Validation of Synchronous Reactive Systems: from Formal Verification to Automatic Testing*

Nicolas Halbwachs, Pascal Raymond
{Nicolas.Halbwachs,Pascal.Raymond}@imag.fr

Vérimag**, Grenoble – France

Abstract. This paper surveys the techniques and tools developped for the validation of reactive systems described in the synchronous data-flow language Lustre [HCRP91]. These techniques are based on the specification of safety properties, by means of *synchronous observers*. The model-checker Lesar [RHR91] takes a Lustre program, and two observers — respectively describing the expected properties of the program, and the assumptions about the system environment under which these properties are intended to hold —, and performs the verification on a finite state (Boolean) abstraction of the system. Recent work concerns extensions towards simple numerical aspects, which are ignored in the basic tool. Provided with the same kind of observers, the tool Lurette [RWNH98] is able to automatically generate test sequences satisfying the environment assumptions, and to run the test while checking the satisfaction of the specified properties.

1 Introduction

Synchronous languages [Hal93,BG92,LGLL91,HCRP91] have been proposed to design so-called "reactive systems", which are systems that maintain a permanent interaction with a physical environment. In this area, system reliability, and therefore design validation, are particularly important goals, since most reactive systems are safety critical. As a consequence, many validation tools have been proposed, which are dedicated to deal with systems described by means of synchronous languages. These tools either concern automatic verification [LDBL93,DR94,JPV95,Bou98,RHR91], formal proof [BCDP99], or program testing [BORZ98,RWNH98,MHMM95,Mar98].

As a matter of fact the validation of synchronous programs, on one hand raises specific problems — like taking into account known properties of the environment — and on the other hand allows the application of specific techniques — since the programs to be validated are *deterministic* systems with inputs, in contrast with classical concurrent processes, which are generally modelled as non-deterministic and closed systems. Both for formal verification and for testing, the user has to specify:

* This work was partially supported by the ESPRIT-LTR project "SYRF".
** Verimag is a joint laboratory of Université Joseph Fourier, CNRS and INPG associated with IMAG.

1. the intended behavior of the program under validation, which may be more or less precisely defined. In particular, it may consist of a set of properties, and, for the kind of considered systems, critical properties are most of the time *safety properties*.
2. the assumptions about the environment under which the properties specified in (1) are intended to hold. These assumptions are generally safety properties, too.

In synchronous programming, a convenient way of specifying such safety properties is to use "*synchronous observers*" [HLR93], which are programs observing the inputs and the outputs of the program under validation, and detect the violation of the property. Once these observers have been written, automatic validation tools can use them for

formal verification: One can verify, by model-checking, that for each input flow satisfying the assumption, the corresponding output flow satisfy the property. In general, this verification is performed on a finite-state abstraction of the program under verification.

automatic testing: The assumption observer is used to generate realistic test sequences, which are provided to the program; the property observer is used as an "oracle" determining whether each test sequence "passes" or "fails".

In this paper, we present these approaches in the context of the declarative language LUSTRE [HCRP91]. A model-checker for LUSTRE, called LESAR [RHR91], has been developped for long, and extended towards dealing with simple numerical properties. Two testing tools, LUTESS [BORZ98] and LURETTE [RWNH98] are also available; here, we focus on LURETTE, which has some numerical capabilities.

2 Synchronous Observers in LUSTRE

2.1 Overview of Lustre

Let us first recall, in a simplified way, the principles of the language LUSTRE: A LUSTRE program operates on *flows* of values. Any variable (or expression) x represents a flow, i.e., an infinite sequence $(x_0, x_1, \ldots, x_n, \ldots)$ of values. A program is intended to have a cyclic behavior, and x_n is the value of x at the nth cycle of the execution. A program computes output flows from input flows. Output (and possibly local) flows are defined by means of equations (in the mathematical sense), an equation "x=e" meaning "$\forall n, x_n = e_n$". So, an equation can be understood as a temporal invariant. LUSTRE operators operate globally on flows: for instance, "x+y" is the flow $(x_0+y_0, x_1+y_1, \ldots, x_n+y_n, \ldots)$. In addition to usual arithmetic, Boolean, conditional operators — extended pointwise to flows as just shown — we will consider only two temporal operators:

- the operator "pre" ("*previous*") gives access to the previous value of its argument: "pre(x)" is the flow $(nil, x_0, \ldots, x_{n-1}, \ldots)$, where the very first value "*nil*" is an undefined ("non initialized") value.

– the operator "->" ("followed by") is used to define initial values: "x -> y" is the flow $(x_0, y_1, \ldots, y_n, \ldots)$, initially equal to x, and then equal to y forever.

As a very simple example, the program shown below is a counter of "events": It takes as inputs two Boolean flows "evt" (true whenever the counted "event" occurs), and "reset" (true whenever the counter should be reinitialized), and returns the number of occurrences of "events" since the last "reset". Once declared, such a "node" can be used anywhere in a program, as a user-defined operator. For instance, our counter can be used to generate an event "minute" every 60 "second", by counting "second" modulo 60.

```
node Count(evt, reset: bool)
    returns(count: int);
let
    count = if (true -> reset) then 0
              else if evt then pre(count)+1
              else pre(count)
tel
```

```
mod60 = Count(second, pre(mod60=59));
minute = (mod60 = 0);
```

2.2 Synchronous Observers

Now, an observer in LUSTRE will be a node taking as inputs all the flows relevant to the safety property to be specified, and computing a single Boolean flow, say "ok", which is true as long as the observed flows satisfy the property.

For instance, let us write an observer checking that each occurrence of an event "danger" is followed by an "alarm" before the next occurrence of the event "deadline". It uses a local variable "wait", triggered by "danger" and reset by "alarm", and the property will be violated whenever "deadline" occurs when "wait" is on.

```
node Property(danger, alarm, deadline: bool)
    returns (ok: bool);
var wait: bool;
let
    wait = if alarm then false
              else if danger then true
              else (false -> pre(wait));
    ok = not(deadline and wait);
tel
```

Assume that the above property is intended to hold about a system S, computing "danger" and "alarm", while "deadline" comes from the environment. Obviously, except if S emits "alarm" simultaneously with each "danger", it cannot fulfill the property without any knowledge about "deadline". Now, assume we know that "deadline" never occurs earlier than two cycles after "danger".

```
node Assumption(danger, deadline: bool)
    returns (ok: bool);
let   ok = not deadline or
             (true -> pre(not danger and
                 (true -> pre(not danger))));
tel
```

This assumption can also be expressed by an observer.

```
(danger, alarm, ...) = S(deadline, ...);
realistic = Assumption(danger, deadline);
correct = Property(danger, alarm, deadline);
```

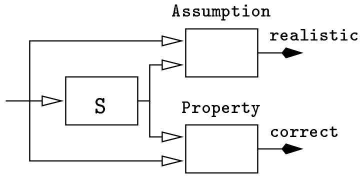


Fig. 1. Validation Program

2.3 Validation Program

Now we are left with 3 programs: the program S under validation, and its two observers, Property and Assumption. We can compose them in parallel, in a surrounding program called "Validation Program" (see Fig.1). Our verification problem comes down to showing that, whatever be the inputs to the validation program, either the output "correct" is always true, or the output "realistic" is sometimes false. The advantages of using synchronous observers for specification have been pointed out:

- there is no need to learn and use a different language for specifying than for programming.
- observers are *executable*; one can test them to get convinced that the specified properties are the desired ones.

Notice that synchronous observers are just a special case of the general technique [VW86] consisting in describing the negation of the property by an automaton (generally, a Büchi automaton), and showing, by performing a synchronous product of this automaton and the program, that no trace of the program is accepted by the automaton. The point is that, in synchronous languages, the synchronous product is the normal parallel composition, so this technique can be applied within the programming language.

3 Model-Checking

3.1 Lustre programs as state machines

Of course, a LUSTRE program can be viewed as a transition system. All operators, except pre and ->, are purely combinational, i.e., don't use the notion of *state*. The result of a -> operator depends on whether the execution is in its first cycle or not: let *init* be an auxiliary Boolean state variable, which is initially true, and then always false. The result of a pre operator is the value previously taken by its argument, so each pre operator has an associated state variable. All these state variables define the state of the program. Of course, programs that have only Boolean variables have finitely many states and can be fully verified by model-checking [QS82,CES86,BCM$^+$90,CBM89]: when the program under verification and both of its observers are purely Boolean, one can traverse the finite set of states of the validation program. Only states reached from the initial state without falsifying the output "realistic" are considered, and in each reached state, one check that, for each input, either "realistic" is false, or "correct" is true. This can be done either enumeratively (i.e., considering each state in turn) or symbolically, by considering sets of states as Boolean formulas.

3.2 Lustre programs as interpreted automata

Programs with numerical variables can be partially verified, using a similar approach. We consider such a program as an intepreted automaton: the states of the automaton are defined by the values of the Boolean state variables, as above. The associated interpretation deals with the numerical part: conditions and actions on numerical variables are associated with the transitions of the automaton. An example of such an interpreted automaton will be shown in Section 4. If it happens that a property can be proved on the (finite) control part of the automaton, then it is satisfied by the complete program. Otherwise, the result is unconclusive.

3.3 LESAR

LESAR is a verification tool dedicated to LUSTRE programs. It performs the kind of verification described above, by traversing the set of control states of a validation program, either enumeratively of symbolically. More precisely, it restricts its search to the part of the program that can influence the satisfaction of the property. This part, sometimes called the *cone of influence*, can be easily determined, because of the declarative nature of the language: all dependences between variables are explicit. This is an important feature, since experience shows that, in many practical cases, the addressed property only concerns a very small part of a program: in such a case, LESAR may be able to verify the property, even if the whole state space of the program could not be built.

4 Towards Numerical Properties

Only properties that depend only on the control part of the program can be verified by model checking. The reason is that LESAR can consider as reachable some control states that are in fact unreachable because of the numerical interpretation, which is ignored during the state space traversal: some transitions are considered feasible, while being forbidden by their numerical guards. Let us illustrate this phenomenon on a very simple example, extracted from a subway speed regulation system:

A train detects beacons placed along the track, and receives a signal broadcast each second by a central clock. Ideally, it should encounter one beacon each second, but, to avoid shaking, the regulation system applies a hysteresis as follows: let #b and #s be, respectively, the current numbers of encountered beacons

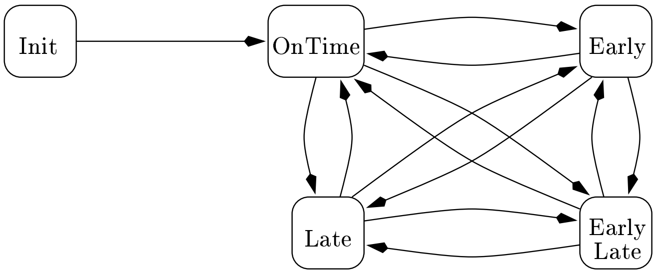


Fig. 2. Interpreted automaton of the subway example

and of elapsed seconds. Whenever $\#\mathsf{b} - \#\mathsf{s}$ becomes greater 10, the train is considered early, until $\#\mathsf{b} - \#\mathsf{s}$ becomes negative. Symmetrically, whenever $\#\mathsf{b} - \#\mathsf{s}$ becomes smaller than -10, the train is considered late, until $\#\mathsf{b} - \#\mathsf{s}$ becomes positive. We only consider the part of the system which determines whether the train is early of late. In LUSTRE, the corresponding program fragment could be:

```
diff =  0 -> if second and not beacon then pre(diff)-1
             else if beacon and not second then pre(diff)+1
             else pre(diff);
early = false -> if diff > 10 then true
                 else if diff < 0 then false
                 else pre(early);
late =  false -> if diff < -10 then true
                 else if diff > 0 then false
                 else pre(late);
```

This program has 3 Boolean state variables: the auxiliary variable *init* (initially true, and then false forever) and the variables storing the previous values of early and late. The corresponding interpreted automaton has the control structure shown by Fig 2, and, for instance, the transitions sourced in the state "OnTime" are guarded as follows:

g_1: $\mathsf{diff} > 10 \wedge \mathsf{diff} \geq -10 \rightarrow$ Early $\quad g_3$: $\mathsf{diff} > 10 \wedge \mathsf{diff} < -10 \rightarrow$ EarlyLate
g_2: $\mathsf{diff} \leq 10 \wedge \mathsf{diff} < -10 \rightarrow$ Late $\quad g_4$: $\mathsf{diff} \leq 10 \wedge \mathsf{diff} \geq -10 \rightarrow$ OnTime

Without any knowledge about numerical guards, the model-checker does not know that some of these guards (g_1 and g_2) can be simplified, nor that one of them (g_3) is unsatisfiable. This is why the state "EarlyLate" is considered reachable.

A transition the guard of which is numerically unsatisfiable will be called *statically unfeasible*. In our example, if we remove statically unfeasible transitions, we get the automaton of Fig. 3, where the state "EarlyLate" is no longer reachable. A simple way of improving the power of a model-checker is to provide

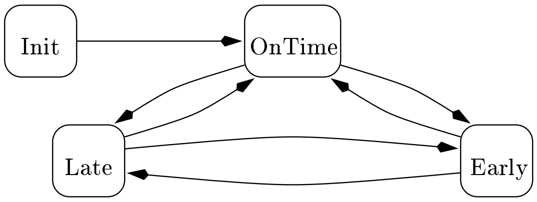


Fig. 3. The subway example without statically unfeasible transitions

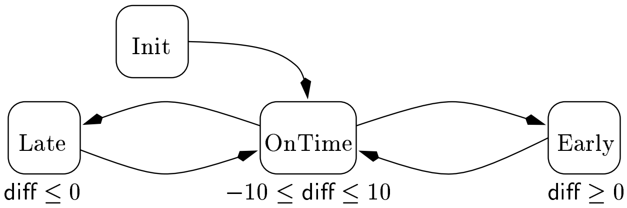


Fig. 4. The subway example without dynamically unfeasible transitions

it with the ability of detecting statically unfeasible transitions, in some simple cases. For instance, unfeasibility of guards made of *linear relations* is easy to decide[1].

This is why Lesar has been extended with such a decision procedure in linear algebra: when a state violating the property is reached by the standard model-checking algorithm, the tool can look, along the paths leading to this state, for transitions guarded by unfeasible linear guards. If all such "bad" paths can be cut, the "bad" state is no longer considered reachable. This very partial improvement significantly increases the number of practical cases where the verification succeeds.

Of course, we are not always able to detect statically unfeasible transitions. Moreover, some transitions are unfeasible because of the dynamic behavior of numerical variables. For instance, in the automaton of Fig. 3, there are direct transitions from state "Early" to state "Late" and conversely. Now, these transitions are clearly impossible, since diff varies of at most 1 at each cycle, and cannot jump from being ≥ 0 in state "Early" to becoming < -10 in state "Late". Such transitions are called *dynamically unfeasible*. Detecting dynamically unfeasible transitions is much more difficult. We experiment "linear relation analysis" [HPR97] — an application of abstract interpretation — to synthesize invariant linear relations in each state of the automaton. If the guard of a transition is not satisfiable within the invariant of its source state, then the transition

[1] at least for rational solutions; but since unfeasibility in rational numbers implies unfeasibility in integers, such an approximate decision is still conservative.

is unfeasible. In our example, we get the invariants shown in Fig. 4, which allow us to remove all unfeasible transitions.

5 Automatic Testing

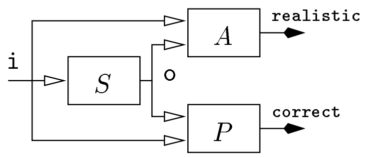
In spite of the progress of formal verification, testing is and will remain an important validation technique. On one hand, the verification of too complex systems — with too complex state space, or important numerical aspects — will remain unfeasible. On the other hand, some validation problems are out of the scope of formal verification: it is the case when parts of the program cannot be formally described, because they are unknown or written in low level languages; it is also the case when one wants to validate the final system within its actual environment. So, verification and testing should be considered as complementary techniques. Moreover, testing techniques and tools should be mainly devoted to cases where verification either fails or does not apply. This is why we are especially interested in techniques that cope with numerical systems, that don't need a formal description of the system under test (black box testing), and the cost of which doesn't depend on the internal complexity of the tested system.

Intensive testing requires automation, since producing huge test sets by hand is extremely expensive and error-prone. Now, it appears that the prerequisite for automatic generation of test sets is the same as for verification: an automatic tester will need a formal description of both the environment — to generate only realistic test cases — and the system under test — to provide an "oracle" deciding whether each test passes or fails. In section 2, we proposed the use of synchronous observers for these formal descriptions. In the Lurette [RWNH98] and Lutess [BORZ98] tools, such observers are used to automatically generate and run test sequences. In this section, we explain the principles of this generation.

The specific feature of reactive systems is, of course, that they run in closed loop with their environment. In particular, they are often intended to *control* their environment. This means that the current input (from the environment) may depend on the past outputs (from the system). In other words, the realism of an input sequence does not make sense independently of the corresponding output sequence, computed by the system under test. This is why, in our approach, test sequences are generated on the fly, as they are submitted to the system under test.

More precisely, we assume that the following components are available:

- an executable version of the system under test, say S. We only need to be able to run it, step by step.
- The observers A and P, respectively describing the assumptions about the environment and the properties to be checked during the test.



Moreover, the output "realistic" of the observer A is required *not* to depend instantaneously of the outputs "o" of S. Since "o" is supposed to be computed from the current input "i", it would be a kind of causality loop that the realism of "i" depend on "o".

Basically, the tester only needs to know the source code of the observer A, and to be able to run the system S and the observer P, step by step. It considers, first, the initial state of A: in this state, the LUSTRE code of A can be simplified, by replacing each expression "e_1 -> e_2" by "e_1", and each expression "pre(e)" by "*nil*". After this simplification, the result "realistic" is a combinational expression of the input "i", say "b(i)". The satisfaction of the Boolean formula b(i) can be viewed as a constraint on the initial inputs to the system. A constraint solver — which will be detailed below — is used to randomly select an input vector i_0 satisfying this constraint. Now, S is run for a step on i_0, producing the output vector o_0 (and changing its internal state). Knowing both i_0 and o_0, one can run A and P for a step, to make them change their internal state, and to get the oracle "correct" output by P. The LUSTRE code of A can be simplified according to its new state, providing a new constraint on "i". The same process can be repeated as long as the test passes (i.e., P returns "correct = *true*"), or for a given number of steps.

The considered tools mainly differ in the selection of input vectors satisfying a given constraint. In LUTESS [BORZ98], one consider only purely Boolean observers. A constraint is then a purely Boolean formula, which is represented by a Binary Decision Diagram. A correct selection corresponds to a path leading to a "true" leaf in this BDD. The tool is able to perform such a selection, either using an equiprobable strategy, or taking into account user-given directives. LURETTE [RWNH98] is able to solve constraints that are Boolean expressions involving Boolean inputs and *linear relations* on numerical inputs.

Example: Let us illustrate the generation process on a very simple example. Assume S is intended to regulate a physical value u, by constraining its second derivative. Initially, both u and its derivative are known to be 0. Then, the second derivative of u will be in an interval $[-\delta, +\delta]$ around the (previous) output x of S. An observer of this behavior can be written as follows:

```
node A (u, x: real) returns (realistic: bool);
var dudt, d2udt2: real;
let
    dudt = 0 ->(u – pre(u));
    d2udt2 = dudt – pre(dudt);
    realistic = (u=0) -> ((pre(x) – delta <= d2udt2)
                          and (d2udt2 <= pre(x) + delta));
tel
```

At the first cycle, the code of A is simplified to

```
dudt = 0; d2udt2 = nil; realistic = (u=0);
```

There is only one way of satisfying the constraint, by choosing $u_0 = 0$. The system S is run for one cycle, with this input value, let x_0 be the returned value. At the second cycle, we know that

$$\mathsf{pre(u)} = 0\ ,\ \mathsf{pre(dudt)} = 0\ ,\ \mathsf{pre(x)} = x_0$$

So the code of A is simplified to

```
dudt = u; d2udt2 = dudt;
realistic = (x0−delta <= d2udt2) and (d2udt2 <= xo+delta);
```

which gives the (linear) constraint $x_0 - \delta \leq u \leq x_0 + \delta$. Assume the value $u_1 = x_0 + \delta$ is selected, and provided to S, which returns some new value x_1. At the next cycle, we know that

$$\mathsf{pre(u)} = \mathsf{pre(dudt)} = x_0 + \delta\ ,\ \mathsf{pre(x)} = x_1$$

So, the code of A simplifies to

```
dudt = u − (x0+delta); d2udt2 = dudt − (x0+delta);
realistic = (x1−delta <= d2udt2) and (d2udt2 <= x1+delta)
```

which gives the constraint $x_1 + 2x_0 \leq u \leq x_1 + 2x_0 + 2\delta$, and so on...

6 Conclusion

We have presented some validation techniques, which mainly derive from the specification of properties by synchronous observers. While not being restricted to synchronous models, this way of specifying properties is especially natural and convenient in that context, since the same kind of language can be used to describe the system and its properties.

Our presentation was centered on the language LUSTRE, but the techniques could be adapted to any synchronous language. Notice, however, that some ideas were directly suggested by the declarative nature of LUSTRE. For instance, synchronous observers were a natural generalization of the *relations* in ESTEREL, which are a way of expressing known implications or exclusion between input events. When transposed into LUSTRE, these relations are just special cases of invariant Boolean expressions. Generalized to any Boolean LUSTRE expression, this mechanism provides a way of specifying any safety property. Also, in test sequence generation, the idea of considering an observer as a (dynamic) constraint is especially natural when the observer is written in LUSTRE, but can be adapted to any synchronous language.

References

[BCDP99] S. Bensalem, P. Caspi, C. Dumas, and C. Parent-Vigouroux. A methodology for proving control programs with Lustre and PVS. In *Dependable Computing for Critical Applications, DCCA-7, San Jose*. IEEE Computer Society, January 1999.

[BCM+90] 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.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BORZ98] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: testing environment for synchronous software. In *Tool support for System Specification Development and Verification*. Advances in Computing Science, Springer, 1998.

[Bou98] A. Bouali. Xeve: an Esterel verification environment. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

[CBM89] 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.

[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.

[DR94] R. De Simone and A. Ressouche. Compositional semantics of ESTEREL and verification by compositional reductions. In D. Dill, editor, *6th International Conference on Computer Aided Verification, CAV'94*, Stanford, June 1994. LNCS 818, Springer Verlag.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[HCRP91] 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.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[JPV95] L. J. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunication software. In P. Wolper, editor, *7th International Conference on Computer Aided Verification, CAV'95*, Liege (Belgium), July 1995. LNCS 939, Springer Verlag.

[LDBL93] M. Le Borgne, Bruno Dutertre, Albert Benveniste, and Paul Le Guernic. Dynamical systems over Galois fields. In *European Control Conference*, pages 2191–2196, Groningen, 1993.

[LGLL91] 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.

[Mar98] B. Marre. Test data selection for reactive synchronous software. In *Dagstuhl-Seminar-Report 223: Test Automation for Reactive Systems - Theory and Practice*, September 1998.

[MHMM95] M. Müllerburg, L. Holenderski, O. Maffeis, and M. Morley. Systematic testing and formal verification to validate reactive programs. *Software Quality Journal*, 4(4):287–307, 1995.

[QS82] 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.

[RHR91] 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.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science*, June 1986.