Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE *

Nicolas HALBWACHS, Fabienne LAGNIER
IMAG / LGI
Grenoble, France

Christophe RATEL
Merlin-Gerin / SES
and IMAG / LGI

June 10, 1994

Abstract

We investigate the benefits of using a synchronous data-flow language for programming critical real-time systems. These benefits concern ergonomy — since the dataflow approach meets traditional description tools used in this domain —, and ability to support formal design and verification methods. We show, on a simple example, how the language LUSTRE and its associated verification tool LESAR, can be used to design a program, to specify its critical properties, and to verify these properties. As the language LUSTRE and its use have been already published in several papers (e.g., [11, 18]), we put particular emphasis on program verification. A preliminary version of this paper has been published in [28].

1 Introduction

It is useless to repeat why real-time programs are among those in which errors can have the most dramatic consequences. Thus, these programs constitute a domain where there is a special need of rigorous design methods. We advocate a "language approach" to this problem, arguing that the programming language used has a direct influence in the quality of the software, from several points of view:

(i) The language should allow a natural description of the problem to be solved. In particular, it should be close to the traditional tools used in its application field.

(ii) The language should be formally sound, in order to support formal verification methods.

(iii) The language should be simple enough to minimize the risk of misunderstanding about program meanings.

These were our main goals in designing the language LUSTRE [11, 18]. To meet the criterion *(i)*, we started from the traditional description tools used in the design of process control systems: At a higher level, these tools consist of mathematical formalisms (differential equations, boolean equations, ...) while at a lower level, people often use data-flow nets (block-diagrams, analog schemas, switch or gate networks, ...). These two classes of tools are closely related: For instance,

*This work was partially supported by ESPRIT Basic Research Action "SPEC" and by a contract from Merlin Gerin

differential equations, finite difference equations, boolean equations can be straightforwardly translated, respectively, into analog schemas, block-diagrams and gate networks. At least the class of high level tools also meets our criterion *(ii)*, since they are derived from the mathematical language. Other authors (e.g., [23, 2]) claimed that such declarative formalisms are simpler and cleaner than usual imperative languages, where assignments, side-effects, aliasing, parameter passing mechanisms, are unnatural phenomena which are difficult to understand and manage. We agree with this claim, and therefore consider that these declarative formalisms constitute a good basis for designing a programming language meeting the criterion *(iii)*.

LUSTRE is a synchronous data-flow language, initially inspired from LUCID. As in LUCID, any LUSTRE variable or expression is considered to represent the sequence of values it takes during the whole execution of the program, and LUSTRE operators are considered to operate globally over these sequences. The synchronous nature of the language consists in assuming that all the variables and expressions in a program take the n-th value of their respective sequences *at the same time*. More concretely, a program is intended to have a cyclic behavior, an execution cycle consisting in computing the n-th value of each variable or expression. Basically, a program is a set of equations. When we write an equation "`X=E`", where `X` is a variable and `E` is an expression, we mean that the sequences of values associated with `X` and `E` are identical (it is an actual equation, in the mathematical sense), or equivalently, that at any cycle of the program, `X` and `E` take the same value.

Time and synchrony

The "real time" capabilities of the language are derived from this synchronous interpretation, like in other synchronous languages [6, 8, 13, 21, 24]: We consider the following logical notion of time: As soon as we can specify the order or simultaneity relations between events occurring both inside and outside the program, we can express time constraints about the behavior of the program. For instance, a constraint like

> *"any occurrence of a dangerous situation must be followed by the emission of an alarm within a delay of 2 seconds"*

will be expressed as

> *"after any occurrence of the event* `dangerous_situation` *there must be an occurrence of the event* `alarm` *before the 2nd next occurrence of the event* `second`*"*

This example shows that, in synchronous programming, the "real", physical time is considered as an external event (here `second`), which has no privileged nature. This is the *multiform time* point of view: Time may be counted in "seconds" as well as in "meters", since there is no conceptual difference between the two following requirements:

> *the train must stop within 10 seconds* and *the train must stop within 100 meters*

In a synchronous language, these constraints are expressed in similar ways, in contrast with languages like ADA, where "real" time is handled by special statements.

In LUSTRE, an event will be modeled by a boolean variable, whose value is *true* whenever the event occurs.

The synchronous interpretation is an abstract point of view which consider the program reaction time to be negligible with respect to the reaction time of its environment. The advantages of this abstraction have been pointed out [5, 4, 3] concerning the semantics cleanness and the fact that it reconciles concurrency and determinism. In particular [4] argues that synchronous languages are particularly well-suited for programming *reactive kernels* of real-time systems, while complex systems generally require the combination of asynchronous and synchronous modules. We do not claim, therefore, that synchronous languages are general purpose languages, but rather specialized tools to design such kernels.

One may wonder about the realism of the synchronous hypothesis, since it assumes that the machine *instantly* reacts to input events. In fact, it only assumes that the reaction time is short enough to accurately distinguish and order the incoming events. In practice, it can be checked by measuring the maximum reaction time of the program. Synchronous languages can be implemented in a particularly efficient and *measurable* way, following a technique developed for ESTEREL [6]. Two versions of LUSTRE compilers have been written [11, 20], which use this technique. The object code is structured as a finite automaton, a transition of which corresponds to a reaction of the program[1]. The code corresponding to such a transition is linear (loop-free), and its maximal execution time can be accurately bounded, on a given machine. Therefore, the validity of the synchrony hypothesis can be checked, and it is the only "real-time" issue in the synchronous approach. All the design and verification we will perform rely on that checking.

In Section 2, we briefly present the language, whose use is illustrated in Section 3 on a small example adapted from an actual subway device.

Specification and verification

The remainder of the paper is devoted to program verification. As said before, the synchronous approach limits the "timing" verification to checking the validity of the synchrony hypothesis. So, the kind of verification we have in mind is similar to standard verification on transition systems. It has nothing to do with methods and models (like, e.g., [16, 26]) taking into account the program execution time — which is always assumed to be zero in the synchronous model.

Moreover, our goal is not to prove the correctness of a program with respect to some complete specifications, but rather to express and verify some critical properties. For instance, in an aircraft flight control system, an error in the speed computation can have only slight consequences, whereas it is critical that the undercarriage be locked when landing. Our claim is that these critical properties are usually simple, and can be verified by means of available automatic techniques. The main reason for this claim is that experience shows that most of these properties are "safety" properties which state that a given situation should never occur, or that a given statement should always hold, in contrast with "liveness" properties which state that a given situation should eventually occur in the future[2]. 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:

- Safety properties can be checked on program abstractions: Intuitively, one can simplify a given program P into an abstract program P', having "more behaviors" than P. If P' satisfies a safety property, so does P. This abstraction technique is valuable all the more as experience shows that the considered critical properties seldom depend on numerical

[1] This compiling technique is specific to synchronous languages, since, in an asynchronous language, the non deterministic interleaving of asynchronous actions would involve immediately a combinatorial explosion of the automaton size.

[2] As a matter of fact, liveness properties are often introduced to "abstract" response time constraints, which must be taken fully into account in a real-time system.

relations and computations, but more often on logical dependencies over events. So their proof can often be handled on finite state abstractions of programs.

- Safety properties can be checked on program *states*, rather than on *execution paths*. In the approach we will propose, any verification problem amounts at proving that some program never outputs the value *false*. In the finite state case, such a verification can be done by a simple traversal of the state graph, without keeping track of *any* path in that graph. Very efficient methods [22] have been proposed for such a traversal.

- Safety properties can be modularly verified: With any process composition operator $\star$, one can easily associate an operator $\underline{\star}$ such that, for any two processes P_1 and P_2, respectively satisfying safety properties ϕ_1 and ϕ_2, their composition $P_1 \star P_2$ satisfies $\phi_1 \,\underline{\star}\, \phi_2$. So, a verification problem can be decomposed into simpler ones.

So, our ambition is rather modest — since we restrict ourselves to checking safety properties on finite state (abstractions of) programs — with the hope of getting efficient tools tackling many real-life cases.

In view of this discussion, we propose a method for specifying and checking simple safety properties about LUSTRE programs. In Section 4, we show that we can take advantage of the declarative nature of LUSTRE, to express the properties in the same language: It has been shown [7] that any safety property about a program can be expressed by the invariance of some boolean LUSTRE expression. This is due to the fact that LUSTRE can be viewed as an executable temporal logic. Concerning ergonomics, there is an obvious advantage of using a full programming language instead of a temporal logic. In particular, the user can define its own temporal operators[3], thus reducing the complexity of property expression.

Moreover, LUSTRE provides a means of expressing assumptions about the program environment. This is an essential feature, since the interaction of a real-time program with its environment is particularly important. In general, the properties of a real-time program are intended to hold only under some assumptions about the behavior of its environment, and these assumptions can be quite complex. So, the verification process deals with three entities: The program, the property (expressed by an invariant) and the assumptions under which the property is intended to hold (also expressed by an invariant).

Sections 5 and 6 illustrates the proposed approach on our example. The verification (Section 7) is performed on a finite state abstraction of the program, which models only the behavior of boolean variables. A verification tool, called LESAR, is available, which can apply two verification techniques:

- the former is an exhaustive enumeration of the states of the (abstraction of the) program, similar to standard model checking [12, 27].

- the later is a symbolic construction of the set of states which satisfy the property, analogous to "symbolic model checking" [10, 14, 15].

These techniques will be applied to the example introduced in Section 3.

Finally, Section 8 outlines a method for modular verification: taking advantage of the fact that the program and its properties are expressed in the same language, we can use the proved properties of a subprogram in the verification of a full program. This technique can reduce the complexity of the verification.

[3] This is a more powerful mechanism than the macro-notation offered by some specification languages [12, 29].

2 The language Lustre

We do not give here a detailed presentation of the language LUSTRE, which can be found elsewhere [11, 18]. We only recall the elements which are necessary for understanding the paper.

A LUSTRE program specifies a relation between input and output variables. Any variable or expression is intended to be a function of time. Time is assimilated to the set of natural numbers. Variables are defined by means of equations (one and only one equation for each variable which is not an input): As said before, an equation "`X=E`", where `E` is a LUSTRE expression, specifies that the variable `X` is always equal to `E`.

Expressions are made of variable identifiers, constants (considered as constant functions), usual arithmetic, boolean and conditional operators (considered as pointwisely applying to functions) and only two specific operators: the "previous" operator and the "followed-by" operator:

- If `E` is an expression denoting the function $\lambda n.e(n)$, then "`pre(E)`" is an expression denoting the function

  $$\lambda n.\begin{cases} nil & \text{if } n = 0 \\ e(n-1) & \text{if } n > 0 \end{cases}$$

  where *nil* is an undefined value.

- If `E` and `F` are two expressions of the same type, respectively denoting the functions $\lambda n.e(n)$ and $\lambda n.f(n)$, then "`E -> F`" is an expression denoting the function

  $$\lambda n.\begin{cases} e(n) & \text{if } n = 0 \\ f(n) & \text{if } n > 0 \end{cases}$$

In addition to equations, a LUSTRE program can contain *assertions*, of the form "`assert E`", where `E` is any boolean expression. This means that `E` is assumed to be always true during the execution of the program. For instance, "`assert not(ON and OFF)`" expresses that the input events `ON` and `OFF` never occur at the same time. Assertions were introduced in order to express some known properties of the environment, for optimization purposes. They will play an important role in program verification.

A LUSTRE program is structured into *nodes*: a node is a subprogram specifying a relation between its input and output parameters. This relation is expressed by an unordered set of equations and assertions, possibly involving local variables. Once declared, a node may be functionally used in any expression, as a basic operator.

For instance the following declaration defines a node, of general usage, which returns *true* whenever its boolean parameter raises from *false* to *true*:

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

Now, the expression `edge(not C)` is *true* whenever the variable `C` has a falling edge.

3 Example of program

Let us introduce an example adapted from a subway device. At each end of a subway line, a special "U-turn" section allows trains to switch from one track to the other, and to go back in the opposite direction (see Fig. 1). A U-turn section is composed of three tracks A, B, C, and a switch S. Assuming the entering track is A and the exiting track is C, trains switching from A to C must first wait for S to connect A with B, then transit on B and wait again for S to connect B with C before going back on C.

Considering that several trains move along the tracks and that the switch is not a safe device, two kinds of accidents may occur within the section:

- if several trains are allowed to access the section together, they may collide.
- if the switch is not well positioned, trains will derail.

It is therefore clear that controlling a U-turn section is a highly critical task. Any automatic U-turn section management system (later on called UMS) must both drive the switch and manage train movements along the section so as to avoid accidents.

Such a system is typically *reactive*: upon receipt of information about the switch status and about the train position inside the section, it should deliver positioning requests to the switch and access grants to trains. These four kinds of events can be modeled by the following signals:

- `ack_AB` and `ack_BC` indicate whether the switch actually connects A with B or B with C. If none of these signals is active, trains must not take the switch.
- `on_A`, `on_B` and `on_C` are three sensors, one on each track of the section. They are active as long as there is a train on the track they observe.
- `do_AB` and `do_BC` are the requests for the switch to connect A with B or B with C.
- `grant_access` and `grant_exit` are grants for trains to move along the section. They correspond to traffic lights. The first one will allow trains to access the section only if it is empty and if the switch connects A with B. The second one will allow trains to exit B only if the switch connects B with C.

An overview of the system and its environment is given in Fig. 2. Let us now implement the UMS in LUSTRE. In the sequel, it is assumed that initially, there is no train in the section. Let us define the equations for the switch positioning requests. The switch will be requested to connect A with B each time the section is empty and to connect B with C each time a train has arrived on B. These requests will remain active until the switch is in the right position. In LUSTRE, we directly get the equations:

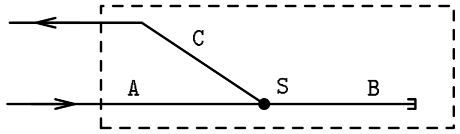

Figure 1: A subway U-turn section

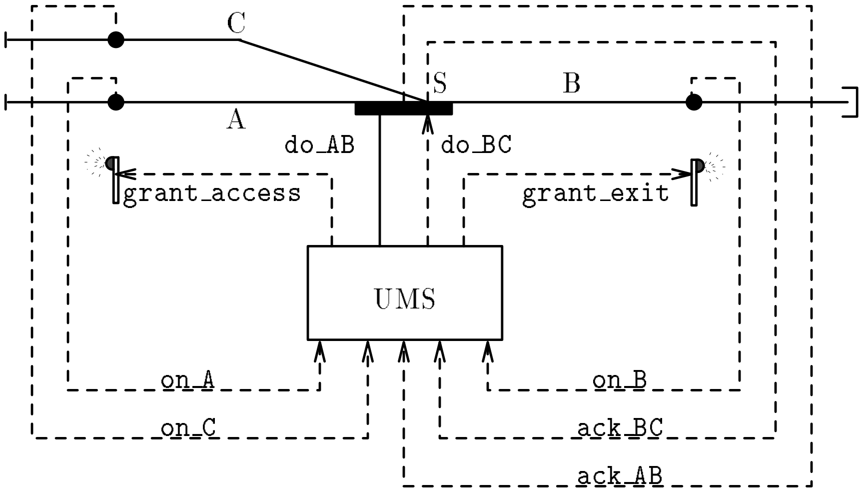

Figure 2: The UMS system and its environment

```
do_AB = not ack_AB and empty_section;
do_BC = not ack_BC and only_on_B;
```

Here, `empty_section` states that no train is in the section, while `only_on_B` states that the only trains in the section are on B. The equations for these variables are the followings:

```
empty_section = not(on_A or on_B or on_C);
only_on_B = on_B and not(on_A or on_C);
```

Now, we have to write the equations defining the movement grants. As mentioned before, access to the section will be granted only if it is empty and if the switch connects A with B. We therefore get the following equation:

```
grant_access = empty_section and ack_AB;
```

Trains will be granted to exit the section when the switch connects B with C and when the only trains in the section are on B. So, we get:

```
grant_exit = only_on_B and ack_BC;
```

We finally get the whole LUSTRE program for the UMS system, shown by Fig. 3. This simple example intends to show that LUSTRE is well-suited for programming such systems, as all the equations written in the program are straightforwardly deduced from the informal specifications of the U-turn section. The fact that equations can be written in any order encourages a progressive translation of the specifications: each requirement is expressed in turn, possibly involving the introduction of auxiliary variables. Notice that introducing an auxiliary variable (like `empty_section`), for naming an expression, has no influence on the program semantics, but can increase its readability.

```
node UMS(on_A,on_B,on_C,ack_AB,ack_BC: bool)
         returns (grant_access,grant_exit, do_AB,do_BC: bool);
var empty_section, only_on_B: bool;
let
   grant_access = empty_section and ack_AB;
   grant_exit = only_on_B and ack_BC;
   do_AB = not ack_AB and empty_section;
   do_BC = not ack_BC and only_on_B;
   empty_section = not(on_A or on_B or on_C);
   only_on_B = on_B and not(on_A or on_C);
tel
```

Figure 3: The LUSTRE program for the UMS system

4 Expressing critical properties

Let us now consider the expression of the critical properties of a program. Many formalisms have been proposed for that, most of them being inspired either from temporal logics or from process algebras. However, in order to reduce the user's effort, we have been looking for a formalism being as close as possible to the programming language. So, we propose to express a property as the invariance of some boolean LUSTRE expression. Having to express a property P about a program Π, we will write a boolean expression `B` such that P is satisfied if and only if `B` is always true during any execution of Π. It has been shown elsewhere [7] that LUSTRE can be viewed as a subset of linear temporal logic [25], and that any *safety property* can be expressed in that way. From our experience, the critical properties required for 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!

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

> *"Any occurrence of a critical situation causes an alarm, which must be sustained within a five seconds delay"*

Such a property relates three events: the occurrence of the critical situation, the alarm, and the deadline. A general pattern for this property is the following one:

> *"Any occurrence of the event A must cause the condition B to be true until the next occurrence of the 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 A has occurred in the past, either B has been continuously true, or C has occurred at least once, 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 `X` is always true if and only if the property holds:

```
node always_from_to(B,A,C: bool) returns (X: bool);
let
   X = implies(after(A), always_since(B,A) or once_since(C,A));
tel
```

The equation defining `X` uses four 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 `after` returns the value *false* until the first time its input is *true*. Then it returns *true* for ever:

  ```
  node after(A: bool) returns (afterA: bool);
  let
     afterA = false -> pre(A or afterA);
  tel
  ```

- The node `always_since` has two inputs and returns *true* if and only if its first input has been continuously *true* since the last time its second input was *true*:

  ```
  node always_since(B,A: bool) returns (alwaysBsinceA: bool);
  let
     alwaysBsinceA = if A then B
                     else if after(A) then B and pre(alwaysBsinceA)
                     else true;
  tel
  ```

- Finally, the node `once_since` has two inputs and returns *true* if and only if its first input has been at least once *true* since the last time its second input was *true*:

  ```
  node once_since(C,A: bool) returns (onceCsinceA: bool);
  let
     onceCsinceA = if A then C
                   else if after(A) then C or pre(onceCsinceA)
                   else true;
  tel
  ```

These nodes will be used in our example. Of course, other operators could be defined in the same way (see [19]).

5 Critical properties of the example

Let us express in LUSTRE the set of *safety* properties required from our U-turn management system. As mentioned before, this system must *always* avoid the occurrence of two dangerous situations.

The first one concerns train collisions. We have to check that a train may enter the section only if this one is empty. In LUSTRE, this property will be expressed by the invariance of a boolean variable `no_collision`, defined as follows:

```
no_collision = implies(grant_access,empty_section);
```

From the equation defining `grant_access`, this property is obviously true. So, assuming that no train enters the section by track C, it is possible to consider by now that no more than one train is in the section at any time.

We only have to check now that a train entering the section by track A will always leave it by track C, and that no derailment is possible. This leads to verify how the switch is driven. First, it is clear that the switch positioning requests should never be both active at the same time. This is simply expressed by the invariance of the following LUSTRE expression:

```
exclusive_req = not(do_AB and do_BC);
```

The switch should also connect A to B from the instant when a train is allowed to enter the section until it has arrived on track B. This leads to the following property:

```
no_derail_AB = always_from_to(ack_AB, grant_access, only_on_B);
```

Similarly, the switch should always connect B to C from the instant when a train is allowed to leave the section until it has actually left it:

```
no_derail_BC = always_from_to(ack_BC, grant_exit, empty_section);
```

Notice that if the above property is true, a train cannot leave the section by track A.

Finally, the global property to prove is expressed by the following LUSTRE equation:

```
property = no_collision and exclusive_req and no_derail_AB and no_derail_BC;
```

6 Modeling the environment

The next step in the verification process should consist in running the verification tool and check whether the safety properties above are preserved by the UMS. However, at that point, an important and crucial task is to provide a description of how the environment of the system behaves. Actually, the environment obeys some rules which restrict its possible behaviors. For instance, in a U-turn section, trains are assumed to stop when traffic lights are red. The verification tool would certainly not achieve checking the system without being aware of such an information. It is therefore necessary to define some assumptions about the behavior of the environment. In LUSTRE, this is done using the assertion mechanism. As said before, LUSTRE assertions express that some boolean expressions can be assumed to be always true.

Let us describe in that way some important features of the U-turn section environment. We can make the following assumptions about the switch:

- the switch cannot both connect A with B and B with C:

```
assert not(ack_AB and ack_BC);
```

- Once in a given position, the switch remains stable unless it is requested to move to the opposite position:

```
assert always_from_to(ack_AB,ack_AB,do_BC)
       and always_from_to(ack_BC,ack_BC,do_AB);
```

About train movements inside the section, we make the following assumptions:

- Initially, there is no train in the section.

```
assert empty_section -> true;
```

  (Remember that "`empty_section -> true`" is equal to "`empty_section`" at the initial instant, and then is true forever; so the above assertion only restricts the initial value of "`empty_section`")

- Trains obey traffic lights. So, if a train enters or leaves the section, then the corresponding traffic light was green at the instant before. Therefore, we get (using the node `edge` defined in Section 2):

```
assert true -> implies(edge(not empty_section), pre grant_access);
assert true -> implies(edge(on_C), pre grant_exit);
```

- When a train leaves A, it is on B. When a train leaves B, it is either on A or on C.

```
assert implies(edge(not on_A),on_B);
assert implies(edge(not on_B), on_A or on_C);
```

This example and our experience show that specifying the environment behavior of a program requires important efforts. Such a work undoubtly adds some complexity to the verification task. However, giving such a precise specification of the assumptions made on the environment behavior is certainly a useful task in designing a critical system. Moreover, notice that these assumptions can be dynamically checked, during the execution of the program (a compiler option produces the corresponding code). They can be used also in a testing phase, for choosing valid testcases.

7 Program verification

We consider now the verification problem. Given a program Π, a property P expressed by a boolean expression `B` which must be invariantly *true* under some assumptions expressed by an assertion `A`, we can build a new program Π' by putting together Π, the computation of `B` and the assertion `assert A`, considering the result of `B` to be the only output of Π' (see Fig. 4). The problem thus reduces to proving that the only boolean output of Π' is always *true* during any execution of the program which permanently satisfies the assertion `assert A`.

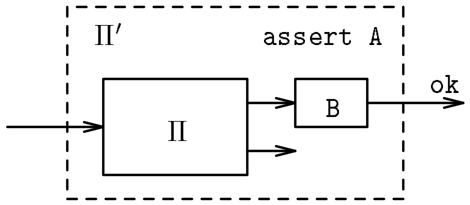

Figure 4: Building a verification program

The verification is performed on a finite state abstraction of the program. Any numerical computation is deliberately ignored, and boolean expressions depending on numerical variables (e.g., comparisons) are considered non deterministic. However, assertions can be used again to restrict this non determinism: For instance, if tests on conditions "`X<Y`", "`Y<Z`" and "`X<Z`" appear in the program, the assertion "`implies(X<Y and Y<Z, X<Z)`" will prevent the prover to consider absurd cases corresponding to "`X<Y<Z`" and "`X≥Z`".

So, we consider a purely boolean, non deterministic program Π'', which approximates Π' in the sense that "it has more behaviors": Any execution trace of Π' is also an execution trace of Π''. Therefore, if the output of Π'' is always *true* so is the output of Π' (Notice that, if the property depends on the values of numerical variables, our tool may fail in proving it). Now, since Π'' only contains boolean variables, it represents a finite state machine, on which any verification problem is decidable: proving that its output is always *true* amounts to enumerate its finite set of states, checking that, in each state — belonging to a path starting from the initial state and on which the assertions are always *true* — and for each input vector, the output evaluates to *true*. Two "verification engines" have been implemented and integrated into a verification tool, called LESAR:

- The first one explicitly enumerates the reachable states, as done by standard "model checkers" [12, 27]. The main limitation of such an approach is obviously the number of states that can be considered. The present version of the tool deals with programs of about 1,000,000 states in reasonable time (less than 1 hour).
- The second engine proceeds symbolically: starting from a boolean formula F_0, characterizing the set of states where the output is true (in LUSTRE this formula is the expression of the property), it iteratively computes a sequence $F_1, F_2, \ldots, F_n$ of formulas, where F_{i+1} characterizes the set of states, belonging to F_i and necessarily leading (in one execution step) into F_i. As soon as the initial state doesn't satisfy F_i, one can conclude that the property is not satisfied, since there exists an execution path leading to a state where the output is false. Otherwise, since the state space is finite, the sequence of formulas converges after a finite number of steps to a formula F which characterizes the set of states from which it is not possible to reach a state violating the property. Our tool performs symbolic computations over formulas using *binary decision diagrams* ("BDDs" [9]), a compact canonical encoding of boolean formulas. This approach is sometimes called "symbolic model checking" [10, 14, 15].

The two approaches are complementary: in some cases, the enumerative method is cheaper than the symbolic one, and conversely. The main limitation of the enumerative method is the number of reachable states which must be considered, whereas the symbolic method is limited by the complexity of boolean formulas (the size of the BDDs encoding it). Now, the complexity of

```
node UMS_verif(on_A,on_B,on_C, ack_AB,ack_BC: bool)
      returns(property: bool);
var
   grant_access,grant_exit: bool;
   do_AB,do_BC: bool;
   no_collision,exclusive_req: bool;
   no_derail_AB,no_derail_BC: bool;
   empty_section, only_on_B: bool;
let
   empty_section = not(on_A or on_B or on_C);
   only_on_B = on_B and not(on_A or on_C);

   -- ASSERTIONS
   assert not(ack_AB and ack_BC);
   assert always_from_to(ack_AB,ack_AB,do_BC)
           and always_from_to(ack_BC,ack_BC,do_AB);
   assert empty_section -> true;
   assert true -> implies(edge(not empty_section), pre grant_access);
   assert true -> implies(edge(on_C), pre grant_exit);
   assert implies(edge(not on_A),on_B);
   assert implies(edge(not on_B), on_A or on_C);

   -- UMS CALL
   (grant_access,grant_exit,do_AB,do_BC) =
              UMS(on_A,on_B,on_C,ack_AB,ack_BC);

   -- PROPERTIES
   no_collision = implies(grant_access,empty_section);
   exclusive_req = not(do_AB and do_BC);
   no_derail_AB = always_from_to(ack_AB, grant_access, only_on_B);
   no_derail_BC = always_from_to(ack_BC, grant_exit, empty_section);
   property = no_collision and exclusive_req and no_derail_AB
           and no_derail_BC;
tel
```

Figure 5: The verification program

a boolean formula is not related with the number of states it characterizes: The formula "true" can represent billions states!.

The program provided to LESAR for dealing with our example is listed in Fig. 5. The times (in sec., on SUN4) needed for proving its properties (separately and globally) using each method are gathered in the table below:

Property	States	Enum	Symb
`no_collision`	27	0.7	0.3
`exclusive_req`	27	0.7	0.4
`no_derail_AB`	37	1.0	0.5
`no_derail_BC`	39	1.0	0.5
`property`	54	1.2	1.3

Of course, the example is so simple that these results are hardly meaningful. However, we have treated more significant programs: In particular, the experience driven in [17] deals with a real nuclear plant control system; all the critical properties required of this system have been expressed (so, they were all safety properties) and verified using LESAR, in spite of the fact that this example involves a lot of real-valued variables. As a matter of fact, these variables only appeared in the properties by means of threshold comparisons, which have been handled using assertions.

8 Modular verification

The fact that program and properties are expressed in the same language, together with the assertion mechanism provide also a method for modular verification: Informally, given a program Π compound of n subprograms $\Pi_1, \Pi_2, \ldots, \Pi_n$, and once each Π_i has been proved to satisfy a safety property P_i ($i = 1 \ldots n$), one can prove that Π satisfies a safety property P by proving that the combination of the P_i implies P. More formally, if we note "$\Pi \models P$" the fact that the program Π satisfies the safety property P, and if $||$ denotes the parallel composition, we have

$$\frac{\Pi_1 \models P_1 \quad , \quad \Pi_2 \models P_2}{(\Pi_1 \mid\mid \Pi_2) \models P_1 \wedge P_2}$$

This can be done in LUSTRE in the following way: Let Π be a program using some node Π' (Fig. 6.a). Assume that Π' has been proved to satisfy a property P', expressed by the invariance

(a) (b)

Figure 6: Modular verification

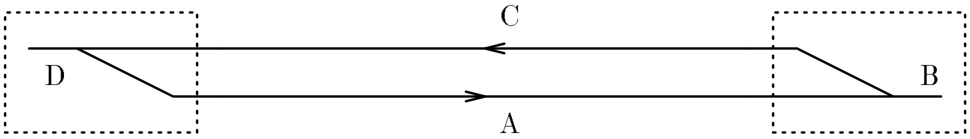

Figure 7: A full subway track

of the boolean expression `B`. Now, for proving that Π satisfies a property P, one can consider Π' as a part of the environment of Π, replacing it by the assertion that `B` is always true (Fig. 6.b). Formally, if we note "$\Box$`B`" the property that `B` is always *true*,

$$\frac{\Pi' \models \Box \mathtt{B} \quad , \quad (\Pi; \mathtt{assert\ B}) \models P}{(\Pi \,||\, \Pi') \models P}$$

Of course, the property P' (which must be found by intuition) may be too weak to allow the proof of P, but such a decomposition may drastically reduce the complexity of the verification.

For instance, the proved properties of the UMS system could be used for verifying the control of a whole subway track (Fig. 7): Informally, let $\mathtt{LINEAR\_TRACK}_X$ be a Lustre program controlling a single track X, and $\mathtt{UMS}_{X,Y}$ be the controller of a UMS with input track X and output track Y, the structure of the whole program would be

$$\mathtt{LINEAR\_TRACK}_A \,||\, \mathtt{UMS}_{A,C} \,||\, \mathtt{LINEAR\_TRACK}_C \,||\, \mathtt{UMS}_{C,A}$$

Instead of considering this whole program, one could try to verify the following (hopefully) simpler one

$$\mathtt{LINEAR\_TRACK}_A \,||\, \mathtt{assert\ B}_{A,C} \,||\, \mathtt{LINEAR\_TRACK}_C \,||\, \mathtt{assert\ B}_{C,A}$$

where $\mathtt{B}_{X,Y}$ denotes the boolean expression which has been proved to be invariantly true in the program $\mathtt{UMS}_{X,Y}$.

Moreover, this approach has been extended in [19] to allow the inductive verification of regular networks of identical processes. Modular verification also allows the verification of partially developed programs: Using the same approach in a slightly different way, you can verify the properties of a program Π before writing a sub-program Π', using a specification of Π'.

9 Conclusion and future work

We have tried to highlight the advantages of using a synchronous data-flow language in designing a real-time program. These advantages are twofold: On one hand, such a language meets traditional tools used in this field, and on the other hand, since it can be viewed as an executable temporal logic, it allows the expression of specifications and the smooth merging of programs and properties.

The synchronous hypothesis relegates "real-time" problems to the evaluation of program reaction time. Under this hypothesis, a program can be modeled by standard transition systems. As a consequence, our approach to program verification is quite standard, and even rather restricted, since it only deals with checking safety properties on finite state program abstractions. These restrictions have been introduced in order to tackle real-life problems, and we have argued that they meet many practical cases. Notice that, in contrast with many verification tools which deal with models of programs, our tools apply directly to programs themselves, thus meeting G. Berry's "WYPIWYE" principle [4]: *What you prove is what you execute*! There is no manual transformation between the program which is verified and the code which is executed.

Of course, both programming and verification rely on the synchronous hypothesis. This hypothesis must be checked *in fine* for the results to be valid. However, this checking only consists in evaluating the maximum execution time of linear pieces of code (the reactions), and is therefore easy.

This work will be pursued, at least in two directions:

- The restriction to boolean abstractions is a strong limitation. Dealing with some properties about bounded integers would certainly be worthwhile. In particular, integer variables are often used in LUSTRE to count *delays*. Up to now, our tools are not aware that a delay of 3 seconds is shorter that one of 10 seconds! Recent works on timed graph analysis [1, 16] could be adapted for that.

- A lot of work remains to be done around modular verification and program synthesis. In the approach proposed in §8, the specification of a subprogram must be provided by intuition. It would be appealing to synthesize automatically this specification, using our symbolic verification tool: Assume a program Π, calling a sub-program Π', is required to satisfy a property P, i.e., the invariance of some boolean expression `B`. We can first remove Π' from Π, considering it as an unknown part of Π's environment. If the resulting program satisfies P, then Π' has nothing to do with P. Otherwise, the symbolic verifier exhibits a formula F which characterizes the set of states of the program in which `B` is *false*. The role of Π' is then to avoid these states. The idea is to extract from F a specification of Π'. Moreover, since in our approach, specifications are programs, one can wonder if it is possible to synthesize Π' from this specification.

References

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

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

[3] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

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

[5] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems, an introduction to ESTEREL. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers.* Elsevier Science Publisher B.V. (North Holland), 1988. INRIA Report 647.

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

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

[8] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.

[10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, 1990.

[11] 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.

[12] 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.

[13] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proceedings of the IEEE*, 79(9):1283–1292, September 1991.

[14] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[15] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In R. Kurshan, editor, *International Workshop on Computer Aided Verification, Rutgers*, June 1990.

[16] D. L. Dill. Timing assumptions and verification of finite state concurrent systems. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*, Grenoble, June 1989. LNCS 407, Springer Verlag.

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

[18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[19] 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.

[20] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[21] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3), 1987.

[22] 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.

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

[24] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[25] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Conference on Logics of Programs*. LNCS 194, Springer Verlag, 1985.

[26] J. S. Ostroff. Automated verification of timed transition systems. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*, Grenoble, June 1989. LNCS 407, Springer Verlag.

[27] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*. LNCS 137, Springer Verlag, April 1982.

[28] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New Orleans, December 1991.

[29] 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.