Y_0 \rightarrow a*\text{pre}(Y) + b*X;$$

This equation can be seen as the network of Figure 1. In this network, each operator is a function from sequences to sequences. LUSTRE allows, as in automatic control, the construction of complex block diagrams from simpler ones. This capability is important for increased modularity and ease of programming. It is achieved by defining a “node”, which is a function from its input sequences to its output sequences, built from simpler functions.

The definition of the first-order filter as a node can be written:
node FIRST_ORDER (const A,B,Y0: real; X: real)
returns (Y: real),
let
\[ Y = Y0 \rightarrow A*\text{pre}(Y) + B*X \]
tel;

Thus, a node definition consists of a name, a list of input parameters with their types, a list of output parameters with their types, and a body. Here, the body is simply an equation defining the output parameter by means of the input parameters, but, in general, it may contain local variable declarations and a system of equations defining local variables and output parameters. The keyword “const” indicates that some variables are constant, which will help the compiler in optimizing the code. An adaptive filter could have been defined, by moving \( a \) and \( b \) declarations out of the constant scope, thus letting the filter parameters be variable ones.

### 4.2 A general filter

![Figure 2](image)

This example will illustrate the use of local variables, arrays and the “for···in···” construct.
Let us consider the sampled transfer function

\[ H(z) = \sum_{i=0}^{K} b_i z^{K-i} + \sum_{i=1}^{K} a_i z^{K-i}. \]

A corresponding block diagram is shown in Figure 2. The corresponding LUSTRE node is built very directly from this diagram, by noticing that the LUSTRE counterpart of \( s(z)/z \), with initial value 0, is \( 0 \rightarrow \text{pre}(s) \):

```luster
node FILTER (const K: int; const A,B: array [0..K] of real; E: real)
  returns (S: real);
  var X: array [1..K] of real;
  let
    S = B[0]*E + (0 -> pre(X[1]));
    for I in 1..K-1
      let
        X[I] = B[I]*E - A[I]*S + (0 -> pre(X[I+1]))
      tel;
  tel;
This example illustrates several features of the language:

• Equations have been written very directly, by expressing what is actually true at chosen points of the network, and this has been done in an arbitrary order. This avoids the headache of imperative languages, where one always has to check whether a variable should be assigned before or after its use.

• Arrays and the “for · · · in · · ·” construct allow the definition of sets of variables in a parametric way. These have nothing to do with iterative statements in imperative languages. Iterative statements do not exist in LUSTRE, since they do not meet the declarative style of the language. They could have been replaced, as in other functional languages, by recursive node definitions, but this possibility has not been allowed in LUSTRE: the reason is that it is very difficult to control the execution time of a recursive computation, whereas an accurate evaluation of execution times is an important goal in real-time programming. However, when complex data operators are needed (such as matrix product or array sorting, · · · ), we allow their definition through procedural nodes (called “functions”) written in a host language.

• Finally, a node definition is a self-contained one: the body of a node can only contain equations that bear on variables declared as input, output and local ones. Thus, a node truly defines a function from its input sequences to its output sequences, and it avoids the side effects due to global variables and parameter-passing mechanisms in usual languages.

4.3 Finite-state automata

Deterministic, synchronous, finite-state automata are easily programmed in LUSTRE thanks to Boolean and scalar variables, conditional and Boolean operators. For instance, let us consider the mode logic of an automatic control system. Two buttons are provided to the operator; when the “manual” button is pushed, the mode is manual; when the “auto” button is pushed, the mode is preset, until a given condition raised by the numerical controller becomes true, indicating that the mode is automatic. The “manual” light is on in manual and preset mode, off in automatic mode. The “auto” light is off in manual mode, red in preset mode, and green in automatic mode. The initial mode is manual. A corresponding LUSTRE node could be:
node LOGIC_MODE (MBUTN, ABUTN, COND: bool)
returns (MANLIGHT: bool;
AUTOLIGHT: {OFF,RED,GREEN};
MODE: {MAN,PRESET,AUTO});
let
    MODE = MAN ->
        if MBUTN then MAN
        else case pre(MODE) of
            MAN : if ABUTN
                then if COND then AUTO else PRESET
                else MAN;
            PRESET : if COND then AUTO else PRESET;
            else : pre(MODE);
    MANLIGHT = not (MODE = AUTO);
    AUTOLIGHT = case MODE of
        PRESET : RED;
        AUTO : GREEN;
        else : OFF;
tel;

4.4 Using nodes: The functional style

Once defined, nodes can be used in expressions like predefined operators. For instance:

Y = FIRST_ORDER(A, B, XINIT, X);
Z = FIRST_ORDER(C, D, YINIT, Y);
T = Z + X;

corresponds to the network of Figure 3. Thanks to the substitution principle, Z could have been defined by

Z = FIRST_ORDER(C, D, YINIT, FIRST_ORDER(A, B, XINIT, X))

When a node has several outputs. its output is intended to be a tuple: for instance, one can write

(ML, AL, M) = LOGIC_MODE(MK, AK, C)

Let us show the architecture of a full program, that mixes numerical control with mode logic: in auto mode, a numerical controller elaborates a control signal as a function of a sensor signal from the process and of a command signal from the operator, while in other modes the operator controls directly the process through the computer. A condition node computes a condition indicating that the controller is able to control the process.
program AUTO_PILOT;

type TYPLIGHT = {OFF, RED, GREEN};
   TYPMODE = {MAN, PRESET, AUTO};

function READ_SENSOR () returns (SENSOR: real);
   proc .... corp;

function READ_COMMAND () returns (COMMAND: real);
   proc .... corp;

function READ_CONTROL () returns (CONTROL: real);
   proc .... corp;

function READ_BUTN () returns (MBUTN, ABUTN: bool);
   proc .... corp;

function EMIT_LIGHT (MAN_LIGHT: bool;
   AUTO_LIGHT: TYPLIGHT)
   returns (DUMMY: any); proc .... corp;

function EMIT_CONTROL (CONTROL: real)
   returns (DUMMY: any); proc .... corp;

node MODE_LOGIC (MBUTN, ABUTN, COND: bool)
   returns (MANLIGHT: bool;
      AUTO_LIGHT: TYPLIGHT;
      MODE: TYPMODE);
   let .... tel;

node CONTROLLER (COMMAND, SENSOR: real)
   returns (CONTROL: real);
   let .... tel;

node CONDITION (COMMAND, SENSOR: real)
   returns (COND: boolean);
   let .... tel;

var SENSOR, COMMAND, CONTROL: real;
   MANLIGHT: bool;
   AUTOLIGHT: TYPLIGHT;
   MODE: TYPMODE;
   DUMMY1, DUMMY2: any;

let
   COMMAND = READ_COMMAND();
   SENSOR = READ_SENSOR();
   (MANLIGHT, AUTOLIGHT, MODE) =
      MODE_LOGIC(READ_BUTN(), CONDITION(COMMAND, SENSOR));
   CONTROL = if (MODE = AUTO)
          then CONTROLLER(COMMAND, SENSOR)
          else READ_CONTROL();
   DUMMY1 = EMIT_CONTROL(CONTROL);
   DUMMY2 = EMIT_LIGHT(MANLIGHT, AUTOLIGHT)
tel;
Several remarks must be made about this program:

- A program describes a closed system that communicates with the environment through functions, written in an host language, which are able to address computer input/output devices. This allows a clean separation between functional behaviour and input/output. Notice that, like in other functional languages, output appears as a side-effect of a functional node, issuing a dummy variable of universal type “\texttt{any}”.

- The strong typing provides many interface checks, and allows many errors to be detected at compile time.

- The main program merely expresses the composition of elementary nodes and there is a large versatility of styles, balancing readability and conciseness. For instance, the variables \texttt{COMMAND} and \texttt{SENSOR} could have been replaced by their corresponding \texttt{READ} expressions. Conversely, the output of the controller could have been identified as a variable. This means also that the controller node is permanently running at the basic clock rate, whatever be the mode. In the next section, we shall introduce primitives allowing nodes to run intermittently.

5 Multisampling Primitives

There can be several reasons for having different clocks in a control system. One is to have nodes running intermittently, as a function of node variables. Another one appears when fast and slow processes coexist in a given system. This could be done in LUSTRE, as defined so far, by providing a node with a Boolean variable that freezes the node functioning when its value is false. However this is a rather awkward programming style since it involves the reprogramming of nodes when they are to be used at different rates. Furthermore it does not allow a clean separation between numerical control nodes and mode logic nodes. An alternative solution to the problem arises from the very definition of a node: it is a sequence function that runs at the rate of its input sequences. Thus it would be possible to modify the rate of a node without changing its definition, if we had some means of modifying the rate of its input sequences. In the sequel, we propose two operators in order to achieve this goal.

5.1 Sampling

If \( E \) is an expression of any type and \( C \) is a Boolean variable, then “\( E \texttt{ when } C \)” is an expression whose sequence of values is extracted from the sequence of \( E \), by taking only those values which occur when \( C \) is true. In other words, the \( n \)-th term of “\( E \texttt{ when } C \)” is the \( m \)-th term of \( E \), where \( m \) is such that \( c_m \) is the \( n \)-th true value of \( C \). The following table, where \( \texttt{tt} \) and \( \texttt{ff} \) stand for true and false and \( X = E \texttt{ when } C \), illustrates this operation:

<table>
<thead>
<tr>
<th>( C )</th>
<th>\texttt{tt}</th>
<th>\texttt{ff}</th>
<th>\texttt{tt}</th>
<th>\texttt{tt}</th>
<th>\texttt{ff}</th>
<th>\texttt{ff}</th>
<th>\texttt{tt}</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E = )</td>
<td>( e_1 )</td>
<td>( e_2 )</td>
<td>( e_3 )</td>
<td>( e_4 )</td>
<td>( e_5 )</td>
<td>( e_6 )</td>
<td>( e_7 )</td>
</tr>
<tr>
<td>( X = )</td>
<td>( e_1 )</td>
<td>( e_3 )</td>
<td>( e_4 )</td>
<td>( e_7 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thus \( E \texttt{ when } C \) is a sequence that has at slower rate than \( E \) and \( C \). For instance, we can have a first-order node running at half the basic clock frequency:

\[
\text{C = true -> not pre(C);}
\]
\[
\text{Y = FIRST ORDER((A, B, XINIT, X) when C);}
\]

Similarly, we can make our numerical controller work only in preset and auto mode:

\[
\text{ACONTROL = CONTROLLER((COMMAND, SENSOR) when NOMAN);}
\]
\[
\text{NOMAN = not(MODE = MANUAL);}
\]

The introduction of the operator “\texttt{when}” has several consequences.
• Now, every variable and expression has a clock, which is either the basic clock or a Boolean variable. Thus the semantics of an expression is twofold: it consists of a sequence of values together with a clock. In the above example, the clock of \( \text{ACONTROL} \) is the Boolean variable \( \text{NOMAN} \).

• Nodes and operators are not only functions from sequences to sequences but also from input clocks to output clocks.

• Strong conditions are needed for applying nodes and operators to variables with distinct clocks: for instance, the equation

\[
\text{CONTROL} = \text{if (MODE = AUTO) then ACONTROL else MANCONTROL};
\]

makes no sense since the \( n \)-th value of \( \text{ACONTROL} \) does not take place at the same time as the \( n \)-th value of the other variables involved in the equation, thus possibly violating either the causality principle or the bounded memory principle. Those conditions, and a clock calculus for checking them at compile time, are described in detail in [3]. In valid programs, a variable has a unique clock, and from the knowledge of clock behaviours, the programmer can know precisely at which time a variable takes its \( n \)-th value.

5.2 Projection

The question is now to use \( \text{ACONTROL} \), whose clock is \( \text{NOMAN} \), so as to define the output \( \text{CONTROL} \), whose clock is the basic clock. This is the role of the operator “current”, the effect of which is to “project” an expression of clock \( C \) so as to get an expression whose clock is the clock of \( C \) (as clocks are Boolean variables, they have clocks too). This operation is illustrated by the following table:

<table>
<thead>
<tr>
<th>( C )</th>
<th>( tt )</th>
<th>( ff )</th>
<th>( tt )</th>
<th>( ff )</th>
<th>( ff )</th>
<th>( tt )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Z )</td>
<td>( z_1 )</td>
<td>( z_2 )</td>
<td>( z_3 )</td>
<td>( z_4 )</td>
<td>( z_5 )</td>
<td>( z_6 )</td>
</tr>
<tr>
<td>( X = \text{when } C(Z) )</td>
<td>( z_1 )</td>
<td>( z_3 )</td>
<td>( z_4 )</td>
<td>( z_7 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Y = \text{current}(X) )</td>
<td>( z_1 )</td>
<td>( z_1 )</td>
<td>( z_3 )</td>
<td>( z_4 )</td>
<td>( z_4 )</td>
<td>( z_4 )</td>
</tr>
</tbody>
</table>

We are now able to solve our problem by writing

\[
\text{CONTROL} = \text{if (MODE = AUTO) then current(ACONTROL) else READCONTROL();}
\]

This equation is now valid concerning clock consistency, since all expressions involved in the if–then–else construct have the same clock (the basic clock).

6 Conclusion

Our main objective was to define a simple and formally well-founded real-time language that could help in achieving reliable programming in this critical field. Simplicity has been gained through a very reduced number of primitives, while formal foundation is asserted by the existence of a simple denotational semantics [3], based on fixed-points of sequence functions, in the style of [8].

Current activities about LUSTRE present several aspects:

• A prototype compiler, producing sequential code in C, is being written and we may expect to be able to experiment practical applications by the time of this conference.

• We are studying the relations between LUSTRE Boolean variables and temporal logic. Automatic theorem provers for this kind of logic are currently being developped at several places, and thus we may expect to prove properties of programs in a very standard way.

• Finally, some experiments are undertaken about the use of LUSTRE for the description and safe design of hardware systems, mainly in connection with the logical design of integrated circuits.
References


