Specifying, Programming and Verifying Real-Time Systems Using a Synchronous Declarative Language

N. Halbwachs, D. Pilaud, F. Ouabdesselam
Laboratoire de Génie Informatique - Institut IMAG
B.P. 53, 38041 Grenoble Cedex - France

A-C. Glory
Merlin-Gerin / SES
38050 Grenoble Cedex - France

Abstract: We advocate the use of the synchronous declarative language LUSTRE as a unique language for specifying and programming real-time systems. Furthermore, we show that the finite automaton produced by the LUSTRE compiler may be used for verifying many logical properties, by model checking. The paper deals with an example program, extracted from a railways regulation system.

INTRODUCTION

The formal validation of programs is an especially important goal in the design of real-time systems, as these systems are involved in such crucial applications as automatic plant regulation and supervision, railways, aircraft or spacecraft control, etc... As usual, the validation of these programs raises the problems of specification and programming languages, together with the problem of verification methods. In particular, the question of the adequation between the specification and the programming languages is an important one. However the adressed domain presents some characteristic features:

- From its "real time" nature, some notion of time must be taken into account during the design and the verification processes: clearly, the correctness of a real-time program does not only depends on *what* it does, but also on *when* it does it.
- From our experience, the crucial properties of a real-time program (for instance "whenever a dangerous situation occurs, an alarm must be set") rarely depend upon deep arithmetic theorems. So, one can hope that they can be proven only by analysis of the logical dependencies of logical events.

As a consequence of these remarks, the present paper makes the following propositions:

This work was partially supported by MRT under contract nr 88-E-0444

1) Use a programming language which be formal enough to be easily extended towards a convenient specification language. So, the same concepts are used in the program and in the expression of the desired properties.

2) Provide the language with a notion of time which be together simple, formal, and general enough. Such a goal is approached here by means of a *synchronous* model: a program is intended to react instantaneously to external events; more precisely, no external event is supposed to occur during the reaction of the program.

3) Design a model checker to evaluate the desired properties over some logical abstractions of the program behaviour.

This paper intends to show that the programming language LUSTRE satisfies the first two propositions. LUSTRE is a synchronous, data-flow language, inspired from LUCID [16]. A program is made of equations — in the mathematical sense — specifying identities between *flows* . A flow is a sequence of values together with a *clock* specifying the sequence of instants when these values appear. So a flow equation may be viewed as an invariant assertion specifying that, at any instant of their common clock, the two members of the equation are equal. This point of view suggests that the language can be easily extended to express more general temporal assertions. The third proposition can be satisfied using the LUSTRE compiler: As for other synchronous languages [1,2], the compiler produces efficient code by synthesizing the sequential control structure of the object code. This structure is a finite automaton which summarizes many logical properties of the program. So, existing model checkers [4,13,14] can be applied to this automaton. However, we shall see that it seems useful to design a verification tool especially suited for LUSTRE.

The first part of the paper is devoted to a rapid presentation of LUSTRE (more complete presentations of the language, its formal semantics and its compiler, can be found in [3,9,10]). Its use is illustrated in Part 2, on an example program wich will be used throughout the paper. Then we propose and illustrate an extension of LUSTRE to specify desirable properties of programs (Part 3). In part 4, we show how the LUSTRE compiler may be used to generate an automaton on which the specified properties can be checked. In conclusion, we sketch the presentation of a verification tool specifically suited to LUSTRE programs: such a tool, which involves incremental automata generation, is under development in our

1. THE LANGUAGE LUSTRE

1.1 Variables, Clocks, Equations, Data operators

As indicated above, any variable or expression in LUSTRE denotes a sequence of values. Moreover, each variable has a clock, and is intended to take the n-th value of its sequence at the n-th tick of its clock. A program has a cyclic behaviour, which defines its *basic clock* : a variable which is on the basic clock its n-th value at the n-th cycle of the program. Other, slower, clocks may be defined by means of boolean variables: any boolean variable C may be used as a clock, which is the sequence of cycles when C is true.

For instance, consider a boolean variable C on the basic clock, and a boolean variable C′ on the clock C. The following table shows the different time scales they define:

basic time	0	1	2	3	4	5	6	7
C	true	false	true	true	false	true	false	true
time on C	0		1	2		3		4
C′	false		true	false		true		true
time on C′			0			1		2

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 ($x_0=e_0$, $x_1=e_1$, ..., $x_n=e_n$, ...) where (e_0, e_1, ..., e_n, ...) is the sequence of values of the expression E. Moreover, the equation states that X has the same clock as E.

Expressions are built up from variables, constants (considered to be infinite constant sequences on the basic clock) and operators. Usual operators on values (arithmetic, boolean, conditional operators) 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. The operands of a data-operator must be on the same clock, which is also the clock of the result.

1.2 Sequence Operators

In addition to the 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 X=(x_0,x_1,...,x_n,...) then

pre(X) = (nil,x_0,x_1,...,x_{n-1},...)

where nil is an *undefined* value, akin to the value of an uninitialized variable in imperative languages. The clock of pre(X) is the clock of X.

To initialize variables, the -> (followed by) operator is introduced. If X=(x_0,x_1,...,x_n,...) and Y=(y_0,y_1,...,y_n,...) are two variables (or expressions) of the same type and the same clock, then

X->Y = (x_0,y_1,y_2,...,y_n,...)

The last two operators are used to define expressions with different clocks:

If E is an expression, B is a boolean expression, and if E and B are on the same clock, then "E when B" is an expression on the clock defined by B, and whose sequence of values is the sequence of values taken by E when B is true.

If E is an expression on a clock CK different from the basic clock, then "current(E)" is an expression on the same clock as CK, whose value at each cycle of this clock is the value taken by E at the last cycle when CK was true.

The following table shows the effect of these operators.

B = (	false	true	false	false	true	true	false	true	...)
X = (	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	...)
Y = X when B = (		x_1			x_4	x_5		x_7	...)
Z = current(Y) = (	nil	x_1	x_1	x_1	x_4	x_5	x_5	x_7	...)

The rules for statically computing the clock of any expression are presented in [3].

1.3. Program structure

The procedural abstraction in LUSTRE is called the *node* . A LUSTRE node defines a set of output parameters, and possibly a set of local variables, from input parameters, by means of a system of equations. Here is an example of node declaration, which define the rising edge of its boolean parameter: the output is true whenever the input rises from false to true; moreover, at the initial instant, the output is assumed to be false:

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

Such a node is instanciated in a functional way, and may appear in any expression. For instance, one can write the equation:

```
falling_edge_of_c = Edge(not c);
```

When a node returns several outputs, its instanciations are supposed to be tuples of variables. Tuples may appear in equations, and as parameters of some polymorphic operators: assuming that node N returns 3 parameters, one can write:

```
(x,y,z) = if c then N(a) else N(b);
```

Concerning clocks, a node behaves as a data-flow operator: its basic clock is the clock of its *basic input parameters* . The node performs a basic cycle when and only when these parameters are available. Other input or output parameters may be on slower clocks, but these clocks must be visible from inside and outside the node.

1.4. Assertions

Initially, assertions have been introduced in LUSTRE (as in ESTEREL) to specify some known properties of the environment. Such properties are used by the compiler to optimize the object code: for instance, knowing that some input events never occur simultaneously may allow some dynamic tests to be dropped out. There are two kinds of assertions in LUSTRE: The syntax of the former is

```
assert <boolean_expression>;
```

which specify that the boolean expression is invariantly true. The later states that a set of boolean expressions are exclusive, i.e. only one of them may be true at a given instant; its syntax is the

```
assert #<boolean_expression_list>
```

Any LUSTRE boolean expression may appear in an assertion. We shall see that assertions play an important role in the validation process, on one hand as a way of restricting the model, and on the other hand as they form the basis of our specification language.

1.5. Compilation: The automaton generation

After usual checking of types and clocks, the LUSTRE compiler expand any node call in the source program, so as to get a flat program, consisting in a single system of equations. Then it performs the synthesis of the control structure, which is outlined now. Control synthesis is an important goal in sequential code generation from declarative languages, since such languages don't contain any notion of control. We first remark that, in our language, what is currently implemented by means of control structures, is hidden behind boolean expressions (clocks, test conditions, ...). The control synthesis consists of computing, as far as possible, these boolean expressions at compile time. More precisely, any boolean memory (result of a "pre" or "current" operator with boolean input) will become a component of the control state of the code. So the control structure will be a finite automaton, each state of which corresponds to a reachable configuration of boolean memories (moreover, the initial state will be distinguished in order to evaluate the "->" operators). The generation process is described in [3,9]. We only illustrate it on a very simple example. Let us consider the node Edge as a whole program:

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

It has only one state variable: pre(b) .

- In the initial state, the output "edge" equals false. According to the value of "b", the next state will correspond either to "pre(b)=true" or to "pre(b)=false".
- In the state corresponding to "pre(b)=true", the output "edge" equals "b and false", so it is necessarily false. According to the value of "b", the next state will correspond either to "pre(b)=true" or to "pre(b)=false".
- In the state corresponding to "pre(b)=false", the output "edge" equals "b and true", so it equals the input. Again, the next state will correspond either to "pre(b)=true" or to "pre(b)=false", according to the value of "b".

So, we got the automaton of Fig. 1. In building the automaton, the compiler takes the assertions into account, avoiding the construction of states and pathes where the assertions are violated.

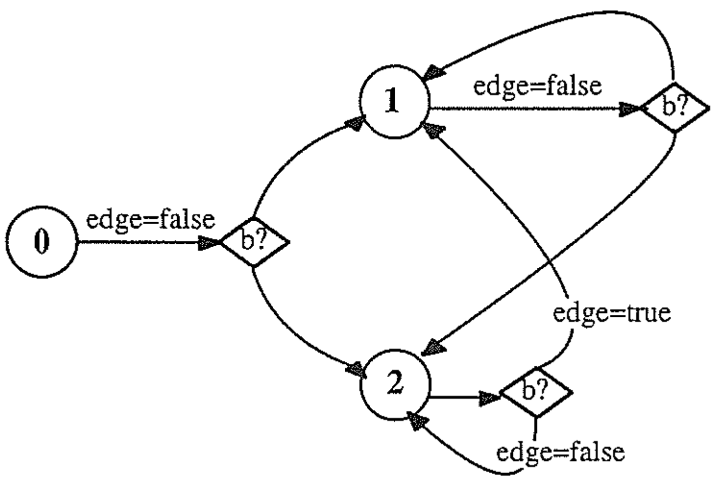


Fig.1: A control automaton

2. AN EXAMPLE PROGRAM

Let us illustrate the use of LUSTRE on a small example extracted from an actual railway regulation system: A track is divided into districts. At the boundary between two districts, there are two pedals which overlap each other, as shown in Fig. 2. The device must detect the traversal of each axle from on district to the other. It knows the state of the pedals, by means of two variables p1 and p2 (pi is true whenever there is a wheel on the corresponding pedal). It must compute two boolean variables: the output "from_left_to_right" (respectively "from_right_to left") is true when and only when an axle, coming from the left (resp.right) district, enters the right (resp. left) district. There can be only one axle in the zone Z, but an axle may turn back within the zone Z, and even oscillate on or about the zone Z.

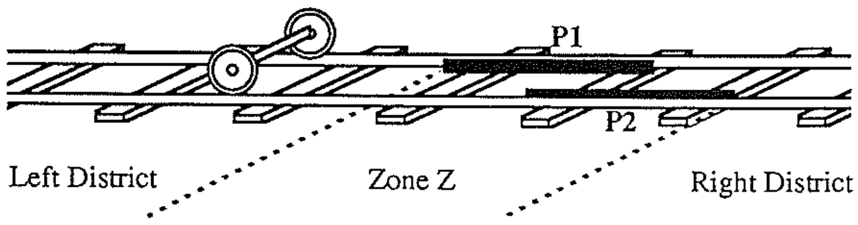


Fig. 2: The axle detector

Let us write the program which computes "from_left_to_right". We introduce two boolean variables: "last_in_left" and "exit_right". The former is true whenever the last axle which entered the zone Z came from the left district; the later is true whenever an axle leaves the zone Z to the right district. Using these variables, the output "from_left_to_right" can be defined by the equation

```
from_left_to_right = exit_right and last_in_left;
```

The variable "last_in_left" becomes true whenever an axle enters the zone Z from the left district, and becomes false whenever an axle enters the zone Z from the right district. Let us use a node "Switch", of general usage, whose boolean output "state" may be given an initial value "val_init", and which may be turned on and off by means of two variables "set" and "reset":

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

Now, we can define "last_in_left" by means of two new variables "enter_left" and "enter_right":

```
last_in_left = Switch(false, enter_left, enter_right);
```

Finally, noticing that and axle enters the zone Z from the left (resp. right) district whenever p1 (resp.p2) has a rising edge when p2 (resp.p1) is false, we get:

```
enter_left = Edge(p1) and not p2;
enter_right = Edge(p2) and not p1;
```

and similarly

```
exit_right = Edge(not p2) and not p1;
```

Defining "from_right_to_left" in a symmetrical way, we get the following node implementing the whole axle detector:

```
node Detector (p1,p2: bool) returns (from_left_to_right, from_right_to_left: bool);
var last_in_left, last_in_right, enter_left, enter_right, exit_left, exit_right: bool;
let
    from_left_to_right = exit_right and last_in_left;
    from_right_to_left = exit_left and last_in_right;
    last_in_left = Switch(false, enter_left, enter_right);
    last_in_right = Switch(false, enter_right, enter_left);
    enter_left = Edge(p1) and not p2;
    enter_right = Edge(p2) and not p1;
    exit_left = Edge(not p1) and not p2;
    exit_right = Edge(not p2) and not p1;
tel.
```

In addition, the system is assumed to work correctly only when any change in the pedal state is perceived separately. This assumption may be expressed in the program as an assertion stating that the edges of the pedal variables are exclusive. Moreover, we assume that, initially, there is no axle in the zone Z. The corresponding assertions follow:

```
assert #(Edge(p1), Edge(p2), Edge(not p1), Edge(not(p2));
assert not(p1 or p2) -> true;
```

Notice that many auxiliary variables have been introduced. It does not have any consequence concerning the generated code, since the compiler will select a minimal set of memories. However, it will be convenient later, for specifying program properties, to name any relevant expression.

3. SPECIFYING TEMPORAL PROPERTIES IN LUSTRE

3.1 Translating temporal properties into LUSTRE invariants

Turning to the validation problem, we consider the expression of desired properties of a program. Many formalisms have been proposed for this sake, most of them being inspired either from temporal logic [5,7] or from process algebras [6,15]. However, in order to reduce the user's effort, we are looking for a formalism being as close as possible to the programming language. It has been shown elsewhere [8] that LUSTRE can be viewed as a subset of linear temporal logic. Here, we investigate the expressive power of a specification language consisting of invariant assertions written in LUSTRE itself. Of course, only *safety* properties may be expressed by this way, but we shall show that, using memory operators and recursive definitions, the range of such properties is quite large. In addition, from our experience, the critical properties of a real time system almost always fall into this class: as a matter of fact, nobody cares whether an alarm *eventually* follows a dangerous situation, but rather whether it occurs within a given delay! From this remark, other authors [13] have identified some temporal constructions which appear particularly useful in expressing real-time properties. Examples of such constructions are:

"always P from Q to R"

or

"once P from Q to R"

respectively stating that, in any path leading from a state satisfying Q to a state satisfying R, the property P always holds (resp. the property P holds at least once). As an illustration of the expressive power of LUSTRE invariants, we shall try to express such properties in the following sense: we shall define boolean expressions which are invariantly true on a model if and only if the model satisfies the property. We first need a slightly formal development in order to explain what we call a model, and to make precise the semantics of the considered properties.

A model for a LUSTRE program is a tree of stores. A store, noted σ, is a partial function from variable identifiers to values. With the root of the tree is associated a store σ_{-1} associating with any variable the value nil. It corresponds to a virtual instant "-1". The sons of this root correspond to the possible initial states of the variables. More generally, any branch in the tree corresponds to a possible execution of the program: let $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$ be the sequence of stores encountered along such a branch (we call such a sequence a *history*) and x be a variable on the basic clock, then $\sigma_0(x), \sigma_1(x), \ldots, \sigma_n(x), \ldots$ is the sequence of values of x in the corresponding execution. In particular, the value of "pre(x)" according to the history $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_n$ is the value of "x" according to the history $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_{n-1}$. The rules for building the execution tree of a LUSTRE program are given by the natural semantics of [9]. Now, a formula consisting of a LUSTRE boolean expression is true on a vertex of the tree, if and only if its value is true when evaluated according to the history associated with the branch leading to the considered vertex. Such a formula is satisfied by a LUSTRE program if and only if it is true on every vertex (except the virtual root) of the associated tree.

Now, "always P from Q to R" is satisfied by a tree if and only if for any branch $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$ of the tree, for any integer i such that Q is true on $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_i$,

- either there exists $j>i$ such that R is true on $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_j$, and for any k, $i \leq k < j$, P is true on $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_k$,
- or for any $k \geq i$, P is true on $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_k$.

Similarly, "once P from Q to R" is satisfied by a tree if and only if for any branch $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$ of the tree, for any integer i such that Q is true on $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_i$, if there exists $j>i$ such that R is true on $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_j$, then there exist k, $i \leq k < j$, such that P is true on $\sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_k$.

Let us show how these properties can be translated into invariant LUSTRE properties:

- We shall translate "always P from Q to R", where P,Q,R are LUSTRE boolean variables, into a boolean variable which is true when and only when
  - either P was continuously true since the last time Q was true
  - or R has been true at least once strictly after the last time Q was true

  So we write:

```
node Always_From_To (P,Q,R: bool) returns (X: bool);
let
    X = Always_Since(P,Q) or Once_After(R,Q);
tel.
```

- The boolean variable corresponding to "once P from Q to R", is true when and only when
  - either R is false
  - or P has been at least once true since the last time Q was true

  We get the following node:

```
node Once_From_To (P,Q,R: bool) returns (X: bool);
let
    X = not R or Once_Since(P,Q);

tel.
```

Other nodes used in the previous definitions are given without comments:

```
node Always_Since (P,Q: bool) returns (X: bool);
-- returns true whenever P has been continuously true since the last time Q was true
let
    X =  if Before_Or_At(Q) then true
         else if pre(Q) then pre(P)
         else pre(P and X);
tel.

node Once_After (P,Q: bool) returns (X: bool);
-- returns true whenever P has been at least once true strictly after the last time Q was true
let
    X =  if Before_Or_At (Q) then true
         else if pre(Q) then false
         else pre(P or X);

tel.

node Once_Since (P,Q: bool) returns (X: bool);
-- returns true whenever P has been at least once true since the last time Q was true
let
    X =  if Before_Or_At (Q) then true
         else if pre(Q) then pre(P)
         else pre(P or X);

tel.
```

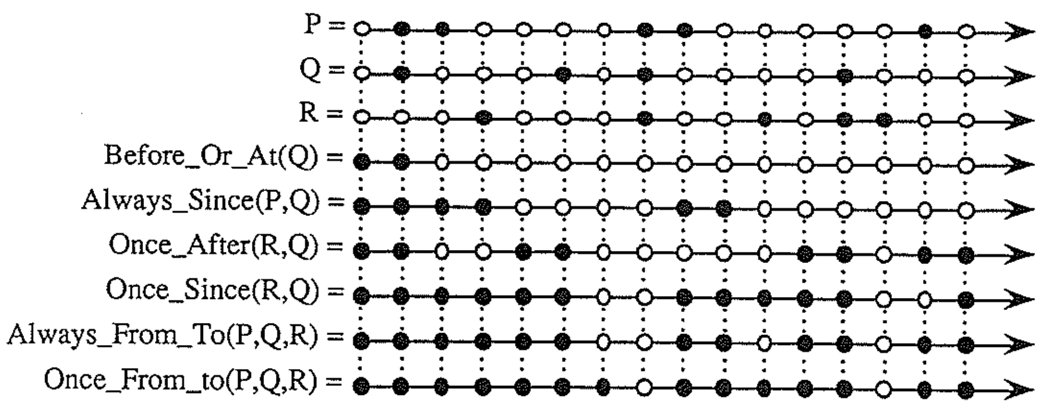


Fig. 3 : Behaviour of some temporal operators

```
node Before_Or_At (Q: bool) returns (X: bool);
-- returns true whenever Q  has never been true, or is true for the first time
let
   X = true -> pre(not Q and X);
tel.
```

Fig. 3 illustrates the behaviour of these operators, with a graphical representation where black and white circles respectively stand for "true" and "false"

3.2 Application to the axle detector

Coming back to the program of section 2, we can first express its desired properties, in a quite abstract way: for instance, we write that whenever an axle leaves the zone Z from the right side, and the last entering was at the left side, the output "from_left_to_right" is emitted:

not from_left_to_right or (exit_right and not Once_After(enter_right, enter_left))

Now, we would like to get an abstraction of the axle detector. We shall simulate the behaviour of a train, which receives two commands "left" and "right": when receiving the command "left" (resp. "right") an axle straightly traverses the zone Z from right to left (resp. from left to right). So, the simulation program computes the pedal state as follows:

"p1" is set whenever
- either the command "right" was sent at the previous instant
- or any command was sent two instants before
- or the command "left" was sent three instants before

and conversely for "p2".

As it is often convenient when dealing with multiple delays, we shall use an initializing "previous" operator, in order to avoid problems with "nil":

```
node Pre(cond, val_init: bool) returns (Pre: bool);
let
   Pre = val_init -> pre(cond);
tel.
```

The simulation program is as follows:

```
node Train(left,right: bool) returns (p1,p2: bool);
let
   p1 = Pre(right or Pre(right or left or Pre(left,false),false),false);
   p2 = Pre(left or Pre(left or right or Pre(right,false),false),false);
tel.
```

Let us put the train and the detector together, assuming that no command is sent when an axle is in the zone Z:

```
node Z_Zone(left,right: bool) returns (to_left,to_right: bool);
var p1,p2,occupied:bool;
let
   (p1,p2) = Train(left,right);
   (to_right,to_left) = Detector(p1,p2);
   occupied = p1 or p2;
   assert #(occupied, left, right);
tel.
```

Now, we want to describe the whole system without regards to pedals. If we observe it only when the zone Z is idle, it must behave only as a delay on the commands, i.e. "to_left" (resp. "to_right") is sent whenever "left" (resp. "right") was true at the last time the zone was idle. This property may be written as follows, using "idle" as a clock:

```
to_left when idle = Pre((left,false) when idle);
to_right when idle = Pre((right,false) when idle);
   where idle = not occupied
```

Comments:

- Notice the application of the "when" operator to tuples: "Pre((left,false)when idle)" stands for "Pre(left when idle, false when idle)" and is therefore equivalent to "(false when idle) -> pre(left
- The above property mixes equations, whose invariance must be proved, with a LUSTRE definition (definition of idle).
- This example shows the importance of clocks, as a way of defining observation criteria.

4. VERIFYING TEMPORAL PROPERTIES

4.1 Model Checking

The verification of program properties by model checking consists of building an extended representation of possible program behaviours (generally as a state graph), and of exhaustively checking the considered properties on this representation. Clearly, this approach only works for properties which can be checked on a finite representation. Concerning LUSTRE, our first task concerns the construction of such a finite abstraction of the infinite tree of program behaviours. Fortunately, this abstraction is already performed by the LUSTRE compiler, when generating the control automaton of the object code: Formally, the automaton results from a folding of the behaviour tree, according to the greatest

• Two stores are considered equivalent if and only if they give the same values to each boolean memory (result of a "pre" or "current" operator). Other store equivalences could be considered, for instance by replacing "boolean" by "bounded range", in the definition above.

• Two vertices of the tree bisimulate each other iff
- their associated stores are equivalent, and
- in response to the same boolean inputs, they lead to vertices which bisimulate each other.

So, existing model checkers may be applied to our control automata. As a matter of fact, the use of EMC [4] on automata produced by the ESTEREL compiler has been successfully experimented. Let us briefly discuss such a solution for LUSTRE (as a conclusion from an experience using XESAR [12]):

• In addition to the advantage of using existing tools, it provides the power of whole branching-time temporal logic ("eventuality" and "possibility" modalities, nesting of modal operators).

• However, the associated specification languages have been designed in connection with imperative programming languages: concepts addressed there are actions and labels, which make no sense in a LUSTRE program, and the translation of our LUSTRE properties in these language is far from being obvious.

• Moreover, in order to be accepted by these tools, the automaton must be enriched of many intermediate states. However, the notion of state is strongly meaningfull in LUSTRE: for instance, the "pre" operator refers to the actual preceding state, not to an artificial intermediate state. So, the intermediate states must be abstracted, an operation which can be complex.

• "Last but not least", there is a semantic problem in the evaluation of LUSTRE properties on an automaton. Consider, for instance, the property "$P \Rightarrow$ pre(Q)" where P and Q are some LUSTRE expressions, and assume P holds in some state s, which is reachable from two states s' and s'' (cf. Fig. 4.a). If Q holds in s' but not in s'', we cannot decide of the value of "$P \Rightarrow$ pre(Q)". This problem results from the fact that the folding of the initial tree may depend on the considered property. The evaluation of a given property may necessitate an expansion of the automaton (cf. Fig. 4.b). A solution consists of calling the LUSTRE compiler to perform this extension.

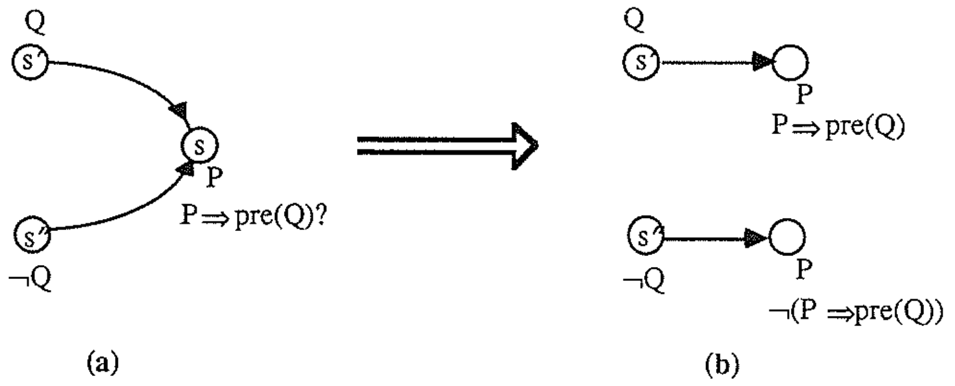


Fig. 4 : Automaton expansion

4.2 Using the LUSTRE compiler as a model generator

The proposed technique, for verifying the invariance of a LUSTRE formula F on a program P, consists of:

- augmenting the program P into a program P′, computing the formula F;
- compiling the program P′, to get its control automaton;
- checking, on the automaton, that the variable corresponding to F is never set to false.

Let us illustrate this technique on some examples.

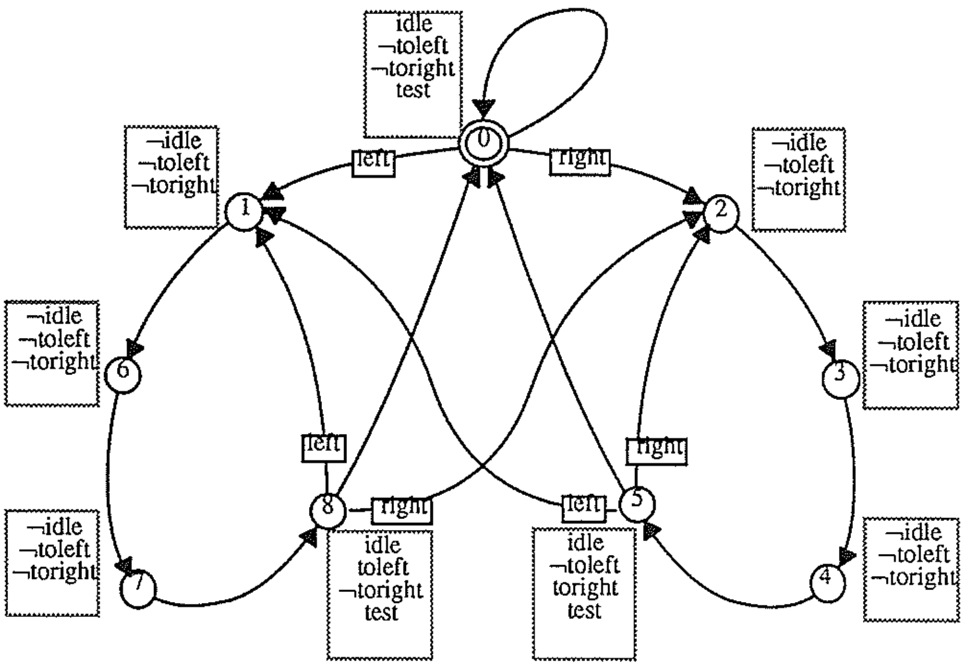
Proving properties of the axle detector

We detail the proof of the invariance of the property

```
to_left when idle = Pre((left,false) when idle);
```

on the program Z_Zone. This program is first augmented in order to compute a new variable "test", which is true whenever the property is true. We have also to introduce the variable "idle", which must be an output since it is the clock of "test" (parameters' clocks must be visible from outside a node, cf.§1.3). The resulting program is

```
node Z_Zone(left,right: bool) returns (to_left,to_right,idle,test: bool);
var p1,p2,occupied:bool;
let
    (p1,p2) = Train(left,right);
    (to_right,to_left) = Detector(p1,p2);
    occupied = p1 or p2;
    assert #(occupied, left, right);
    idle = not occupied;
    test = (to_left when idle = Pre((left,false) when idle));
tel.
```



idle ¬toleft ¬toright test
¬idle ¬toleft ¬toright
left
right
0
1
2
3
4
5
6
7
8
idle toleft ¬toright test
idle ¬toleft toright test

Fig. 5 : The verification automaton for the axle detector

Fig. 5 shows the control automaton corresponding to this program. One can easily see that the output "test" is only defined in states 0, 5 and 8 (when "idle" is true), where it is set to true.

Proving program equivalence

An other important application field of the proposed technique, is the proof of equivalence between several versions of the same program. Given two programs Π and Π′, we can prove that they have the same behaviours by the following procedure:

- write a main program, containing Π and Π′ in parallel, provided with the same inputs;
- compute, as the output of the main program, the result of a comparison of the outputs of Π and Π′ (Fig. 6);
- check on the resulting automaton that the output is never set to false.

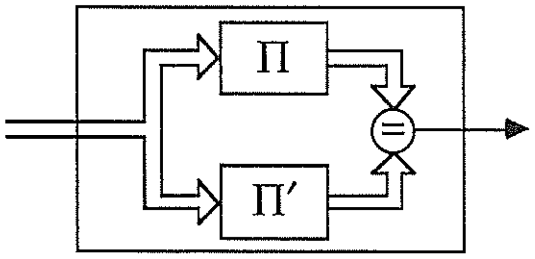


Fig. 6 : Comparing two programs

Of course, using assertions and filtering by clocks, other equivalence criteria may be taken into account.

For instance, we can prove that

Once_After(P, Q)

is equivalent to

Once_Since(P and not Q, Q)

by writing the following node:

```
node Test_Equivalence (P,Q: bool) returns (equivalent: bool);
let
   equivalent = (Once_After(P,Q) = Once_Since(P and not Q, Q));
tel.
```

4.3 Towards an integrated tool for compiling and verifying LUSTRE programs

The drawback of the proposed proof technique is that it involves the modification and recompilation of the program whenever a new property has to be proved. Moreover, some tool is needed for automatic checking of invariance on automata. So, we are considering the design of a verification system

- LUSTRE program compilation, with automaton generation at several levels of detail;
- LUSTRE property incremental compilation, with automaton expansion: it would make possible to enrich a program with the computation of new formulas, without compiling again the program.
- Property checking on the automaton, which may involve other properties than only invariance: Once a formula has been computed in each state, questions concerning eventuality and possibility could also be considered.
- Errors explanation, which is a critical issue in program verification: When some desired property is not satisfied, the user needs some hints in determining which is erroneous, of the program or the specification, and in locating, as precisely as possible, the source of the error. Such a tool is available in the system XESAR [11].

CONCLUSION

There are two traditional approaches to program verification: In the former, the specification consists of formulas written in some logic (Hoare's logic, temporal logic,...) and the verification consists in proving that these formulas are valid on the model represented by the program. In the later approach, the specification and the program are both written in an executable formalism, so the verification reduces to proving program equivalence (for some equivalence criterion). In this paper, we have tried to conciliate these two approach, using a logic based programming language, both for programming and property specification. Of course, as any language, LUSTRE needs some learning and experience, but it should be clear that the use of only one language reduces the user's effort, both concerning learning and conceptualization.

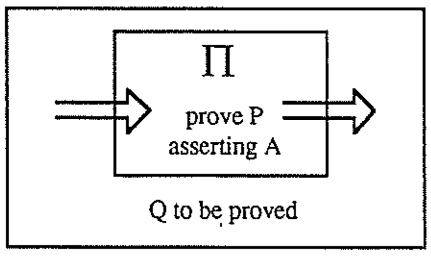
The usefulness of assertions has been shown. Assertions form an incursion of specifications into the programs. Their importance will also be illustrated in modular proofs.

Concerning specification, we restricted ourselves to invariant properties. The reasons of this restriction are twofold:

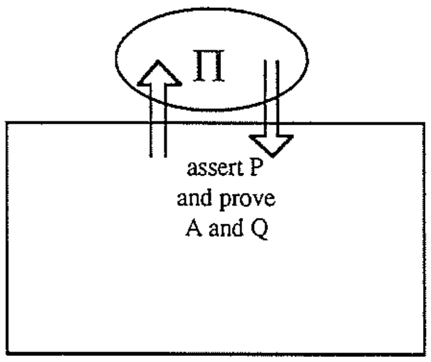
- On one hand, the invariant properties that can be expressed in LUSTRE, using memory operators and recursive definition, are quite powerful. We are not convinced that more general properties are needed, in practice, for expressing critical properties of real-time systems.
- On the other hand, these properties are extremely easy and cheap to check on an automaton: you never need to keep the whole automton in store (states may be considered once at a time) not ever to keep track of the transition relation. So, we can hope to deal with larger problems than more general

Turning to the verification method, some comments must be made: All the given examples deal with boolean programs. As a matter of fact, in the present state of our tools, the only properties we can prove concern the control structure of the program. However, they could be extended to deal with bounded integer variables (as done in XESAR), the only requirement being that the number of states remain finite (and not too large). Concerning the size of automata which can be processed (the central problem in model checking), we have some good reasons in believing that the approach is realistic:

- Experiences with XESAR show that, in its present state, the system can reasonably deal with graphs of about 1,000,000 states, a size compatible with many practical problems. Moreover, as noticed above, we can expect better results if we restrict ourselves to invariant properties.
- The use of assertions allows a kind of modular proof, which can reduce the size of the considered automaton: You can separately prove a property P of the parameters of an internal node Π, and then consider Π as a part of the environment of the main program (thus avoiding the full expansion of Π in the main program), asserting P about the input/output of Π. Here, assertions and properties play symmetrical roles: If P has been proved for Π under some assertion A (Fig. 7.a), then P may be asserted outside Π provided A is proved (Fig. 7.b).



(a)



(b)

Figure 7 : Modular proof

Finally, what about real_time? One can argue that our proofs always rely on the assumption of synchrony, and that we prove nothing about the execution time of the programs. That's true, indeed, and its is a constant point of view in the synchronous approach. It can be justified by remarking that the assumption of synchrony can be verified (a problem which is not adressed in this paper), because the execution time of the code produced by our compilers is *mesurable* . As a matter of fact, the code corresponding to a transition of the automaton is linear (no loops , no recursion). It is the responsibility of the user to check that the maximum transition time is shorter than the minimum delay separating two distinct external events.

REFERENCES

1. Berry G., Couronné P., Gonthier G.: Synchronous programming of reactive systems, an introduction to ESTEREL. INRIA report nr.647, 1987.

2. Berry G., Gonthier G.: The synchronous programming language ESTEREL, design, semantics, implementation. INRIA report nr. 327, 1985, to appear in Science of Computer Programming.

3. Caspi P., Halbwachs N., Pilaud D., Plaice J.A.: LUSTRE a declarative language for programming synchronous systems. Proc. 14th. Annual ACM Symp. on Principles of Programming Languages, Munich, 1987.

4. Clarke E.M., Emerson E.A., Sistla A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems, 8(2), January 86.

5. Koymans R., deRoever W.P.: Examples of a real-time temporal logic specification. In The analysis of concurrent systems, Cambridge,September 83, LNCS vol. 207, Springer Verlag, 1985.

6. Milner R.: Calculi for synchrony and asynchrony. Theor. Comp. Sci., vol.28, 1983.

7. Moszkowski B., Manna Z.: Reasoning in interval temporal logic. Proc. Workshop on Logics of Programs, LNCS vol. 164, Springer Verlag, 1984.

8. Pilaud D., Halbwachs N.: From a synchronous declarative language to a temporal logic dealing with multiform time. Proc. Symposium on Formal Techniques in Real_Time and Fault_Tolerant Systems, Warwick,September 88.

9. Plaice J.A.: Sémantique et compilation de LUSTRE, un langage déclaratif synchrone. PhD. Thesis, Institut National Polytechnique de Grenoble,May 88.

10. Plaice J.A., Halbwachs N. : LUSTRE-V2 User's guide and reference manual. Technical Report L2, SPECTRE Project, IMAG, Grenoble,October 87.

11. Rasse A. : CLEO, interprétation de la non correction de programmes sur un modèle. Technical Report C10, SPECTRE Project, IMAG, Grenoble, June 88.

12. Ratel C. : Etude de la conformité d'un programme LUSTRE et de ses specifications en logique temporelle arborescente. Master Thesis, Institut National Polytechnique de Grenoble, June 88.

13. Richier J.L., Rodriguez C., Sifakis J., Voiron J.: Verification in XESAR of the sliding window protocol. Proc. IFIP WG 6.1 7th. International Conference on Protocol Specification, Testing and Verification, North Holland, Zurich, 1987.

14. Richier J.L., Rodriguez C., Sifakis J., Voiron J.: XESAR user's guide. Technical Report, SPECTRE Project, IMAG, Grenoble, September 87.

15. Vergamini D.: Verification by means of observational equivalence on automata. INRIA report nr. 501, 1986.

16. Wadge W.W., Ashcroft E.A.: LUCID, the data-flow programming language. Academic Press,