LUSTRE: A declarative language for programming synchronous systems*

P. Caspi D. Pilaud
Laboratoire "Circuits et Systèmes"

N. Halbwachs J. A. Plaice
Laboratoire de Génie Informatique

BP68, 38402 St Martin d'Hères, FRANCE

Received 10/15/86

Abstract

LUSTRE is a synchronous data-flow language for programming systems which interact with their environments in real-time. After an informal presentation of the language, we describe its semantics by means of structural inference rules. Moreover, we show how to use this semantics in order to generate efficient sequential code, namely, a finite state automaton which represents the control of the program. Formal rules for program transformation are also presented.

Introduction

This paper presents the language LUSTRE, whose main application field is the programming of automatic control and signal processing systems. In this field, design is traditionally driven by means of two types of tools. First, specifications are often systems of equations (differential or finite difference equations, boolean equations, ...). Second, implementations are often nets of operators connected with wires (switches, gates, analog diagrams). Such tools present several advantages as a basis for a programming language.

- Systems of equations are mathematically tractable objects. In such systems, variables are interpreted in the mathematical sense, without any notion of assignment, side effect, etc., often carried by variables in usual programming languages. An equation is then an invariant assertion, true *at each instant.*
- Both in systems of equations and in operator nets, there is neither the notion of control nor that of sequentiality. The only constraints on the evaluation order arise from the dependencies between variables. As a consequence, any implementation, be it sequential or highly parallel, can be easily derived.
- As pointed out above, the considered equations are generally time invariant: variables may be considered to be functions of time, and X=E means that at each instant t, $x_t = e_t$. Hence, such models are likely to provide a simple and natural way of handling time, a problem which is never adequately solved in usual languages, in spite of the work increasingly devoted to it.

LUSTRE is a programming language founded on these remarks. A program is a system of equations defining variables, which are functions from time to their domain of values. Since we are concerned with discrete systems, time is projected onto the set of naturals, making variables infinite sequences of values. Furthermore, a program may be viewed as an operator net, as is standard for data-flow languages, with a further assumption called *synchrony*, which states that operators respond instantaneously to their input.

Our equational point of view may be summarized by the two following principles:

Substitution principle An equation X=E specifies a full synonymy between the variable X and the expression E. Thus, in every context, the identifier X may be replaced by the expression E, and conversely. This property is very useful in program transformation.

Definition principle Let X=E be the equation defining the variable X. Then the behavior of X must be completely specified by this equation and the behavior of variables appearing in the expression E.

A program defines a function from its input (sequences) to its output (sequences). From the assumption of synchrony, all functions expressible in the language must satisfy the following properties:

*This work was partially supported by a grant from PRC-C³ (CNRS).

Causality The output at any instant t may only depend upon input received before or at t. Notice that in this sense, LUCID [1], a close parent of LUSTRE, allows the definition of uncausal programs.

Bounded memory There must exist a finite bound such that, at each instant, the number of past input values that are necessary to produce the present and future output values remains smaller than that bound.

Moreover, in order to be usable for real-time programming, the language must present the following features:

Efficient code generation possibility This feature entails that most consistency checks must be possible at compile time. Moreover, we shall see that very efficient code can be produced from LUSTRE programs, through the construction of finite automata, as done for the language ESTEREL [7].

Execution time predictability It is necessary to be able to verify the realism of the synchrony assumption, which holds only when the response time of the program is negligible with respect to the reaction time of its environment. This constraint forbids the use of unbounded loops and recursive functions.

Finally, in the considered application field, program reliability is especially important. This goal has been approached in LUSTRE by two means:

Strong consistency checks allowed by a coercive syntax and strict rules of static semantics.

Simple formal semantics It minimizes the risk of program misunderstanding. In particular, it must be perfectly clear *when* a variable changes.

The language is informally presented in Section 1, and its use is illustrated in Section 2 through the programming of a stopwatch. Section 3 presents the most important aspects of the formal semantics of LUSTRE. It is showed in Section 4 how this semantics may be used to produce sequential code from LUSTRE programs. Section 5 is devoted to formal transformations of programs, which benefit from the mathematical nature of the language. In conclusion, LUSTRE is compared with related languages and formalisms and future work is outlined.

1 Informal presentation

1.1 Variables, Equations, Data Operators

As indicated above, any variable or expression in LUSTRE denotes an infinite sequence of values. Variables are defined by means of equations: if `X` is a variable and `E` is an expression, the equation `X=E` defines `X` to be the sequence

$$(x_0 = e_0, x_1 = e_1, \ldots, x_n = e_n, \ldots),$$

where $(e_0, e_1, \ldots, e_n, \ldots)$ is the sequence of values of the expression `E`.

Expressions are built up from variables, constants (considered to be infinite constant sequences) and operators. The usual arithmetic, boolean and conditional operators on values are extended to pointwisely operate over sequences, and are hereafter referred to as *data operators*. For instance, the expression

```
if X>Y then X-Y else Y-X
```

denotes the sequence whose n-th term is the absolute difference of the n-th values of `X` and `Y`.

1.2 Sequence Operators

In addition to data operators, LUSTRE contains only four non-standard operators, called *sequence operators*, which actually manipulate sequences.

To keep track of the value of an expression from one cycle to the next, there is a memory or delay operator called `pre` ("previous"). If

$$\mathtt{X} = (x_0, x_1, \ldots, x_n, \ldots),$$

then

$$\mathtt{pre(X)} = (\mathtt{nil}, x_0, x_1, \ldots, x_{n-1}, \ldots),$$

where `nil` is an *undefined* value, akin to the value of an uninitialized variable in imperative languages. The compiler ensures that no data operator is ever applied to `nil`.

To initialize variables, the `->` ("followed by") operator is introduced. If

$$\mathtt{X} = (x_0, x_1, \ldots, x_n, \ldots)$$

and

$$\mathtt{Y} = (y_0, y_1, \ldots, y_n, \ldots)$$

are two variables (or expressions) of the same type, then

$$\mathtt{X->Y} = (x_0, y_1, y_2, \ldots, y_n, \ldots),$$

i.e., `X->Y` is equal to `Y` except at the first instant.

As an example of use of these operators, the equation

```
X = 0 -> pre(X) + 1;
```

defines `X` to be 0 initially, and its previous value incremented by 1 subsequently. Hence, `X` is the sequence of naturals.

The last two operators require some introduction. Up to now, one can consider a program to have a cyclic behavior, namely computing at its n-th cycle the n-th value of every variable. However, if we are to consider variables whose values only make sense under some condition, it becomes necessary to be able to define variables which are not computed at every cycle. We thus introduce the sampling operator `when`.

Let `E` be an expression and `B` be a boolean expression. `E when B` is then an expression whose sequence of values is extracted from the sequence of `E` by taking only those values which occur when `B` is true. The point is that `E when B`

E =(	e_0	e_1	e_2	e_3	e_4	e_5	...)
B =(	tt	ff	tt	tt	ff	ff	...)
X=E when B =(	$x_0 = e_0$		$x_1 = e_2$	$x_2 = e_3$			...)

Table 1: The when operator

B =(	tt	ff	tt	tt	ff	ff	...)
E =(	e_0	e_1	e_2	e_3	e_4	e_5	...)
X = E when B =(	e_0		e_2	e_3			...)
Y = current(X) =(	e_0	e_0	e_2	e_3	e_3	e_3	...)

Table 2: The current operator

does not "have the same notion of time" as E and B, as shown by Table 1. Now, several remarks must be made about this operation.

In the above example, X is said to be computed on *clock* B. This means that the only notion of time that X "knows" is the sequence of cycles where B is true. As a consequence, the question "what is the value of X when B is not true?" makes no more sense than the question "what is the value of a variable between two integer instants?".

Our model of variables consisting of sequences is no longer sufficient: two variables may describe the same sequence of values without being equal. Henceforth, a variable will be characterized not only by its sequence of values, but also by its clock. Let us call a *stream* the couple formed by a sequence and a clock. Streams and clocks are recursively defined as follows:

<clock> ::= true | <boolean stream>
<stream of type T> ::= <sequence of type T><clock>

Intuitively, if the stream associated with an expression is of clock true, then the expression is renewed with the basic cycle of the program. Constants are always assumed to be on the basic clock. A non basic clock is defined by a boolean expression, which in turn has a clock. Thus clocks may be nested: for instance, the millisecond can be modeled as a clock (a boolean variable which is true at each *tick* of a quartz) and the second can be another clock defined on the millisecond clock. Moreover, this example shows that the when operation allows the basic cycle to be quite unrelated to physical time, which can be handled as an input to the program.

Now, suppose that we wish to apply an operator on expressions with different clocks (e.g., to sum X and E in the example of Table 1). Since an operator operates on terms of the same rank, and since these terms can exist at different instants, either the causality or the bounded memory condition could be violated. In order to operate on expressions with different clocks, we must first put them on the same clock, either by sampling (when) or by projecting, using our last operator, current.

If E is an expression of clock B, then current(E) is an expression whose clock is the same as that of B and whose value at each cycle of this clock is the value taken by E at the last cycle when B was true. Table 2 illustrates the combination of the when and current operators. The current operator allows operations over variables of different clocks, since if X and X' are variables of respective clocks B and B', and if B and B' have the same clock,

current(X) op current(X')

is a legal expression for every binary operator op.

1.3 Nodes and Nets

A node is a LUSTRE subprogram. It receives input variables, computes output variables, and possibly local variables, by means of a system of equations. For instance, we can define a general counter as follows:

```
node COUNT (init, incr: int; reset: bool)
   returns (n: int);
let
   n = init ->
       if reset then init else pre(n) + incr;
tel;
```

Node instantiation takes a functional form: if N is the name of a node declared with heading

node N ($i_1 : \tau_1$; $i_2 : \tau_2$; ...; $i_p : \tau_p$)
returns ($j_1 : \theta_1$; $j_2 : \theta_2$; ...; $j_q : \theta_q$);

and if $E_1, \ldots, E_p$ are expressions of type $\tau_1, \ldots, \tau_p$, then the instantiation N($E_1, \ldots, E_p$) is an expression of type tuple($\theta_1, \ldots, \theta_q$) whose n-th value is the tuple ($j_{1_n}, \ldots, j_{q_n}$) computed by the node from input parameters $E_1, \ldots, E_p$. Conditional and sequence operators are polymorphic, and can be applied to tuples. Coming back to the general counter, one may write

```
even = COUNT(0, 2, false);
mod5 = COUNT(0, 1, pre(mod5=4));
```

thus defining even to be the sequence of even numbers and mod5 to be the cyclic sequence of integers modulo 5.

Concerning clocks, and in agreement with the data-flow philosophy, the basic execution cycle of a node is determined by the clock of its input parameters. As an example, the instantiation

B =(	*tt*	*ff*	*tt*	*ff*	*tt*	...)
COUNT((0,1,false) when B) =(	0		1		2	...)
COUNT(0,1,false) when B =(	0		2		4	...)

Table 3: Sampling input vs. sampling output

```
COUNT((0, 1, false) when B)
```

counts whenever B is true. Now the when operator does not distribute over sequence operators, so synchronyzing the execution of a node by sampling its input generally differs from sampling its output, as shown by Table 3.

A node may receive input parameters with different clocks, but when the clock of a parameter is not the basic clock, this clock must be passed as a parameter. For instance, one can declare a node with heading

```
node N (millisecond: bool;
        (second: bool) when millisecond)
    returns ...
```

This header means that the first parameter (whose clock is always the basic clock of the node) is the clock of the second parameter. A node may also return parameters with different clocks, provided these clocks be visible from outside the node.

This mechanism for declaring and instantiating nodes allows the definition of whole nets of parallel, synchronous operators. In fact, any operator may be considered to be a node: for instance, the equation

```
X = 0 -> pre(X) + 1;
```

describes the net in Figure 1.

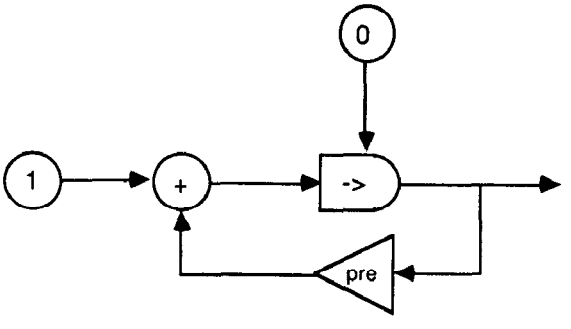

Figure 1: An operator net

1.4 Practical issues

Of course, the language is much too simple to be practically usable. In the design of LUSTRE, we focused on temporal aspects, thus forsaking standard topics, such as type mechanisms, input/output facilities, etc. Moreover, we do not consider the data-flow approach as dogma. There are indeed many programs which are easier to write in an imperative style. For all these reasons, there is the possibility in a LUSTRE program to call external functions written in a host language (currently C, which is also the target language of our compiler). These functions are treated as data-operators which operate in null-time. They are also used for interfacing programs: a LUSTRE program calls its environment as a function (e.g. "teletype" or "sensor") which takes as inputs the outputs of the program, and/or returns its inputs. One can also use external types, whose members are handled by means of functions.

2 An Example

Let us illustrate the use of our language through the programming of a stopwatch (other examples may be found in [5,6]). We begin with a simplified version.

The stopwatch receives three signals: a `start-stop` button, a `reset` button, and the 1/100 sec. `hs` emitted from a quartz. It computes the time as follows: the time is initially 0, and is reset to 0 whenever the `reset` button is pushed. It is incremented at each 1/100 sec. when the stopwatch is in the `running` state. Initially the stopwatch is in the `not running` state, and the state changes whenever the `start-stop` button is pushed.

All the events (buttons, 1/100 sec.) will be implemented as boolean variables: the variable is true when and only when the event occurs. The integer output `time` will be computed by the node `COUNT` defined above, with parameters `0`, `1`, `reset`, sampled according to a suitable clock `CK`. So, `time` will be the projection onto the basic clock of the expression

```
COUNT((0, 1, reset) when CK);
```

Now, the node `COUNT` must perform a cycle at the initial instant, and whenever either the `hs` event occurs in the `running` state, or the `reset` event occurs. So, the clock CK is as follows

```
CK = true -> (HS and running) or reset;
```

The `running` state variable remains to be defined. It is initially false and changes whenever the `start-stop` event

occurs. We use a node TWO_STATES, of general usage, which takes as parameters an initial value and two events, set and reset, and returns a boolean value.

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

running =
 TWO_STATES(false, start_stop, start_stop);
```

The complete program of this first version of the stopwatch is as follows:

```
node SIMPLE_STOPWATCH
     (start_stop, reset, hs: bool)
        returns (time: int);
var CK, running: bool;
let
  time =
   current(COUNT((0, 1, reset) when CK));
  CK = true -> (HS and running) or reset;
  running =
   TWO_STATES(false, start_stop, start_stop);
tel;
```

Now, let us consider a more realistic stopwatch: it has a lap button for handling intermediate time by freezing the display. The stopwatch computes two times.

- an internal time, computed as before;
- a displayed time, which is equal to the internal time when the stopwatch is not in lap mode, and which is frozen when the lap mode is entered.

The stopwatch is initially in not_lap mode. The lap mode is entered whenever the lap button is pushed in running state and left the next time the lap button is pushed.

Moreover, the reset event for the internal time corresponds now to a pushing of the lap button in not running state and in not lap mode.

The displayed time is always equal to the value of the internal time the last time the stopwatch was not in lap mode:

```
disp_time = current(int_time when not_in_lap);
```

The mode is again described using the TWO_STATES node:

```
not_in_lap =
   TWO_STATES(true, lap, lap and running);
```

It remains to define int_time:

```
int_time =
  SIMPLE_STOPWATCH(start_stop, reset, hs);
reset = lap and pre(not_in_lap)
           and pre(not running);
```

and the variable running, as before, since it is not exported by the node SIMPLE_STOPWATCH.

3 Operational Semantics

Although the first semantics written for LUSTRE [4] was denotational à la Kahn [16], we shall present the operational one which has been used as a basis for the construction of our compiler. We use structural inference rules à la Plotkin [20]. While this semantics is quite simple, it cannot be thoroughly presented in a single paper (a complete description may be found in [9]). We only present its most illustrative features: the clock consistency and the kernel of the dynamic semantics.

3.1 Clock Rules

As mentioned above, a clock may be statically associated with each expression of a program. The clock consistency rules essentially state that operands of any operator must be on the same clock. The rules are checked at compile time.

First, we must make precise what we mean by "on the same clock". Ideally, it would refer to equality of boolean streams. However, static checking of the equality between two boolean variables being undecidable, we are led to consider finer equivalences on clocks. Let $\equiv$ be some equivalence relation between boolean expressions such that two equivalent expressions denote the same stream. In this subsection, we consider clocks to be the equivalence classes of the relation $\equiv$. In the current version, the relation $\equiv$ is the syntactic identity of identifiers. Notice that this definition seems to violate the substitution principle, since replacing the second operand of a when operator by its definition may result in changing its clock. We do admit that we allow substitution to change the result of consistency checks, but once these checks have been passed, the dynamic semantics will not be affected by substitution.

We define a clock environment ω to be a function from identifiers to clocks. The clock environment associates with each variable the clock of the right-hand side of its definition. Let CK(exp,ω) be the clock of the expression exp in the environment ω. From an equation X = exp, we shall deduce

$$\omega(X) = CK(exp, \omega).$$

Hence, the definition of ω is recursive, and ω will be defined as a solution of a fix-point equation. However, not every solution of this equation is suitable, as is shown by the following example. Let M and N be two nodes returning output parameters on the same clock as their input parameters, and consider the program

```
X = M(Y);  Y = N(X);
```

The only information that we can acquire about the clocks of X and Y is

$$\omega(X) = \omega(Y)$$

which is clearly insufficient to give a determined meaning to the program. Now consider the program

```
X = M(Y);  Y = N(X);  Z = X+1;
```

Since the constant 1 is on the basic clock and the operator + must be applied to operands on the same clocks, the only solution which makes sense of the program is

$$\omega(X) = \omega(Y) = \omega(Z) = \texttt{true}(\text{the basic clock})$$

However, accepting such a solution would violate the definition principle ("the behavior of a variable is completely specified by its definition"), since it would deduce the clock of X from its *use*, rather than from its *definition*. Thus our rules must reject such a program.

We shall define an ordering relation among clocks, so that the environment be the least solution of the fix-point equation. The set of clocks is structured as a flat lattice as follows

$$ck \leq ck' \Leftrightarrow (ck = \bot \vee ck' = \top \vee ck \equiv ck'),$$

where $\bot$ and $\top$ must be respectively interpreted as the undefined clock and the erroneous clock. Let us denote the least upper bound operator in this lattice by $\bigcup$.

Now the clock environment associated with a system of LUSTRE equations is the least solution of the associated system of clock equations. The program is correct with respect to its clocks if the range of this environment does not intersect the set $\{\bot, \top\}$.

We must now define the function CK, which gives the clock of an expression in a given environment. The rules are as follows.

Constants

For any constant k, $\texttt{CK}(k, \omega) = \texttt{true}$.

Variables

For any identifier X, $\texttt{CK}(X, \omega) = \omega(X)$.

Synchronous operators

For any $op \notin \{\texttt{when}, \texttt{current}\}$,

$$\texttt{CK}(op(exp_1, exp_2, \ldots, exp_n), \omega) = \bigcup_{i=1}^{n} \texttt{CK}(exp_i, \omega)$$

Sampling

The operands of the when operator must be on the same clock:

$$\frac{\texttt{CK}(exp, \omega) \bigcup \texttt{CK}(ck, \omega) \neq \top}{\texttt{CK}(exp \texttt{ when } ck, \omega) = ck}$$

Projection

One may only project an expression which is not on the basic clock:

$$\frac{\texttt{CK}(exp, \omega) = ck, \quad ck \neq \texttt{true}}{\texttt{CK}(\texttt{current}(exp), \omega), \texttt{CK}(ck, \omega)}$$

The rules for nodes are more complicated. We only outline the algorithm for finding the clock (generally a tuple of clocks) of a node instantiation:

The analysis of a node declaration provides a local environment which subsumes:

- the relations between the clocks of input parameters;
- the clocks of local variables and output parameters, computed under the assumption that the first input parameter is on the clock true (since its clock defines the basic clock of the node)

The restriction of this local environment according to (input and output) parameters must range over parameters (since the clock of an output parameter must be visible from outside the node).

When the node is called, this restricted environment is considered in order to

- check that the actual input parameters are on suitable clocks;
- compute the clocks of output parameters, by renaming formal by actual parameters, and renaming the clock true by the clock of the first actual input parameter.

3.2 Dynamic Semantics

For simplicity, and without loss of generality, we define the dynamic semantics on a very simplified *basic syntax*, defined as follows:

```
prog ::= eqs
eqs ::= eq | eq;eqs
eq ::= id=(ck)exp
exp ::= sexp | sexp->sexp | k fby sexp | curr(sexp,k)
sexp ::= k | id | dop(sexp,...,sexp)
ck ::= exp | true
```

where k stands for a constant and dop stands for any data operator.

It can be shown that any LUSTRE program which is correct according to the static semantics can be translated into that syntax. This translation consists of

- computing the clocks;
- expanding nodes, by replacing each node instantiation by its body, after suitable instantiation of parameters and clocks, and renaming of local variables. Thus we only consider flat programs (systems of equations);
- translating pre(exp) and current(exp) into nil fby exp and curr(exp, nil) respectively (fby is the LUCID "followed by" operator; it is equivalent to -> pre);

- introducing auxiliary variables and performing suitable substitutions so that the right-hand side of each equation contains at most one sequence operator;
- indicating the clocks in the equations (true stands for the basic clock of the program). The when operators are then redundant, and may be dropped.

3.2.1 Semantic Domains

Let us define a memory σ to be a function from identifiers to values, and a history h to be a sequence of memories. A memory associates $\perp$ with an identifier when the corresponding variable does not have to be computed (its clock is false or does not have to be computed). A memory will give the values of variables at a given cycle. A system of equations eqs is *compatible* with a memory σ if the first cycle of evaluation of eqs associates the value $\sigma(X)$ with each identifier X defined in eqs. The semantics of a program is the transformation

$$\text{(input history)} \Longrightarrow \text{(output history)}$$

it computes.

We shall not give the semantics of simple expressions (sexp) as they are obvious. Let us define the following predicates.

$\sigma \vdash \texttt{sexp} : \texttt{k}$ In the memory σ, the simple expression exp evaluates as k.

$\sigma \vdash \texttt{exp} \xrightarrow{\texttt{k}} \texttt{exp}'$ In the memory σ the expression exp evaluates as k, and exp will be later on evaluated as exp'.

$\texttt{eq} \xrightarrow{\sigma} \texttt{eq}'$ The equation eq is compatible with the memory σ and will later on be considered as eq'.

$\texttt{eqs} \xrightarrow{\sigma} \texttt{eqs}'$ The system of equations eqs is compatible with the memory σ, and will later on be considered as eqs'.

$h \vdash \texttt{eqs} : h'$ From its input history h, the program defined by the system of equations eqs produces the output history h'.

3.2.2 Rules

Programs

$$\frac{\texttt{eqs} \xrightarrow{\sigma} \texttt{eqs}', \quad h \vdash \texttt{eqs}' : h'}{\sigma[\text{input}].h \vdash \texttt{eqs} : \sigma[\text{output}].h'}$$

where σ[input] and σ[output] respectively denote the restriction of σ to the input and output variables of the program.

Systems of equations

$$\frac{\texttt{eq} \xrightarrow{\sigma} \texttt{eq}', \quad \texttt{eqs} \xrightarrow{\sigma} \texttt{eqs}'}{\texttt{eq;eqs} \xrightarrow{\sigma} \texttt{eq}'\texttt{; eqs}'}$$

Equations

If the clock is true, the right-hand expression is evaluated and its value is associated with the variable on the left-hand side.

$$\frac{\sigma(\texttt{ck}) = tt, \quad \sigma \vdash \texttt{exp} \xrightarrow{\texttt{k}} \texttt{exp}', \quad \sigma(\texttt{id}) = \texttt{k}}{\texttt{id=(ck)exp} \xrightarrow{\sigma} \texttt{id=(ck)exp}'}$$

If the clock is not true, the left-hand variable is not evaluated.

$$\frac{\sigma(\texttt{ck}) \neq tt, \quad \sigma(\texttt{id}) = \perp}{\texttt{id=(ck)exp} \xrightarrow{\sigma} \texttt{id=(ck)exp}}$$

These rules define σ to be the solution of a fixpoint equation. Moreover, this solution must be unique (otherwise the program contains a *deadlock*; this problem will be detailed in section 4.1).

Expressions

Simple expressions are always evaluated in the same manner.

$$\frac{\sigma \vdash \texttt{sexp} : \texttt{k}}{\sigma \vdash \texttt{sexp} \xrightarrow{\texttt{k}} \texttt{sexp}}$$

The result of a -> operator is the value of its first operand. The expression will henceforth be evaluated as the second operand.

$$\frac{\sigma \vdash \texttt{sexp}_1 : \texttt{k}}{\sigma \vdash \texttt{sexp}_1\texttt{->sexp}_2 \xrightarrow{\texttt{k}} \texttt{sexp}_2}$$

The result of a fby operator is the value of its first operand. The result of the second operand is stored in the expression to be evaluated later on.

$$\frac{\sigma \vdash \texttt{sexp} : \texttt{k}_1}{\sigma \vdash \texttt{k fby sexp} \xrightarrow{\texttt{k}} \texttt{k}_1 \texttt{ fby sexp}}$$

If an expression is evaluated, its current value is the result of the expression, which must be stored for further evaluations. If the expression is not evaluated, its current value is the stored value.

$$\frac{\sigma \vdash \texttt{sexp} : \texttt{k}_1, \quad \texttt{k}_1 \neq \perp}{\sigma \vdash \texttt{curr(sexp,k)} \xrightarrow{\texttt{k}_1} \texttt{curr(sexp,k}_1\texttt{)}}$$

$$\frac{\sigma \vdash \texttt{sexp} : \perp}{\sigma \vdash \texttt{curr(sexp,k)} \xrightarrow{\texttt{k}} \texttt{curr(sexp,k)}}$$

4 Compilation of LUSTRE

This section deals with the most specific features of the compilation of LUSTRE.

4.1 Variable dependencies and deadlock detection

As usual in non-procedural languages, the only constraints on the ordering of computations in LUSTRE result from the dependencies between variables: a variable X instantaneously depends on Y at a given cycle if the value of Y at this cycle must be known in order to compute the value of

X. At any cycle, this relation must be irreflexive: as mentioned in the semantics of equations, the memory must be the *unique* fixpoint of a function, and this unicity is only ensured when no variable instantaneously depends on itself: of course, we don't intend to solve implicit equations such as

```
X = X**2 + 1;
```

Such a situation, which is called a *deadlock* in the terminology of LUCID, may be difficult to detect. So, in the present compiler, we are led to consider a coarser, static dependency relation, which is the transitive closure of the relation "the variable Y appears outside any pre operator in the expression defining X". This relation is easy to build from the text of the program, and when it is not irreflexive, the program will be rejected. Clearly, this approximation ensures the detection of any deadlock, but may result, in some cases, in rejecting correct programs, such as the following one:

```
X = if C then Y else Z;
Y = if C then Z else X;
```

4.2 Sequential code generation

4.2.1 Node expansion

First of all, let us notice, following a remark by G. Gonthier [11], that in synchronous languages, sequential code cannot be obtained in a modular way. This means that we cannot produce sequential code for a LUSTRE node independently of the instantiations of that node. For instance, consider the following, very simple node, which only lets its two inputs traverse it:

```
node DUMMY (X,Y: int) returns (X1,Y1: int);
let X1 = X; Y1 = Y; tel;
```

Clearly, for any execution cycle of this node, the sequential code should be either

```
X1 := X; Y1 := Y;
```

or

```
Y1 := Y; X1 := X;
```

Now, consider the call

```
(A,B) = DUMMY(C,A);
```

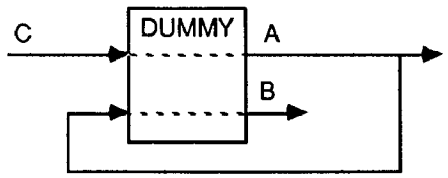

Figure 2:

corresponding to the net of Figure 2. This call is perfectly causal, but only the first version of sequential code is correct. Of course, one can exhibit a call for which only the second version would be correct.

So, we are led to expanse the node calls, and to consider only flat programs, as done for the description of the dynamic semantics.

4.2.2 Control synthesis

Clearly, boolean variables play an important role in LUSTRE: as clocks or conditions, they are often used to implement what is usually represented by control in imperative languages. Their computation must thus be carefully implemented. Let us illustrate how the rules of the dynamic semantics can be used to evaluate boolean expressions at compile time. As for the ESTEREL language, we shall build a finite automaton which is the control skeleton of the object program. Moreover, [8] describes a very efficient algorithm for this construction.

Consider the following "program", where b is an input variable:

```
c = false -> b and not pre(c);
```

Translation into basic syntax provides the program P_0:

```
c = false -> b and not pc;
pc = nil fby c;
```

Now, from the rules of dynamic semantics, we have

$$P_0 \xrightarrow{\sigma} P_1$$

where σ(c) = false, σ(pc) = nil and the program P_1 is as follows:

```
c = b and not pc;
pc = false fby c;
```

Again from the rules of dynamic semantics, we get that

- if the input b is false then

  $$P_1 \xrightarrow{\sigma} P_1$$

  where σ(c) = false, σ(pc) = false

- if the input b is true then

  $$P_1 \xrightarrow{\sigma} P_2$$

  where σ(c) = true, σ(pc) = false and the program P_2 is as follows:

  ```
  c = b and not pc;
  pc = true fby c;
  ```

Finally, we have that, whatever be the input b,

$$P_2 \xrightarrow{\sigma} P_1$$

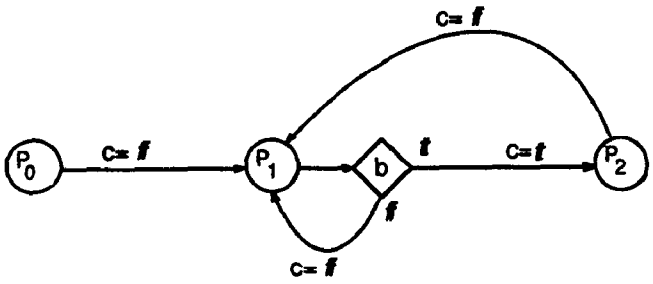

Figure 3:

where σ(c) = false and σ(pc) = true.

We have constructed the automaton of Figure 3, where the diamond stands as a test on the input.

In such a construction, complex boolean expressions (such as comparison of integers) will be considered as input. Non boolean computations are simply reported on the transitions.

At the time of the writing of this paper, the current version of the compiler does not incorporate the generation of finite state automata. Experience with ESTEREL [7] shows that, in synchronous languages, the size of the automaton generally remains small. However, if it should explode, its size can be reduced by several means. Two possibilities would be to allow the programmer to specify certain properties of input variables (e.g., that two variables can never both be true) or to restrict the simulation to some subset of the boolean variables.

In addition to code generation, such automata may be used to refine the static consistency checks:

- check of operation onto nil;
- detailed study of clock equivalence;
- refinement of the dependence relation;

Moreover, model checkers over transitions systems, such as CESAR [21] or EMC [10] can be applied on automata.

5 Program Transformation

The mathematical nature of LUSTRE allows a wide range of formal transformations, usable for program optimization and proof. In this section, we give a set of equivalence rules, which we apply on a simple example. Other transformations can be found in [12,14].

5.1 Equivalence rules

We shall focus on rules concerning operators which are specific to LUSTRE. Of course, usual axioms of boolean algebra and arithmetic will also be used.

5.1.1 Axioms of sequence operators

$$X = X\text{->}X \tag{1}$$

$$(X\text{->}Y)\text{->}Z = X\text{->}(Y\text{->}Z) = X\text{->}Z \tag{2}$$

$$X\text{->}Y = \texttt{if init then X else Y}$$
$$\quad \texttt{where init} = \texttt{true->false} \tag{3}$$

$$\texttt{pre(k)} = \texttt{nil->k}$$
$$\quad \text{for any constant k} \tag{4}$$

$$\texttt{pre(X)} = \texttt{nil->pre(X)} \tag{5}$$

$$\texttt{current(X when B)} = X'$$
$$\quad \texttt{where } X' = \texttt{if B then X else pre(X')} \tag{6}$$

$$\texttt{current(X) when B} = X$$
$$\quad \text{if B is the clock of X} \tag{7}$$

5.1.2 Distributivities

For any data operator op,

$$\texttt{pre(X op Y)} = \texttt{pre(X) op pre(Y)} \tag{8}$$

$$\texttt{(X->Y) op (Z->U)} = \texttt{(X op Z)->(Y op U)} \tag{9}$$

$$\texttt{(X op Y) when B} = \texttt{(X when B)op(Y when B)} \tag{10}$$

$$\texttt{current(X op Y)} = \texttt{current(X) op current(Y)}$$
$$\quad \text{whenever X and Y have the same clock} \tag{11}$$

5.2 Example of transformations

Let us prove that the node

```
node FOO(X) returns (Y);
var even: bool;
let
   Y = if even then current(X when not even)
               else current(X when even);
   even = true -> not pre(even);
tel;
```

behaves as the operator pre.

Proof Let us apply Rule 6 twice, we get

```
Y = if even then Y1 else Y2;
Y1 = if not even then X else pre(Y1);
Y2 = if even then X else pre(Y2);
```

Substitution of Y1 and Y2 in Y and use of the standard rule for conditional expressions yield

```
Y = if even then pre(Y1) else pre(Y2);
```

Now let us substitute even in Y:

```
Y = if (true -> not pre(even))
    then pre(Y1)
    else pre(Y2);
```

From the distributivity of -> over operators (Rules 1 and 9), and from the properties of the conditional, it becomes

```
Y = pre(Y1) -> if not pre(even)
               then pre(Y1)
               else pre(Y2);
```

Now, from the definition of Y1 and distributivity of pre (Rule 8):

```
pre(Y1) = if not pre(even)
          then pre(X)
          else pre(pre(Y1));
```

and similarly for Y2. Substitution of these expressions in Y provides

```
Y = pre(Y1) ->
       if not pre(even)
       then if not pre(even)
            then pre(X)
            else pre(pre(Y1))
       else if pre(even)
            then pre(X)
            else pre(pre(Y2));
```

Simplification of the conditional gives

```
Y = pre(Y1) -> if not pre(even)
               then pre(X)
               else pre(X);
```

A standard rule about the conditional expression yields

```
Y  = pre(Y1)->pre(X);
   = (nil->pre(Y1))->pre(X);   from Rule 5
   = nil->pre(X);              from Rule 2
   = pre(X);                   from Rule 5
```

and we are done.

Conclusion

To conclude, we shall compare LUSTRE with related works, and then consider some directions for further work.

Of course, LUSTRE can be viewed as a strict sublanguage of LUCID. In fact it applies the ideas of LUCID to a domain (continuous real-time processing) for which they are especially well-suited. However, our synchronous interpretation of sequences leads to a restrictive use of LUCID primitives, and these restrictions allow the production of efficient code (uncausality is the main problem in LUCID compilation, and the absence of an efficient compiler is the main obstacle to LUCID development).

LUSTRE must be compared with asynchronous languages with real-time capabilities, such as ADA. Our opinion is that, in asynchronous models, the notion of time cannot be given a precise and clean semantics, since these models have been designed precisely to make the behavior of a parallel system independent of the speed of its components.

Real-time processing has been the main motivation of the development of synchronous models [2,19]. These models are the basis of recent real-time languages such as ESTEREL [7], SIGNAL [17], the Statecharts of [15], and LUSTRE. The difference between LUSTRE and ESTEREL is the difference between a declarative and an imperative language; many arguments have been given in this debate, but our opinion is that the best style to use depends on the addressed problem.

The main originality of LUSTRE with respect to other data-flow languages such as VAL [18], LTS [3] or μFP [22] is the concept of clock, which allows the use of a multiform notion of time. SIGNAL presents an analog concept, but with more permissive rules of usage. Our coercive clock rules have been introduced to minimize the risk of program misunderstanding, and only practice will be able to decide which option is better.

Future work must first concern the development of the language and its compilation. An important problem is the use of arrays. They are very useful for such problems as the programming of systolic algorithms [14], but there are difficulties in finding a suitable implementation, even if we only consider arrays of fixed dimension which are not indexed by variables. A simple but inelegant solution consists of considering each array element as a simple variable. More clever implementations can be found if we consider only systolic arrays, i.e., by forbidding that an array element depend instantaneously on another element of the same array.

Other research will concern code generation for parallel architectures. In particular, it would be interesting to compare a parallel implementation based on the data-flow net with a control-flow implementation using the finite automaton.

Finally, program verification must be further studied, particularly in relation with temporal logic. LUSTRE can be viewed as a subset of some temporal logic [13], and an interesting topic is the expression of clock operators in temporal logic. Furthermore, we must study weaker equivalence relations over programs than the one considered in Section 5. As a matter of fact, especially in real-time systems, program correctness is often defined modulo some temporal approximation. It seems that clocks can be useful in defining such approximate equivalences.

Acknowledgements: *J.-L. Bergerand and Eric Pilaud participated in the design of LUSTRE. We are also indebted to Gérard Berry and his group for their help in writing the semantics of LUSTRE and in designing the compiler.*

References

[1] E. A. Ashcroft and W. W. Wadge. *LUCID, the Data-Flow Programming Language.* Academic Press, 1985.

[2] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theor. Comp. Sci.*, 30(1):91–131, 1984.

[3] S. A. Babiker, R. A. Fleming, and R. E. Milne. *A Tutorial for LTS.* Technical Report 225.84.1, Standard Telecommunication Laboratories, 1984.

[4] J-L. Bergerand. *LUSTRE: Un langage déclaratif pour le temps réel.* PhD thesis, University of Grenoble, 1986.

[5] J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real-time data-flow language. In *Real-time Systems Symposium*, pages 33–42, San Diego, 1985.

[6] J-L. Bergerand, P. Caspi, N. Halbwachs, and J. A. Plaice. Automatic control systems programming using a real-time declarative language. In *4th IFAC/IFIP Symposium on Software for Computer Control (SOCOCO)*, Graz, Austria, 1986.

[7] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar in Concurrency*, Springer-Verlag, 1985.

[8] G. Berry and R. Sethi. From regular expressions to deterministic automata. To appear, 1986.

[9] C. Buors. *Sémantique opérationnelle du langage LUSTRE.* Master's thesis, University of Grenoble, 1986.

[10] E. Clarke, E. A. Emerson, and A. P. Sistla. *Automatic verification of finite state concurrent systems using temporal logic specifications: a practical approach.* Technical Report, Carnegie-Mellon, 1983.

[11] G. Gonthier. Private communication, 1985.

[12] N. Halbwachs, A. Longchampt, and D. Pilaud. Describing and designing circuits by means of a synchronous declarative language. In *IFIP Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs*, Grenoble, 1986.

[13] N. Halbwachs and D. Pilaud. From a real-time data-flow language to a multiple time-scale temporal logic. In preparation, 1986.

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

[15] D. Harel. Statecharts: a visual approach to complex systems. In *Advanced NATO Institute on Logics and Models for Verification and Specification of Concurrent Systems*, La Colle-sur-Loup, France, 1984.

[16] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress*, 1974.

[17] P. le Guernic, A. Benveniste, P. Bournai, and T. Gautier. *SIGNAL: a data-flow oriented language for signal processing.* Technical Report 378, INRIA, 1985.

[18] J. R. McGraw. The Val language: description and analysis. *ACM Trans. on Prog. Lang. and Syst.*, 4(1):44–82, 1982.

[19] R. Milner. Calculi for synchrony and asynchrony. *Theor. Comp. Sci.*, 25(3):267–310, 1983.

[20] G. D. Plotkin. *A structural approach to operational semantics.* Technical Report DAIMI FN-19, Århus University, 1981.

[21] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium of Programming*, Springer-Verlag, 1983.

[22] M. Sheeran. muFP, a language for VLSI design. In *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 1984.