The synchronous dataflow programming language LUSTRE *

N. Halbwachs, P. Caspi, P. Raymond
IMAG/LGI - Grenoble

D. Pilaud
VERILOG - Grenoble

Abstract

This paper describes the language LUSTRE, which is a dataflow synchronous language, designed for programming reactive systems — such as automatic control and monitoring systems — as well as for describing hardware. The dataflow aspect of LUSTRE makes it very close to usual description tools in these domains (block-diagrams, networks of operators, dynamical samples-systems, etc...), and its synchronous interpretation makes it well suited for handling time in programs. Moreover, this synchronous interpretation allows it to be compiled into an efficient sequential program. Finally, the LUSTRE formalism is very similar to temporal logics. This allows the language to be used for both writing programs and expressing program properties, which results in an original program verification methodology.

1 Introduction

Reactive systems

Reactive systems have been defined as computing systems which continuously interact with a given physical environment, when this environment is unable to synchronize logically with the system (for instance it cannot wait). Response times of the system must then meet requirements induced by the environment. This class of systems has been proposed [HP85, Ber89] so as to distinguish them from *transformational* systems — i.e., classical programs whose data are available at their beginning, and which provide results when terminating — and from *interactive systems* which interact continuously with environments that possess synchronization capabilities (for instance

*This work has been partially supported by French Ministère de la Recherche within contract "Informatique 88", and by PRC C^3 (CNRS)

operating systems). Reactive systems apply mainly to automatic process control and monitoring, and signal processing, — but also to systems such as communication protocols and man-machine interfaces when required response times are very small. Generally, these systems share some important features:

- *Parallelism*: First, their design must take into account the parallel interaction between the system and its environment. Second, their implementation is quite often distributed for reasons of performance, fault tolerance, and functionality (communication protocols for instance). Moreover, it may be easier to imagine a system as comprised of parallel modules cooperating to achieve a given behavior, even if it is to be implemented in a centralized way.

- *Time constraints*: These include input frequencies and input-output response times. As said above, these constraints are induced by the environment, and should be imperatively satisfied. Therefore, these should be specified, taken into account in the design, and verified as an important item of the system's correctness.

- *Dependability*: Most of these systems are highly critical ones, and this may be their most important feature. Just think of a design error in a nuclear plant control system, and in a commercial aircraft flight control system! This domain of application requires very careful design and verification methods and it may be one of the domains where formal methods should be used with higher priority; design methods and tools that support formal methods should be chosen even if these imply certain limitations.

The synchronous approach

In our opinion, most programming tools used in designing reactive systems are not satisfactory. Clearly, assembly languages do not, though they are widely used for reasons of code efficiency. Other methods include the use of classical languages for programming sequential tasks that cooperate and synchronize using services provided by a real-time operating system, and the use of parallel languages that provide their own real-time communication services. Even the later, which seems more promising, has been criticized [Ber89] since the services being provided are low level; this does not allow programs to be easily designed and validated, while appears to be rather expensive at run time.

Synchronous languages have been recently proposed in order to deal with these problems: such languages provide "idealized" primitives allowing programmers to think of their programs as reacting *instantaneously* to external events. Thus, each internal event of a program takes place at a known time with respect to the history of external events. This feature, together with the limitation to deterministic constructs, results in deterministic programs from both functional and temporal points of view. In practice, the synchronous hypothesis amounts to assuming that the program is able to react to an external event, before any further event occurs. If it is possible to check that this hypothesis holds for given program and environment, then this ideal behavior represents a sensible abstraction. The pioneering work on ESTEREL has led to propose a general structure for the object code of synchronous programs: a finite automaton whose transition consists of executing a linear piece of code and corresponds to an elementary reaction of the program. Since the transition code has no loop, its execution time can be quite accurately evaluated on a given machine; this enables us to accurately bound the reaction time of the program, thus allowing the synchronous hypothesis to be checked.

Synchronous languages include (see this issue) ESTEREL, SIGNAL, STATECHARTS, SML, and several hardware description languages [BL85].

The dataflow approach

One method for reliable programming is to use high level languages, i.e., languages that allow a natural expression of problems as programs. Within the domain of reactive programming, many people are used with automatic control and electronic circuits; traditionally, these people model their systems by means of networks of operators transforming flows of data — gates, switches, analog devices —, and from a higher level, by means of boolean functions and transfer functions with block-diagram structures, and finally by means of systems of dynamical equations which capture the behavior of these networks. Such formalisms look quite similar to what computer scientists call "dataflow" systems [Kah74, Gra82] (cf. Figure 1). Therefore dataflow can be considered as a high level paradigm in that field. Furthermore, as a basis of a high level programming language, it possesses several advantages:

- It is a *functional model* with its subsequent mathematical cleanness, and particularly with no complex side effects. This makes it well

$X = 2 * Y + Z$

$W = X + 1$

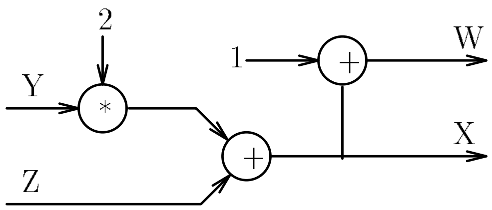

Figure 1: A dataflow description and its associated equations

adapted to formal verification and safe program transformation, since functional relations over dataflows may be seen as time invariant properties. Also, reuse is made easier, which is an interesting feature for reliable programming concerns.

- It is a *parallel model*, where any sequencing and synchronization constraints arise from data dependencies. This is a nice feature which allows the natural derivation of parallel implementations. It is also interesting to notice that, in the above domain, people were accustomed to parallelism, at much earlier times than in other areas in computer science.

Synchronous dataflow

It may thus seem appealing to develop a dataflow approach to reactive programming. However, up to now dataflow has been thought of as essentially asynchronous, whereas a synchronous approach seems necessary to tackle the problem of time, for instance by relating time with the index of data in flows.

This was the first concern of the Lustre [CPHP87] project which is reported here. It resulted in proposing primitives and structures which restrict dataflow systems to only those that can be implemented as bounded memory automata-like programs in the sense of Esterel. The language, together with programming examples, will be presented in Section 2. Then compiling and efficient code generation matters will be discussed in Section 3.

The second main concern of the project is to take advantage of the approach in developing techniques of formal verification (Section 4). The idea is to consider Lustre as a specification language as well, thanks to its declarative aspect. It is then shown that the same compiler can be used as a tool for verifying program correctness with respect to such specifications.

Section 5 presents several other current activities of the project, related to hardware and distributed implementations. Finally comparisons with existing approaches are discussed.

2 The LUSTRE language

2.1 Flows and Clocks

In LUSTRE, any variable and expression denotes a *flow*, i.e., a pair made of

- a possibly infinite sequence of values of a given type,
- a *clock*, representing a sequence of times.

A flow takes the n-th value of its sequence of values at the n-th time of its clock. Any program, or piece of program has a cyclic behavior, and that cycle defines a sequence of times which is called the *basic clock* of the program: a flow whose clock is the basic clock takes its n-th value at the n-th execution cycle of the program. Other, slower, clocks can be defined, thanks to boolean-valued flows: the clock defined by a boolean flow is the sequence of times at which the flow takes the value *true*. For instance table 1 displays the time-scales defined by a flow C whose clock is the basic clock, and by a flow C′ whose clock is defined by C.

basic time-scale	1	2	3	4	5	6	7	8
C	*true*	*false*	*true*	*true*	*false*	*true*	*false*	*true*
C time-scale	1		2	3		4		5
C′	*false*		*true*	*false*		*true*		*true*
C′ time-scale			1			2		3

Table 1: Boolean flows and clocks

It should be noticed that the clock concept is not necessarily bound to physical time. As a matter of fact, the basic clock should be considered as setting the minimal "grain" of time within which a program cannot discriminate external events, and which corresponds to its response time. If "real time" is required, it can be implemented as an input boolean flow: for instance a flow whose *true* value indicates the occurrence of a "millisecond" signal. This point of view provides a multiform concept of time: "millisecond" becomes a time-scale of the program among others.

2.2 Variables, Equations, Expressions, Assertions

Variables should be declared with their types, and variables which do not correspond to inputs should be given one and only one definition, in the form of equations. These are considered in a mathematical sense: the equation "`X = E;`" defines variable `X` as being identical to expression `E`. Both have the same sequence of values and clock. However such an equation is oriented in the sense that it defines `X`. The way it is used in other equations cannot give it more properties than those which arise from its definition. This provides one important principle of the language, *the substitution principle*: `X` can be substituted to `E` anywhere in the program and conversely. As a consequence, equations can be written in any order, and extra variables can be created so as to give names to subexpressions, without changing the meaning of the program.

LUSTRE has only few elementary basic types: boolean, integer, real, and one type constructor: *tuple*. However, complex types can be imported from a host language and handled as abstract types (A similar mechanism exists in ESTEREL).

Constants are those of the basic types and those imported from the host language (for instance constants of imported types). Corresponding flows have constant sequences of values and their clock is the basic one.

Usual operators over basic types are available (arithmetic: `+, -, *, /, div, mod` ; boolean: `and, or, not` ; relational: `=, <, <=, >, >=` ; conditional: `if then else`) and functions can be imported from the host language. These are called *data operators* and only operate on operands sharing the same clock; they operate pointwise on the sequences of values of their operands. For instance, if `X` and `Y` are on the basic clock, and their sequences of values are respectively $(x_1, x_2, \ldots, x_n, \ldots)$ and $(y_1, y_2, \ldots, y_n, \ldots)$, the expression

```
if X>0 then Y+1 else 0
```

is a flow on the basic clock whose n-th value for any integer n is:

if $x_n > 0$ then $y_n + 1$ else 0

Besides these operators, LUSTRE has four more which are called "temporal" operators, and which operate specifically on flows:

- `pre` ("previous") acts as a memory: if $(e_1, e_2, \ldots, e_n, \ldots)$ is the sequence of values of expression `E`, `pre(E)` has the same clock as `E`, and its sequence of values is $(nil, e_1, e_2, \ldots, e_{n-1}, \ldots)$, where nil represents an undefined value denoting an uninitialized memory.

- `->` ("followed by"): if `E` and `F` are expressions with the same clock, with respective sequences $(e_1, e_2, \ldots, e_n, \ldots)$ and $(f_1, f_2, \ldots, f_n, \ldots)$, then `E->F` is an expression with the same clock as `E` and `F`, and whose sequence is $(e_1, f_2, f_3 \ldots, f_n, \ldots)$. In other words, `E->F` is always equal to `F`, but at the first time of its clock.

Table 2 shows the effect of the last two operators:

- `when` "samples" an expression according to a slower clock: if `E` is an expression and `B` is a boolean expression with the same clock, then `E when B` is an expression whose clock is defined by `B`, and whose sequence is extracted from the one of `E` by keeping only those values of indexes corresponding to *true* values in the sequence of `B`. In other words, it is the sequence of values of `E` when `B` is *true*.

- `current` "interpolates" an expression on the clock immediately faster than its own. Let `E` be an expression whose clock is not the basic one, and let `B` be the boolean expression defining this clock. Then `current E` has the same clock `C` that `B` has, and its value at any time of this clock `C`, is the value of `E` at the last time when `B` was *true*.

`B`	*false*	*true*	*false*	*true*	*false*	*false*	*true*	*true*
`X`	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
`Y = X when B`		x_2		x_4			x_7	x_8
`Z = current Y`	nil	x_2	x_2	x_4	x_4	x_4	x_7	x_8

Table 2: Sampling and interpolating

Besides being made of equations, the body of a Lustre program may contain *assertions*. These generalize equations and consist of boolean expressions that should be always true. Their primary use is to give to the compiler indications in order to optimize the code when the environment of the program possesses some known properties (see § 3.4). For instance, if we know that two input events represented by boolean variables `x` and `y` never occur at the same time, we shall write:

```
assert not(x and y);
```

Similarly, the assertion

```
assert (true -> not(x and pre(x)));
```

says that event `x` never occurs twice in a row. Note the initialization to `true`, which prevents the occurrence of value *nil*, which is forbidden in assertions, clocks, and output sequences (cf. §3.1). Besides their use in code optimization, assertions play a important role in program verification (cf. §4).

2.3 Program structure

A LUSTRE system of equations can be represented graphically as a network of operators. For instance, the equation

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

which defines a counter of basic clock cycles, corresponds to the network of figure 2. This naturally suggests some notion of subroutine: a subnetwork can be encapsulated as a new reusable operator which is called a *node*. A node declaration consists of an interface specification — providing input and output parameters with their types and possibly their clocks — optional internal variables declarations, and a body made of equations and assertions defining outputs and internal variables as a function of inputs.

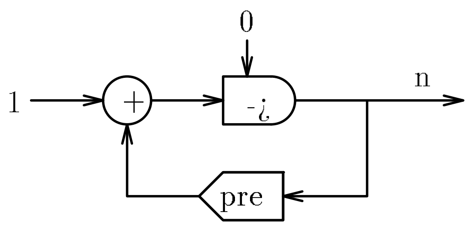

Figure 2: Counter network

For instance, the following node defines a general purpose counter, having as inputs an initial-and-reset value, an increment value, and a reset event:

```
node COUNTER(val_init, val_incr:  int; reset:  bool) returns (n:  int);
let
   n = val_init -> if reset then val_init else pre(n) + val_incr;
tel.
```

Such a node can be functionally instancied in any expression. For instance

```
even = COUNTER(0,2,false);
modulo5 = COUNTER(0,1,pre(modulo5)=4);
```

define the sequence of even numbers and the cyclic sequence of modulo 5 numbers, over the basic clock.

Similarly, if `gamma` is an acceleration expressed in $meter/second^2$, and its clock's rate is $onepersecond$, one could have

```
speed = COUNTER(0,gamma,false);
position = COUNTER(0,speed,false);
```

According to the substitution principle, this is equivalent to:

```
position = COUNTER(0,COUNTER(0,gamma,false),false);
```

A node may have several outputs; in that case, the output is a tuple. For instance

```
node D_INTEGRATOR(gamma: int) returns(speed,position:int);
let
    speed = COUNTER(0,gamma,false);
    position = COUNTER(0,speed,false);
tel.
```

is instancied as

```
(v,x) = D_INTEGRATOR(g);
```

Concerning clocks, the basic clock of a node is defined by its inputs, so as to be consistent with the dataflow point of view. For instance, expression:

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

counts only when `B` is *true*. In the example, operator `when` applies to the tuple `(0,1,false)`[1]. Table 3 shows the result of the expression, and the difference with expression `(COUNTER(0,1,false)) when B` , where sampling applies to the output of the node instead of its inputs.

B	*true*	*false*	*true*	*false*	*true*
`(0,1,false) when B`	(0,1,*false*)		(0,1,*false*)		(0,1,*false*)
`COUNTER((0,1,false) when B)`	0		1		2
`COUNTER(0,1,false)`	0	1	2	3	4
`(COUNTER(0,1,false)) when B`	0		2		4

Table 3: Nodes and clocks

This example also stresses the interest of clocks in reuse; had clocks not been available, the only way of getting the same effect would have required to modify the node by adding a "do-nothing" input.

A node may admit input parameters with distinct clocks. Then the faster one is the basic clock of the node, and all other clocks must be in the input declaration list. In the following example:

```
node N (millisecond:bool; (x:int ; y:bool) when millisecond) returns ...
```

the basic clock of the node is the one of `millisecond`, and the clock of `x` and `y` is the one defined by `millisecond`.

Outputs of a node may have clocks different from its basic clock. Then these clocks should be visible from the outside of the node. Note also that these clocks are certainly slower than the basic one.

2.4 Some programming examples

Linear systems

Translating sampled linear systems into LUSTRE programs is quite an obvious task: if systems are expressed in z-transform equations, it amounts to translating the z^{-1} operator into `0.0 -> pre()`. For instance, consider the 2nd order filter:

$$H(z) = \frac{az^2 + bz + c}{z^2 + dz + e}$$

[1]This is equivalent to `COUNTER(0 when B, 1 when B, false when B)`

The output $y = H(z)x$ can be written:

$$y = ax + (bx - dy)z^{-1} + (cx - ey)z^{-2}$$

and yields the following program:

```
const a,b,c,d,e: real.

node SECOND_ORDER(x: real) returns (y: real);
var u,v: real;
let
   y = a*x + (0.->pre(u));
   u = b*x - d*y + (0.->pre(v));
   v = c*x -e*y;
tel.
```

Furthermore, clocks allow an easy extension to multiply sampled systems.

Non-linear and time-varying systems

Letting identifiers `a,b,c,d,e` be parameters of the `SECOND_ORDER` node, instead of constants, yields a time-varying filter. Non-linear systems are also easy to describe. For instance:

```
y = rho*cos(theta0 -> pre(theta));
```

Logical systems

From the previous discussion, dataflow programs of signal processing systems are very close to their specification in terms of systems of dynamical equations. However many systems have an important logical component, and some of them, for instance monitoring systems, are essentially logical systems. Such systems are most often described in terms of automata, parallel automata (STATECHARTS for instance), and Petri nets, i.e., imperative formalisms which describe states and transitions between states. The question about the adequacy of dataflow paradigms to provide easy descriptions of such systems should therefore be carefully checked. The following examples are intended to show that these paradigms may allow easy, incremental and modular descriptions of logical systems.

In this subsection we shall consider three versions of a “watchdog”, i.e., a device that monitors response times. The first version receives three events:

set and reset commands, and deadline occurrence. The output is an alarm that must be raised whenever a deadline occurs and the last received command was a set.

As usual, events are represented by boolean variables whose value *true* denotes the presence of an event. The watchdog will be a LUSTRE node having three boolean inputs set, reset and deadline and emitting a boolean output alarm. As the order of equations is unimportant, we begin by defining the output: alarm is true when deadline is true and the last true command is set. Let is_set be a local boolean variable expressing the latter condition. Then, we can write:

```
alarm = deadline and is_set;
```

It remains to define is_set, which becomes *true* any time set is *true*, and *false* any time reset is *true*. Initially, it is *true* if set is *true* and *false* otherwise:

```
is_set = set -> if set then true else if reset then false else pre(is_set);
```

We can furthermore assume that set and reset commands never take place at the same time, which can be expressed by an assertion. The full program is:

```
node WD1 (set, reset, deadline: bool) returns (alarm: bool);
var is_set: bool;
let
   alarm = deadline and is_set;
   is_set = set -> if set then true
                   else if reset then false else pre(is_set);
   assert not(set and reset);
tel.
```

Let us consider now a second version which receives the same commands, but raises the alarm when no reset has occurred for a given time since the last set, this time being given as a number of basic clock cycles. This new program reuses node WD1, by providing it with an appropriate deadline parameter: on reception of a set event, a register is initialized, which is then decremented. Deadline occurs when the register value reaches zero; it is built from a general purpose node EDGE which returns *true* at each rising edge of its input:

```
node EDGE (b: bool) returns (edge: bool);
let
   edge = false -> (b and not pre(b));
tel.

node WD2 (set, reset: bool; delay: int) returns (alarm: bool);
var remain: int; deadline: bool;
let
   alarm = WD1(set, reset, deadline);
   deadline = false -> EDGE(remain = 0);
   remain = if set then delay
                   else if pre(remain)>0 then pre(remain)-1
                   else pre(remain);
tel.
```

Assume now that the delay is expressed according to a given time-scale, i.e. as a number of occurrences of an event `time_unit`. We just have to call `WD2` with an appropriate clock: `WD2` must catch any time units `time_unit`, any commands, and must be properly initialized so that alarm never yields *nil*:

```
node WD3 (set, reset, time_unit: bool; delay: int) returns (alarm: bool);
var clock: bool;
let
   alarm = current(WD2((set,reset,delay) when clock));
   clock = true -> (set or reset or time_unit);
tel.
```

Coming back to the question raised at the beginning of the section, we can see that programs have been written without referring to transitions between states, but rather by describing states in terms of state variables, and by stating the strongest invariant property of each state variable. Then, all state variables will evolve in parallel, thus recreating the global state of the system. It has been shown in [BFH90b] that any finite state machine can be described by a boolean LUSTRE program.

Mixed logical and signal processing systems

Finally, mixing signal processing and logical systems is quite an easy task: Signal processing parts provide logical ones with boolean expressions by using relational operators, and conversely, logical components control signal flows by means of conditional operators: `if then else`, `when` and `current`.

3 The Lustre compiler

Let us describe now the main techniques used in the LUSTRE-V2 compiler [Pla88]. This prototype compiler has been written in Le-Lisp by John Plaice.

3.1 Static verifications

Static well-formedness checking is clearly an important issue within the framework of reliable programming, and aims at avoiding the overhead of dynamic checks at run time. Besides classical type checking, the main checks performed by the compiler are:

- Definition checking: any local and output variable should have one and only one equational definition.
- Absence of recursive node call: in view of obtaining automata-like executable programs, LUSTRE allows up to now only static networks to be described. The problem of structuring recursive calls so that the above property is maintained, has not yet been investigated.
- Clock consistency, which will be more intensively discussed below.
- Absence of uninitialized expressions (yielding *nil* values). Such expressions are accepted as far as these do not concern clocks, outputs, and assertions.
- Absence of cyclic definitions: any cycle in the network should contain at least one `pre` operator. In the sense of [Kah74] an equation such that: `X = 3*X + 1` has a meaning which is the least solution with respect to the prefix ordering of sequences; in this case, the solution for `X` is the empty sequence, and it can be interpreted as a deadlock. It is therefore rejected. Note also that LUSTRE also rejects structural deadlocks which are not true ones, such that:

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

  The reason is that the analysis of such networks is undecidable, in general .

Let us discuss now the **clock calculus** which represents an original aspect of LUSTRE with respect to dataflow languages. The following program illustrates the reason for such a calculus:

```
b = true -> not pre b;
y = x + (x when b);
```

In the second equation, a data operator combines two flows of distinct clocks. According to standard dataflow philosophy, such a program has a meaning. However, it is easy to see that the computation of the $2n^{th}$ value of y needs both the $2n^{th}$ and the n^{th} values of x. Since a reactive system may be assumed to run for ever, its required memory will certainly overflow. Such a program could not be compiled into a bounded memory object code, not to speak of the physical incoherency consisting of adding something at time n with something at time $2n$.

The clock calculus consists of associating a clock with each expression of the program, and of checking that any operator applies to appropriately clocked operands:

- any primitive operator with more than one argument applies to operands sharing the "same" clock;
- the clock of any operand of a `current` operator is not the basic clock of the node it belongs to[2];
- the clocks of a node operands should obey the clocks requirements stated in the node definition header.

Let us define here what we mean by "the same clock". Ideally, it could mean the same boolean flow, but this may require semantical analysis which are undecidable in general. Thus the compiler uses a more restricted notion of equality: two boolean expressions define the same clock if and only if these can be unified by means of syntactical substitutions. Consider the example:

```
x = a when (y>z);
y = b+c;
u = d when (b+c>z);
v = e when (z<y);
```

[2] In contrast with Signal, Lustre does not allow basic clock time intervals to be split into smaller ones.

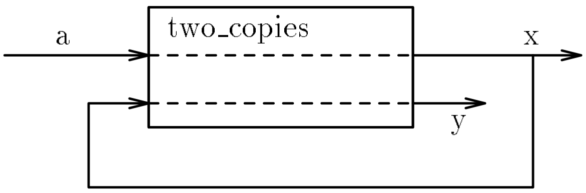

Figure 3: A cyclic call

`x` and `u` share the same clock, which is considered to be distinct from the clock of `v`.

The rules of the clock calculus are formally described in [CPHP87, Pla88].

3.2 Node expansion

The LUSTRE compiler produces purely sequential code. This raises the question of compiling separatedly nodes which are used in other nodes. The following example shows this cannot be easily done for LUSTRE:

```
node two_copies(a, b: int) returns (x, y: int);
let x = a ; y = b ; end.
```

Clearly, there are two possible sequential codes for a basic cycle of this node, either `x:=a;y:=b;` or `y:=b;x:=a;`

But the choice between those two programs may depend on the way the node is used within another node; for instance:

```
(x,y) = two_copies(a,x);
```

corresponding to figure 3. In this case, only the former program is correct.

Thus, before compiling a program, the compiler first expands recursively all the nodes called by that program, i.e., formal parameters are substituted with actual ones, local variables are given an unique name (so as to distinguish that node call from other instances of the same node) and then the called node body is inserted into the calling node body. The code generation step will then start from a "flat" node which does not call any other node[3].

[3] However, we shall see in § 5.1 that some separate compiling technique can also apply.

3.3 Single-loop code

An obvious way of associating an imperative program with a LUSTRE node consists of constructing an infinite loop whose body implements the inputs to outputs transformation performed at any basic cycle of the node. This is done by:

- choosing variables to be computed (the output ones and the least possible number of local ones, which implement either memories or temporary buffers),
- defining the actions which update these variables,
- and choosing an ordering of these actions, according to the dependencies between variables induced by the network structure of the node.

As an example, let us consider a modified version of the watchdog WD3:

```
node WD4 (set,reset,u_tps:bool; delay: int) returns (alarm:bool);
var is_set: bool; remain:int;
let
   alarm = is_set and (remain = 0) and pre(remain)>0;
   is_set = false -> if set then true
                     else if reset then false
                     else pre(is_set);
   remain =  0 -> if set then delay
                      else if u_tps and pre(remain)>0
                           then pre(remain)-1
                      else pre(remain);
   assert not(set and reset);
tel.
```

The single-loop body, which is executed at each program reaction, looks like:

```
   if _init then % first cycle %
      is_set := false; remain := 0; alarm := false; _init := false
   else % other cycles %
      if set then is_set:= true; remain:= delay
      else
         if reset then is_set:= false endif;
         if u_tps and (_pre_remain>0) then remain := _pre_remain-1 endif;
      endif
```

```
    alarm := is_set and (remain=0) and (_pre_remain>0);
 endif
 write(alarm); _pre_remain := remain;
```

Remarks

- The compiler has defined auxiliary variables: the variable `_init` — which is assumed to be initialized to *true* and is used to implement the operator `->` — and the memory variable `_pre_remain`. Note that the expression `pre(is_set)` did not result in the creation of a memory variable since the compiler found a way to avoid it.

- Although it is easy to find an ordering of actions which meets the dependency relations between variables (static checks described above ensure that such an order exists), the choice of a "good" order is quite difficult: particularly, the order according to which conditional statements are opened and closed is critical with respect to code length.

- The code speed could be improved. Note for instance that at any cycle the program tests whether this is the first one or not, and this is particularly awkward. A solution consists of using more complex control structures than the single-loop structure. This is discussed in the following section.

3.4 Automaton-like code

The search for more complex control structures is borrowed from the compiling technique of Esterel and is based on the following remarks:

- The classical concept of control of imperative programs is represented in Lustre by means of boolean variables acting over conditional and clock handling operators.

- If a condition or a clock depends on values of a boolean variable computed at previous cycles (by means of an expression like `pre(B)` or `current(B)`) the code of the actual cycle could be made simpler if that value could be assumed to be known. One could then distinguish the code to be executed according to that value.

The synthesis of the control structure consists of choosing a set of state variables of boolean type, whose values are expected to influence the code of future cycles. This set of variables is called the state of the program and it takes only a finite set of values. For each possible value of the state, one defines the sequential code which would be executed during a cycle if the state of the variables had the above values just before the execution of the cycle. Hence, starting from a given state and executing the corresponding code would result in computing the next state, and be ready for the execution of the next cycle. Finally, a static reachability analysis can be performed so as to delete state values and transitions which cannot be reached from the initial state (As a matter of fact, this reachability analysis is done while generating state values and transitions, so as to avoid generating useless items). The result is a finite state automaton, whose transitions are labeled with the code of the corresponding reaction.

State variables can be chosen in several ways among the following:

- boolean expressions resulting from `pre` and `current` operators,
- auxiliary variables like `_init_C`, associated with some clock `C` whose value is *true* at the first clock cycle and then *false*, and which allow the evaluation of `->` operators.

This control synthesis is illustrated on the watchdog example `WD4` (cf. § 3.3):

The chosen state variables are `pre(is_set)` and `_init`. Then:

1. The first cycle yields `pre(is_set)=nil` and `_init=true`. Let S_0 be this initial state. Since `_init=true` in this state, the value of all `->` operators is the one of their first operand. Thus, `is_set=false`, and `remain=0`. Elementary boolean calculus yields `alarm=false`. Furthermore, since `is_set` evaluates to *false*, this will be the value of `pre(is_set)` at the next state. The next state, S_1, is then `pre(is_set)=false` and `_init=false`. State S_0 code looks like:

```
S0 : remain := 0;
     alarm := false;
     pre_remain := remain;
     goto S1;
```

2. In state S_1, since `pre(is_set)` value is *false*, `is_set` evaluates to *true* if and only if the input `set` value is *true*. Let S_2 be the state where `pre(is_set)` is *true* and `_init` is *false*. The code for state S_1 is:

```
S1 : if set then
        remain := delay;
        alarm := (remain = 0) and (pre_remain > 0);
        pre_remain := remain;
        goto S2;
     else
        remain := if u_tps and pre_remain > 0 then pre_remain-1
                  else pre_remain;
        alarm := false;
        pre_remain := remain;
        goto S1;
     endif
```

3. The code of state S_2 (`pre(is_set)` is *true* and `_init` is *false*), is as follows:

```
S2 : if set then
        remain := delay;
        alarm := (remain = 0) and (pre_remain > 0);
        pre_remain := remain;
        goto S2;
     else
        if reset then
           remain := if u_tps and pre_remain > 0 then pre_remain-1
                     else pre_remain;
           alarm := false;
           pre_remain := remain;
           goto S1;
        else
           remain := if u_tps and pre_remain > 0 then pre_remain-1
                     else pre_remain;
           alarm := (remain = 0) and (pre_remain > 0);
           pre_remain := remain;
           goto S2;
        endif
     endif
```

All reachable states being processed, this ends the code generation. Figure 4 display the resulting automaton.

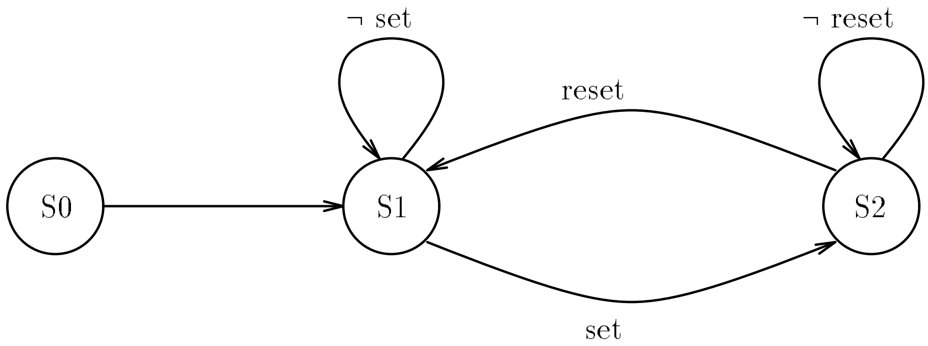

Figure 4: The watchdog control automaton

Remarks

- The obtained transition codes are much simpler than the single-loop code, particularly for S_0 and S_1 codes. This reduction may be even more impressive for larger programs.

- In contrast, the overall length of the code may become very large. That is why, in practice, an action code table is built which uniquely identifies actions that may belong to several transitions, and transition codes refer to actions by means of their indexes in the table.

- Boolean expressions depending on non boolean variables, which are needed for computing state variables (integer comparison for instance) are handled as inputs by means of tests on their value.

- This technique allows assertions to be fully taken into account. Assertions are computed in the same way as state variables, and any branch yielding a false assertion is deleted. A state whose total code has been deleted is then declared unreachable, and branches already computed which lead to that state are recursively deleted. It should be noticed that assertions may increase the number of state variables and reachable states, as well as increase code length by inducing extra tests.

- In contrast with Esterel automata, the obtained Lustre automata are often far from being minimal (this question will be further discussed at § 5.1). This entails a need for minimization.

3.5 The Esterel/Lustre environment

Automata produced by the Lustre compiler are expressed in the oc format [PS87], which is also used by the Esterel compiler. Several common tools take this format as input:

Code generators: Translators towards C, Le-Lisp and ADA languages have been designed by the Esterel team. They produce the procedure which implements the code corresponding to a transition of the automaton.

Automaton minimizer: The Aldebaran [Fer88] minimizer has been interfaced with oc. It allows minimal equivalent automata to be obtained in oc, and this is particularly useful in the case of Lustre.

Interfaces with proof tools: Automata are a common basic model in many analysis and verification tools for parallel systems. It was therefore appealing to experiment with the use of such tools operating on oc automata. Thus, oc has been interfaced with Auto [Ver86]. Some experiments have also been performed with Emc [CES86] and Xesar [RRSV87]. However, we shall see in Section 4 other proof techniques which apply specifically to Lustre.

Display tools: The oc language has been designed for internal code representation, and it thus lacks of readability. For checks and debugging purposes, translators towards readable representations, and graphic display based on the Autograph [RS89] code, have been developed.

4 Verification

As noted in the introduction, reactive systems often concern critical applications, and thus program verification is a key issue. However, many practitioners in the field are skeptical with the use of formal verification methods, and convincing arguments need to be provided in order to support our claim that indeed, such methods are of practical interest. This is the object of the following discussion.

The research on program verification which started in the early seventies intended to provide complete proofs of very general programs. Though this work has led to important contributions concerning programming techniques and language design, one should admit that its use in practice is very limited. However, our goal concerning reactive systems may be less ambitious. Almost always, the safety of a critical application does not depend on the total correctness of its control program, but rather on an often small set of properties that the program should fulfill. For instance, the occurrence of a critical situation should raise an alarm within a given delay. From our experience, the proof of such properties can often be handled within the framework of simple decidable theories, as these properties seldom depend on numerical relations and computations. Furthermore, most of these properties are "safety" properties which state that a given situation should never appear, or that a given statement should always hold, in contrast with "liveness" properties which state that a given situation should eventually appear in the future. For instance, a relevant question is not that a train will eventually stop, but that it never crosses a red light. This is an important remark as proof techniques for safety properties are known to be much simpler than for liveness properties:

- A safety property can be verified by simply checking properties of reachable states, without taking into account the transition relation (it is used only for constructing the reachable states). This allows the use of very efficient methods based on reachability [Hol87, CVWY90].

- A safety property can be checked on an abstraction of the actual program. Informally, if a safety property holds for a program, it also holds for programs whose set of behaviors is a subset of the initial one. Thus it is possible to abstract programs by ignoring details, for instance numerical computations; their set of behaviors will become larger and properties that hold on these abstractions will also hold on the actual programs.

- Safety properties can be checked modularly. Properties of submodules can be combined so as to derive a property of the whole module. This allows proof complexities to be reduced, thanks to modular decomposition according to a program structure.

In view of this discussion, we will propose methods for specifying and checking simple safety properties about LUSTRE programs.

4.1 Specification of safety properties

Many formalisms have been proposed in order to express properties of real time parallel programs. Two main approaches can be distinguished: those based on temporal logics [Pnu77, MM84], and those based on automata theory (Petri nets, STATECHARTS, timed graphs [ACD90] and process calculi [Mil83]). Such formalisms should clearly allow any interesting property to be expressed, but should also provide an easy and readable expression for it; proving a certain property is of poor interest if one cannot be convinced that it is actually the desired property of the system.

This led us to investigate if it were possible to take advantage of LUSTRE's declarative aspect, so as to use it for expressing properties of LUSTRE programs [HPOG89]. A positive answer is based on the following considerations:

- LUSTRE can be considered as a subset of a temporal logic [PH88, BFH90b]. Our proposal is then to express any temporal property P by a boolean expression `B`, such that P holds if and only if expression `B` is always true during any execution path of the program. According to [BFH90b], any safety property can be expressed in such a way.

- The above proposal is easily implementable by using the assertion mechanism of LUSTRE: LUSTRE assertions are already a means of expressing properties of a program's environment.

- The use of a programming language for expressing both programs and their properties is interesting since all the structuring facilities of the language become available for the sake of readability and expressiveness. For instance, as we will show, the node concept will allow the user to define its own temporal operators.

Let us show here how some useful non trivial temporal operators can be expressed as LUSTRE nodes. Consider the following property:

> "any occurrence of a critical situation must be followed by an alarm within a five seconds delay"

Such a property relates three events: the critical situation occurrence, the alarm, and the deadline. The latter can be provided externally as well as it can easily be expressed in LUSTRE. A general pattern for this property is the following one:

"Any occurrence of event A is followed by an occurrence of event B before the next occurrence of event C"

However, this formulation is not directly translatable into LUSTRE as it refers to what happens in the future following an A occurrence, while LUSTRE only allows references to the past with respect to the current instant. That is why we first translate it into the equivalent past expression:

"Any time C occurs, either A has never occurred previously, or B has occurred since the last occurrence of A."

Let us define a node, taking three boolean input parameters `A, B, C`, and returning a boolean output `X` such that such that `X` is always true if and only if the property holds:

```
node onceBfromAtoC(A,B,C: bool) returns (X: bool);
let
   X = implies(C, never(A) or since(B,A));
tel
```

The equation defining `X` uses three auxiliary nodes:

- The nodes implies implements the ordinary logical implication:

  ```
  node implies(A, B: bool) returns (AimpliesB: bool);
  let AimpliesB = not A or B; tel.
  ```

- The node `never` returns the value *true* as long as its input has never been equal to *true*. Then it returns *false* for ever:

  ```
  node never(B: bool) returns (neverB: bool);
  let
     neverB = (not B) -> (not B and pre(neverB));
  tel.
  ```

- Finally, the node `since` has two inputs and it returns *true* if and only if, either its second input has still not been *true*, or its first input has been *true* at least once since the last *true* value of the second input:

```
node since(X,Y: bool) returns (XsinceY: bool);
let
   XsinceY = if Y then X else (true -> X or pre(XsinceY));
tel.
```

A realistic example has been studied in [Glo89]: Most critical properties of a nuclear plant monitoring program have been expressed in LUSTRE, thanks to a small set of general purpose temporal operators similar to `onceBfromAtoC`, `never` or `since`.

4.2 Verification

The proposed verification method is very similar to "model checking" [CES86, RRSV87]: first, the state graph of the program is built (this assumes obviously a finite number of states), and then each property is checked on this state graph. The critical issue in this approach is clearly the number of states which can be very large for realistic programs. We shall see that the restriction to safety properties, and the expression of properties in the same language as the program may help in solving this problem.

In the LUSTRE case, a state graph already exists corresponding to the control automaton built by the compiler. This graph is an abstraction of the actual state graph since it expresses only the control and ignores many details concerning non boolean variables, and boolean ones which do not influence that control. As noticed above, if properties to be checked depend essentially on booleans taken into account in the control graph, and if these properties are safety ones, such an abstraction is a sensible one for checking purposes and yields in general much smaller graphs.

An important observation for decreasing the total graph size consists of taking into account the property to be checked when building the state graph. In the case of LUSTRE this is easily achieved since the same language applies to properties and programs: in order to prove that an expression `B` is an invariant of the program P, we build a new program P' made of the body of P and of the system of equations defining `B`, and whose only output is `B` (cf. Figure 5). Since the compiler is then requested to only compute `B`, it will only take into account the part of the program which concerns that computation, and this can be expected to yield a smaller graph. Given that graph, verifying the property corresponds to check that in none of the states, the code performs an assignment of the output to *false*.

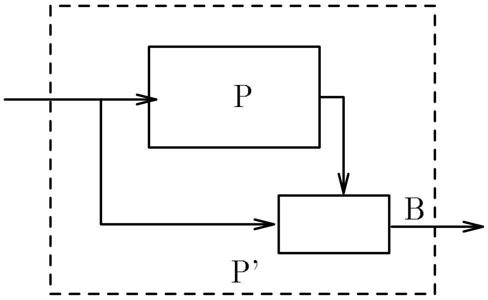

Figure 5: Verification program

A third issue in reducing the size of the graph consists of using assertions for expressing assumptions when the property to be checked is suspected to hold only on these assumptions. Assertions are also useful for expressing properties of numbers which otherwise would be ignored by the compiler. For instance, if a program uses numerical tests such as `X<=Z` and `Y<=Z`, the assertion:

```
assert not(X<=Y and Y<=Z and not X<=Z);
```

prevents the compiler from generating states satisfying $Z<X\leq Y\leq Z$, which of course would not be reachable by the actual program.

As an example, let us consider the following general purpose node[4], which represents a switch: its output alternates from *true* to *false* according to input events `ON` and `OFF`; a third input defines its initial value. A first version of this node could be:

```
node SWITCH_1(ON, OFF, INIT: bool) returns (STATE: bool);
let
   STATE = INIT -> if ON then true
                   else if OFF then false
                   else pre(STATE);
tel.
```

However, this version has a flaw: in the call

[4]Such a node could have been used in defining the variable `is_set` in the CG1 (cf. § 2.4) version of watch-dogs.

```
 state = SWITCH_1(button, button, init)
```

the output does not change each time the button is pushed, as we might expect. Thus a more general version should take into account the previous STATE when checking the inputs ON and OFF:

```
node SWITCH(ON, OFF, INIT: bool) returns (STATE: bool);
let
   STATE = INIT -> if ON and not pre(STATE) then true
                   else if OFF and pre(STATE) then false
                   else pre(STATE);
tel.
```

We could wish to verify that this generalization is correct, in the sense that both versions behave in the same way as soon as the inputs ON and OFF are never true at the same time. This is achieved by constructing a comparison node which calls both nodes with same inputs and compares their outputs, under the assumption that ON and OFF inputs are exclusive (cf. Fig. 6):

```
node COMPARE(ON, OFF, INIT: bool) returns (OK: bool);
     var state, state_1 : bool;
let
   state = SWITCH(ON, OFF, INIT);
   state_1 = SWITCH_1(ON, OFF, INIT);
   OK = (state = state_1);
   assert not(ON and OFF);
tel.
```

Compiling this node yields a five states automaton, each transition of which assigns the value *true* to the output OK.

The last way to tackle the state explosion problem is *modular verification*. Having to prove that an expression B is always true during the execution of a program P, calling a node Q (cf. Fig. 7.a), the idea is to decompose the proof into a sub-proof concerning Q, and a sub-proof concerning P without Q:

- Find (by intuition) a property of Q, i.e., an expression C on the input/output parameters of Q, and prove that C is always true during any execution of Q.

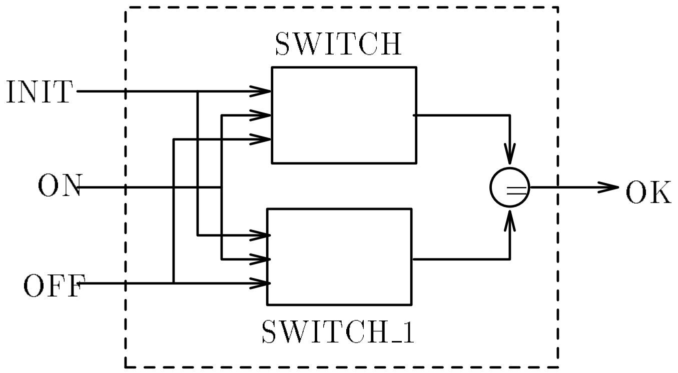

Figure 6: Assumption-dependent equivalence of programs

Figure 7: Modular verification

- Now, consider `Q` as being part of the environment of `P`, i.e., replace in `P` the call to `Q` by the assertion `assert C`. Then try to prove the invariance of `B` on the modified program (cf. Fig. 7.b).

An example making use of this modular decomposition may be found in [HLR92].

A prototype verification tool called LESAR (by analogy with the CESAR family of model checkers) has been implemented: given a program with a single boolean output, it goes through the states and checks that the output is never assigned *false*. It has been used to check the above mentioned nuclear plant control system [Glo89]. Though this program used computations on real numbers, the state graphs it needed to build appeared to be quite small (up to 1000 states).

Of course, the validity of the proof relies on the satisfaction of the synchrony hypothesis: All the proof is performed "inside" the synchronous model, and has nothing to do with performance analysis. As mentioned before, checking the validity of the synchrony hypothesis amounts to evaluate the maximum reaction time of the program on a given machine.

5 Current activities

5.1 The next compiler version

In section 3, the LUSTRE-V2 compiler currently available was described. However, from experiments conducted with this version, some serious drawbacks have been identified, and an improved version is currently being designed. We briefly discuss here the main trends adopted in this new design.

Automata minimization: As indicated above, automata provided by the current compiler are far from being minimal, while this is not the case with ESTEREL generated automata. The suspected reason for this may be the following one: ESTEREL is an imperative language offering powerful control structures (sequencing, interruptions, ...). Furthermore, it is a medium to large grain parallel language in the sense that its parallel construct is an explicit one, and its use may be tightly controlled by a programmer[5]. This allows "good programming" rules to be stated which lead to minimal automata. On the contrary, control in LUSTRE is hidden as it results from data dependencies, and LUSTRE is a fine grain parallel language in the sense that any expression is a potentially parallel construct. Thus minor changes in a program text may induce large variations in the automaton size, and though some causes of state explosion have been identified, these cannot be easily synthesized as sensible programming rules. The problem of efficiently compiling LUSTRE is therefore intrinsically difficult. Several solutions are currently investigated:

- *A posteriori minimization:* The use of an automaton minimizer such that ALDEBARAN (cf. § 3.5) which has already been interfaced so as to

[5] Though the main ESTEREL assumption is that the synchronous product of automata limits state explosion with respect to an asynchronous product, it still may be the main cause of state growth.

process OC automata, is a low cost solution. But it applies only after a successful automaton generation, and this cannot be the case when a state explosion occurs.

- *On the fly minimization:* It is based on an analysis of state explosion. The main reason seems to be that LUSTRE variables are defined during the whole program execution, without taking much care of their effective use. Although this is a nice feature of the language from a programmer's point of view, it leads the compiler to distinguish states which differ only on values that have no influence on the present and future sequence of outputs. This suggests a "demand driven" state generation strategy, where states are created if and only if their influence on the input output behavior of the program is asserted [BFH90a]. This strategy has been successfully implemented.

- *Source code optimization:* As mentioned above, some rules are known which could reduce the automaton size, but cannot be sensibly edicted as programming rules. The idea is then to take advantage of the large versatility of LUSTRE programs which is due to its mathematical aspect (for instance the definition principle) so as to use these rules as optimizing rules. There are experiments being carried on in this direction as well.

Transition code size: Besides the automaton size, it happens that the codes of transitions become exceedingly large. This results from an inadequacy of the scheduling algorithm which produces that sequential code. One of its tasks consists of transforming conditional *expressions* into conditional *statements* and the order according to which tests are opened and closed appears to be critical with respect to code size (cf. § 3.3). Heuristics are being investigated so as to solve this problem.

Modular compiler: It may also happen that a minimal automaton of a program still remains very large. This happens when the program is made of many quasi-independent parts, and then its number of states become as large as the product of state numbers of the parts. A good solution in this case would consist of generating an automaton for each part and then of linking together these automata. This raises two problems. First, it has been noted (§ 3.2) that modularly compiling pieces of LUSTRE programs is in general impossible. However [Ray88] proposed a method for identifying in a program those pieces that can be compiled separately. Second, this may result in a significant decrease of the code length, but at the expense of execution time. Although the method has not yet been implemented, it is foreseen that it should keep under the programmer's control, so as to reach a satisfactory balance between code length and execution time.

5.2 Distributed programming

Up to now, the only execution scheme considered for LUSTRE programs is a purely sequential one. This does not seem very consistent with the highly parallel aspect of the language and with the fact that most parallel languages such that OCCAM and ADA have parallel and concurrent execution schemes. There can be at least two reasons for that discrepancy:

- Parallelism in LUSTRE is intended towards expressiveness and adequacy with the culture of control systems engineers, and this is independent of any execution scheme.
- In contrast with the abovementioned languages, parallelism in LUSTRE is a fine grain one, and its concurrent execution would be rather inefficient. On the contrary, we have seen that very efficient sequential codes (with respect to execution time) can be generated, and furthermore, sequential execution allows the transition time to be accurately bounded.

However, many control and monitoring systems which constitute the main application domain of LUSTRE, are distributed systems for several reasons: performances, fault tolerance, location of sensors and actuators, etc...and these systems are most often programmed separately. This may not be a bad solution, as it may correspond to a modular decomposition of systems, but it frequently raises difficult debugging problems, and an overall validation of such systems is usually impossible.

An alternative method can be based on an automated tool producing distributed code from LUSTRE programs and user-provided distribution commands (for instance, "compute variable `X` on location `L1`"). This would allow a whole application to be programmed in LUSTRE without taking care of distribution problems, and then, this application could be easily debugged and validated using standard LUSTRE methods. Provided the automatically produced distributed program preserves LUSTRE semantics, it can be expected that any debugging and validation performed on the centralized program will also hold for the distributed one.

Such a tool, called OC2REP, is described in [BCP88], and has been implemented. Given an OC program and a set of distribution commands, it automatically produces several OC programs which communicate through FIFO queues thanks to statements such that:

```
put_type(i:location; exp:type);
```

whose execution at location `j` consists of inserting the value of `exp` in the queue `j` of location `i`, and:

```
get_type(j:location; var x:type);
```

whose execution at location `i` consists of waiting if queue `j` is empty, and else, of assigning the head of the queue to `x`.

Note that the queue mechanism and the fact that `put`s and `get`s are inserted in convenient order allow messages not to identify the transmitted values, but only the sending and destination locations. The distributed programs are well synchronized, deadlock free, and meet the functional semantics of LUSTRE[6]. Experiments also show that this method avoids difficult distributed debugging problems. However, accurate bounds on the transition times are difficult to get, and their evaluation constitutes a real problem.

5.3 Hardware issues

The adequacy of LUSTRE for the description of digital circuits has been shown in several papers [HLP86, HP86, TP90]. Moreover, it can be expected that circuit proof and validation may benefit from LUSTRE proof techniques. Another interesting issue is hardware design from boolean LUSTRE specifications and descriptions. Some work on this topic is currently undertaken in cooperation with Digital Equipment "Paris Research Laboratory" [Roc89]. The idea is to implement on hardware the network of operators corresponding to the program, and successful achievements have been obtained in this direction, using "programmable active memory" circuits [BRV90].

[6]This also applies to Esterel since the input of the tool is an OC program.

6 Conclusion

In this paper, the Lustre language, its main applications, and its associated tools have been presented. As concluding remarks, we will compare the Lustre approach with some alternative approaches, from both programming language and verification points of view.

6.1 Related programming languages

6.1.1 Dataflow

The dataflow model has been a basis of several programming languages, for instance [AW85, Gra82, BFM84, Bro89], and it has been given a nice formal definition by Kahn in [Kah74]. When trying to locate Lustre within the dataflow world, it looks very close to Lucid from a syntactical point of view. This similarity is not casual since Lucid was the first main reference in the design of Lustre. However, the final language is quite different from its model. This is due to the choice of the Kahn model as the basic one for Lustre: in this model, newly computed values can only be appended at the end of a sequence of already computed values, while Lucid model allows them to be appended anywhere in the sequence. This raises a lot of problems when efficient execution mechanisms are required, and it poorly meets the point of view of reactive systems. Thus, Lustre can be first seen as some restriction of Lucid to the Kahn model. But the latter soon appeared still too general when bounded memory and bounded reaction time were required. Clearly, recursive node call had to be forbidden, but also the use of sampling and blocking operators had to be strictly restricted for that purpose. This originated the concept of Lustre clocks which is the final distinguishing feature of the language.

6.1.2 Signal

Another language quite similar to Lustre is Signal (see this issue), and comparing both is not an easy task. A main issue here is their distinct semantical model; in our opinion Signal does not belong to the Kahn family of languages, which is based on functions over sequences, and on functional composition, but on a concept of "programming by constraints": each Signal construct denotes a finite-memory relation between "hiatonised" sequences, and a program is the intersection of such relations. A program has a bounded memory but it can be relational (i.e., non deterministic), and the object of SIGNAL clock calculus consists of finding an execution scheme such that the program be deterministic and deadlock-free. The free use of hiatons (i.e., "absent" data symbols) in the semantics makes SIGNAL a more powerful language than LUSTRE in the sense that the internal clock of program can be faster than the inputs faster clock. In our opinion, the drawback of the approach lies in the fact that the clock calculus is much more complex, and can hardly be mentally performed by a programmer.

6.1.3 Imperative synchronous languages

Most synchronous models and languages are imperative ones — e.g., SCCS [Mil83], ESTEREL, SML, STATECHARTS — and therefore their programming style is very different. Comparison experiments undertaken with ESTEREL showed that some problems could fit better with the imperative style, while others did not. This seems to indicate that a good reactive programming toolbox should offer the possibility of mixing both approaches. As both languages share many tools in common, this may become a practical objective in the future.

It should also be noted that the data-flow aspect of LUSTRE makes it less dependent on synchronous execution schemes than imperative languages. For instance a denotational semantics of LUSTRE is given in [Ber86], which does not impose a synchronous execution. This may open the door to many asynchronous execution schemes together with their semantical interpretation.

6.2 Proof techniques

The use of LUSTRE as a language for expressing program properties allows it to be compared with so-called "real-time" logics [MM84, SMSV83, JM86, AH94]. These logics are mainly obtained by adding a quantitative time dimension to ordinary temporal logics where time is only seen as an ordering of events. Our proposal differs in that we remain within the framework of temporal logics, and consider time as a given external event. This presents two advantages: first, the logic does not grow in complexity, and it allows a multiform concept of time to be handled. On the same topic, we have also stressed in the paper the interest of using the same language for both writing programs, and expressing properties to be satisfied by these programs.

Concerning proof techniques, we first began by considering inductive methods, based on an axiomatic approach. Though some work has been done in that direction [CPHP87], it soon appeared that methods based on state enumeration ("model checking") could be more efficient. Several improvements of the method in the particular case of LUSTRE are described in the paper.

Acknowledgments: *Many people have been involved in the project: Jean-Louis Bergerand and Eric Pilaud played a great part in the language definition, John Plaice defined and implemented the* LUSTRE-V2 *compiler, Anne-Cécile Glory and Farid Ouabdesselam were involved in the work on program specification and proof within the framework of a contract with Merlin-Gerin company.*

The LUSTRE *project received constant attention and help from Gérard Berry and the* ESTEREL *team. In particular, we are indebted to them for the compiling technique of* LUSTRE. *We are also indebted to Albert Benveniste for its energy spent in promoting synchronous programming. Finally, we thank Costas Courcoubetis and the referees for their careful rereading of the manuscript.*

References

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model checking of real-time systems. In *Fifth IEEE Symposium on Logic in Computer Science*, Philadelphia, 1990.

[AH94] R. Alur and T.A. Henzinger. A really temporal logic. *JACM*, 41(1), January 1994.

[AW85] E. A. Ashcroft and W. W. Wadge. LUCID*, the data-flow programming language.* Academic Press, 1985.

[BCP88] B. Buggiani, P. Caspi, and D. Pilaud. Programming distributed automatic control systems: a language and compiler solution. Technical Report SPECTRE L4, IMAG, Grenoble, Grenoble, July 1988.

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

[Ber89] G. Berry. Real time programming: Special purpose or general purpose languages. In *IFIP World Computer Congress*, San Francisco, 1989.

[BFH90a] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In R. Kurshan, editor, *International Workshop on Computer Aided Verification, Rutgers*, June 1990.

[BFH90b] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. On the verification of safety properties. Technical Report SPECTRE L12, IMAG, Grenoble, Grenoble, March 1990.

[BFM84] S. A. Babiker, R. A. Fleming, and R. E. Milne. A tutorial for LTS. RR 225. 84. 1, Standard Telecommunication Laboratories, 1984.

[BL85] D. Borrione and C. Le Faou. Overview of the CASCADE multilevel hardware description language and its mixed mode simulation mechanisms. In *Computer Hardware Description Languages and Their Applications*. Elsevier Science, North Holland, 1985.

[Bro89] M. Broy. Functional specification of time sensitive communicating systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozemberg, editors, *REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*. LNCS 430, Springer Verlag, May 1989.

[BRV90] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. In J. McCanny, J. McWhirter, and E. Swartzlander, editors, *Systolic Array Processors*. Prentice-Hall, 1990.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[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*, Munchen, January 1987.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yanakakis. Memory efficient algorithms for the verification of temporal properties. In R. Kurshan, editor, *International Workshop on Computer Aided Verification, Rutgers*, June 1990.

[Fer88] J.-C. Fernandez. Aldebaran : un système de vérification par réduction de processus communicants. Thesis, Université Joseph Fourier, Grenoble, Grenoble, France, 1988.

[Glo89] A-C. Glory. Vérification de propriétés de programmes flots de données synchrones. Thesis, Université Joseph Fourier, Grenoble, Grenoble, France, December 1989.

[Gra82] J. R. Mc Graw. The VAL language: Description and analysis. *ACM TOPLAS*, 4(1), January 1982.

[HLP86] N. Halbwachs, A. Lonchampt, and D. Pilaud. Describing and designing circuits by means of a synchronous declarative language. In *IFIP Working Conference "From HDL Descriptions To Guaranteed Correct Circuit Designs"*, Grenoble, September 1986.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7), 1992.

[Hol87] G. J. Holzmann. On limits and possibilities of automated protocols analysis. In *IFIP WG-6.1 7th. International Conference on Protocol Specification, Testing and Verification*, Zurich, 1987. North Holland.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems,* NATO *Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems.* Springer Verlag, 1985.

[HP86] N. Halbwachs and D. Pilaud. Use of a real-time declarative language for systolic array design and simulation. In *International Workshop on Systolic Arrays*, Oxford, July 1986.

[HPOG89] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.C. Glory. Specifying, programming and verifying real-time systems, using a synchronous declarative language. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*, Grenoble, June 1989. LNCS 407, Springer Verlag.

[JM86] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-2, 1986.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74*. North Holland, 1974.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *TCS*, 25(3), July 1983.

[MM84] B. Moszkowski and Z. Manna. Reasoning in interval temporal logic. In *Workshop on Logics of Programs*. LNCS 164, Springer Verlag, 1984.

[PH88] D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In M. Joseph, editor, *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Warwick, September 1988. LNCS 331, Springer Verlag.

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

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Symp. on the Foundations of Computer Science*, Providence R.I., 1977. IEEE.

[PS87] J. A. Plaice and J-B. Saint. The LUSTRE-ESTEREL portable format. Unpublished report, INRIA, Sophia Antipolis, 1987.

[Ray88] P. Raymond. Compilation séparée de programmes LUSTRE. Technical Report SPECTRE L5, IMAG, Grenoble, Grenoble, June 1988.

[Roc89] F. Rocheteau. Programmation d'un circuit massivement parallèle à l'aide d'un langage déclaratif synchrone. Technical Report SPECTRE L10, IMAG, Grenoble, June 1989.

[RRSV87] J. L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in XESAR of the sliding window protocol. In *IFIP WG-6.1 7th. International Conference on Protocol Specification, Testing and Verification*, Zurich, 1987. North Holland.

[RS89] V. Roy and R. de Simone. An AUTOGRAPH primer. Technical Report INRIA, Sophia-Antipolis, May 1989.

[SMSV83] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An interval logic for higher-level temporal reasonning: language definition and examples. Research Report CSL-138, Computer Science Lab. , SRI International, February 1983.

[TP90] G. Thuau and D. Pilaud. Using the declarative language LUSTRE for circuit verification. In *Workshop on Designing Correct Circuits*, Oxford, September 1990.

[Ver86] D. Vergamini. Verification by means of observational equivalence on automata. Technical Report 501, INRIA, 1986.