Statistical Testing of Synchronous Reactive Systems

Julien Fayolle, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Bruno Marre

Statistical Testing of Synchronous Reactive Systems

Julien Fayolle[1], Marie-Claude Gaudel[1], Sandrine-Dominique Gouraud[1], Bruno Marre[2]

1: LRI; Univ. Paris-Sud, CNRS ; Bât 490,91405 Orsay, France
2: LSL, CEA List Centre de Saclay, 91191 Gif sur Yvette Cedex, France

Abstract: The synchronous data-flow language Lustre is widely used to describe the behavior of reactive systems. Most of these systems are critical so they need intensive testing. Statistical testing allows intensive testing but generally misses special cases. We present a new approach enriching statistical testing with a coverage criteria of the Lustre description.

We have developed sALLUSTe, a software tool dedicated to the statistical testing of Lustre descriptions. sALLUSTe generates test scenarios by drawing uniformly at random paths in a structure related to the Lustre description. Our approach and the early results of sALLUSTe are presented here.

Keywords: Lustre, statistical testing, reactive systems

1. Introduction

This paper presents a new approach to intensive statistical testing of synchronous reactive systems that are specified in the data-flow language Lustre [14].
This approach combines some techniques and tools that already proved their interest in their own fields, namely: the GATeL tool for structural and functional testing based on Lustre descriptions, and the AuGuSTe tool for statistical testing of imperative programs.
Classically, *random testing* methods are based on some distribution on the input domain. They allow intensive and low-cost testing campaigns. But they lead to a poor coverage of particular cases.
Previous works [6,2] on biasing random testing of imperative programs toward the coverage of their control structure (under the name of *statistical testing*) have shown experimentally a very good detection power.
In this paper we study the transposition of such approaches to data-flow reactive programs, taking into account data dependencies and cyclic behaviors.
The new approach consists of 1) translating the Lustre description into a combinatorial structure specification, which represents all behaviors, 2) drawing uniformly at random an object within this structure. This object corresponds to a behavior, and 3) computing a sequence of inputs that ensures this behavior.
The second step exploits results and tools in combinatorics. The third step is based on constraint solving. A prototype called sALLUSTe has been developed. It reuses the front-end and the constraint solver of the GATeL tool.
The paper presents the method, the prototype, and a significant set of first experimental results. These results have been obtained on an academic example formerly used to evaluate GATeL which is representative of most Lustre descriptions, and on an industrial example. These first experiments give hints on how to improve the method and the tool for producing test sequences of greater length.

2. Lustre and testing methods for Lustre

Lustre is a synchronous data-flow language. A data-flow language describes the transformation of a data-flow, namely a sequence of inputs into another data-flow, the sequence of outputs. The dependencies between successive inputs and outputs is described by equations on the data-flows. These specificities are well-suited for the description of reactive systems. In this section we first recall some basics about the Lustre syntax. Then we sketch the specific testing methods for Lustre that have influenced our work.

2.1 Lustre

A Lustre description is structured into nodes. In this paper, we consider only one node but the approach can be generalized to several nodes.

Each node is defined by a set of equations and an equation denotes a flow. An equation is an expression that may be a value, a variable, a conditional expression (if…then…else), a temporal operator (pre, when, current, ->) or any classical operator (and, or, +, -, etc.).

Here is the example of a node called *oven*. It takes three Boolean data-flows and one integer data-flow (*duration*) in input and outputs a Boolean data-flow (*cooking*).

```
node oven (start, abort, open: bool; duration: int)
returns (cooking: bool);
var
  running: bool;
```

```
let
  running = if (open or abort)
          then false
          else if start
              then true
              else (false -> pre(running));

  cooking = running and (duration >0);

tel;
```

2.2 Testing methods for Lustre

Several methods and tools have been proposed to test synchronous reactive systems. We describe some of them adapted to the Lustre language: Lutess [12], Lurette [4], Lustructu [3], GATeL [1,9] or the statistical testing method described in [13].

Lutess, developed by LSR-Imag (Grenoble, France), and Lurette, developed by Verimag (Grenoble, France), consider both synchronous reactive systems as black-boxes. Input sequences are generated from Lustre descriptions of the environment properties. Then, outputs are checked against an oracle: the state of the system (values of each flow) at each cycle should satisfy the safety properties. Lutess only deals with Boolean data flows but allows an elaborated environment i.e., operational profiles, or behavioral pattern. Lutess also allows the design of tests from safety properties in order to test the fault detection power of the system under test. Lurette handles Boolean and numerical data flows. The environment constraints can be described by the Lucky language which allows the generation of tests from non-deterministic environments.

Lustructu, also developed by LSR-Imag, is not exactly a test generation tool. Its goal consists in evaluating the structural coverage of a Lustre description i.e., the quality of a data test set. In order to measure the structural coverage, the authors defined some structural coverage criteria based on the operator network. Given a Lustre description, a coverage criteria and a data test set, Lustructu determines the structural coverage after a symbolic computation of path activation conditions.
Like Lutess, Lustructu only deals with Boolean data flows.

2.3 GATeL

Given a reactive program and two (partial) descriptions of its behavior and of its environment as Lustre models, the main role of GATeL [1,9] is to automatically generate test sequences according to user-oriented test cases. These test sequences can then be submitted to the program under test. In both cases where the program has been automatically generated from the model or not (depending on the development process followed), GATeL also provides a basis for an automatic oracle (expected outputs).

The model of the environment is intended to filter out from all the possible behaviors those corresponding to realistic reactions, decreasing the state space to be explored.

Requested test cases can then be finely characterized in order to exercise meaningful situations. Test cases selection is a crucial part of the testing process. Several approaches have been proposed to automate it, but none of them is universally recognized. On the contrary, GATeL provides the user the means to define his/her own selection strategies. The first step on this direction is the definition of a test objective. The test objective states some important expected properties of the program under test to be checked. It can be either invariant properties or reachability properties. Invariant properties are stated with *assert* directives. The properties that must be satisfied in at least one cycle (in fact, in the last cycle of sequences built by GATeL) are stated by *reach* directives. To build a sequence reaching the test objective according to the Lustre model of the program and its environment, these three elements are automatically translated into a constraint system. A randomized resolution procedure then solves this system.

The random aspect of this resolution procedure implies that the input domains are not fairly covered, thus quite distinct sequences may be generated for the same objective (for instance different ways to raise an alarm). A second step in the definition of a selection strategy is to help GATeL to distinguish these sequences. This can be achieved by splitting the constraint system so that each sub-system characterizes a particular class of behaviors reaching the objective. This splitting can be processed interactively by applying predefined decompositions of Boolean-numerical-temporal operators in the Lustre expressions corresponding to the current constraint system. This interactive splitting process provides the user a way to define a tunable structural coverage of the model. A second way to proceed is to state declaratively the various behaviors one wants to observe through a dedicated selectable directive *split Var with [Cond_1...Cond_n]*. When selected (depending on the visibility of the attached variable), the constraint system is split into *n* sub-systems with each one corresponding to the assignment of one condition to true. These two techniques allow the user to finely tune the kind of selection strategy needed.

Finally, test submission consists in reading input sequences generated with GATeL, computing program outputs, then comparing these values to the expected ones evaluated during the generation procedure. When the program has not been automatically generated from the Lustre model, this gives an automatic oracle. On the other case, the truth value of the test objective can play the role of a partial oracle. GATeL is still under development concerning methodological and efficiency aspects. However, it has been successfully experimented on industrial case studies.

3. Coverage-guided random testing for imperative languages

The first statistical testing method guided by coverage criteria was proposed by Thévenod-Fosse and Waeselynck in [6]. In 2004, Denise, Gaudel, and Gouraud [2] introduced a new method along the same line for the random generation of tests for C programs. The control graph of a C program is represented as a combinatorial structure. A path in the control graph represents a possible execution of the program. Drawing paths randomly within a combinatorial structure (here the control graph) is an active field of research and has produced highly efficient algorithms [10, 11]. Paths within the combinatorial structure are drawn and the testing method uses constraint-solving to find the inputs which execution leads to this path. For any path drawn within the structure, there may be no input which execution is represented by this path: The path is said to be infeasible and this is a general problem to software testing. A further advantage of the method is that the combinatorial decomposition makes possible the generation of inputs aiming to satisfy a coverage criteria on the structure.

This method was the basis for the tool AuGuSTe. AuGuSTe takes a C program as input (currently a program written in a subset of C) and generates at random as many test inputs as asked by the user in order to cover a coverage criteria with some quality [7].

As in the structural statistical testing method proposed by [6], the method is able to draw some interesting test suites that classical random testing would probably not. It also enables the quantification of the coverage ensured by the test suites.

4. A new approach for testing Lustre descriptions

The new approach we present here is the transposition of the one introduced by Denise, Gaudel, and Gouraud in 2004 to the synchronous reactive language Lustre, taking into account data dependencies and temporal behaviors. In this section we detail the brute-force method and explain it on an example. Several optimizations of the method are presented in the next section.

4.1 General scheme

The approach first consists in preprocessing a Lustre description to represent it as a combinatorial structure specification. This structure is related to the decomposition of the Lustre operators into sub-cases. Secondly paths are drawn uniformly at random within this structure. Each edge on a path holds a label. This label is a predicate (called FLA, standing for *feasible local atom*) on the data-flows. The conjunction of all the local predicates along the drawn path is also a (large) predicate. Thirdly a resolution step is performed (using constraint-solving techniques) on the large predicate which may lead to a test scenario.

In the following no difference is made between the edge drawn and the label (FLA) it holds.

4.2 Brute-Force method

Preprocessing During the first step a Lustre node is translated in a combinatorial structure specification. Several flows are defined within a Lustre node. The output value of a flow depends on the values of “definitional” flows (i.e., flows used in the definition of the output flow) and on the operators used in the definition. Hence the output value of a flow constrains the definitional flows. If there are only two ouput values for the flow (for instance in the Boolean case) then there are two large sets of constraints: the constraints on the definitional flows that lead to a **true** output and those leading to a **false** output. We resort to the *unfolding* technique as used in Loft [8] to further decompose the sets of constraints. It leads to a larger number of sets of constraints but this allows the identification of far more interesting behaviors. Furthermore, it seems more likely that all flows satisfying constraints from a given set have the same fault detection power. The unfolding is done on the Lustre operators (the level of the decomposition is parametrizable by the user). The example subsection above provides an example of the unfolding technique.

Unfolding is one part of the preprocessing. Another is the construction of FLAs by constraint-solving: Once the sets of constraints for each flow have been identified, the conjunction of these sets for all the

flows defined within the Lustre node is built. This creates a large number of sets of (larger) constraints. Then the constraints of each of these sets are solved and the sets which constraints are not locally satisfiable are removed. This resolution is done by constraint-solving. In the following, the remaining sets of constraints are viewed as predicates (the conjunction of each of the constraints in the set leads to a predicate) and correspond to the FLA (*feasible local atom*).

The Lustre initialization operator **init** (also noted ->) is also unfolded and it creates two classes of cycles: initial and non-initial. We formally define the sets FLA-Init and FLA-NI as the sets of FLAs for the initial cycle and for any non-initial cycle. Suppose there are *j* FLAs at the initial cycle and *k* for any non-initial cycle, then

$$\text{FLA-Init}:=\{\text{FLA-Init}_1, ..., \text{FLA-Init}_j\} \text{ and } \text{FLA-NI}:=\{\text{FLA-NI}_1, ..., \text{FLA-NI}_k\}. \quad [1]$$

The length *n* of the test scenario (number of cycles) is specified by the user. The structure is called the *tree of FLAs*. It represents the set of sequences of FLAs of length *n* (sequences starting with a FLA from FLA-Init and where any other FLA in the sequence is from a non-initial cycle). Thus the tree is a way of representing the set

$$\text{FLA-Init} \times (\text{FLA-NI})^{(n-1).} \quad [2]$$

Drawing and resolution Our method generates uniformly a path (i.e., a sequence of FLAs) of the given length among all sequences of the same length. To each edge on the path is associated a label which is a FLA. The conjunction of all FLAs along the path is a large predicate. The constraint solver from GATeL then checks whether the large predicate is satisfiable, i.e., if there exists a sequence of inputs satisfying the predicate.

Example We illustrate the brute-force method with the example of the simple microwave oven described in section 2.

The unfolding refines the sets of constraints for the constraint-solving step. For example, the operator > is (default setting) split into three sub-cases: (duration>0) is true, (duration=0) is true, and (duration<0) is true. The operator *and* is unfolded in four cases: A and B leads to (A and B), (not(A) and B), (A and not(B)), and (not(A) and not(B)). The operator *or* is unfolded in four cases: (A and B), (not(A) and B), (A and not(B)), and (not(A) and not(B)). The operator *if then else* is unfolded in two sub-cases: *cond* is true and *cond* is false.

The flow *cooking* has two definitional flows: *running* and *duration*. The unfolding of operators *and* and > lead to six predicates:
co1:={running=true, duration>0, cooking=true},
co2:={running=true, duration=0, cooking=false},
co3:={running=true, duration<0, cooking=false},
co4:={running=false duration>0, cooking=false},
co5:={running=false, duration=0, cooking=false},
co6:={running=false, duration<0, cooking=false}.

At the initial cycle there are 6 cases for the flow cooking and also 6 cases at any non-initial cycle. At the initial cycle there are 5 cases for the flow running and also 5 at any non-initial cycle. Hence there are 30 conjunction predicates for the initial cycle and also 30 for any non-initial cycle. Of these 30 predicates, 18 are locally satisfiable after constraint solving i.e., there are 18 FLAs (and 15 FLAs for the initial cycle).

5. Optimizations

The brute-force approach is simple in its conception but suffers some drawbacks: a large number of paths have to be drawn in order to find satisfiable ones, the resolution of long predicates is costly, and the re-drawing of paths does not take advantage of previously drawn path. The resolution of the predicate associated to a path may be unsatisfiable, meaning that there is no test scenario which satisfies the predicate. This problem of infeasible path is general to software testing. We have developed optimizations of the brute-force method that reduce the number of drawn paths that are unfeasible. Several other optimizations have been developed: memorizing the infeasible paths and rejecting infeasible paths earlier in the resolution step.

5.1 Optimizations for the generation

Real-time systems have the general behavior of having little memory i.e., the values of the flows at a given cycle do not depend too far back on previous cycles. Hence the temporal dependencies between flows are of short time frame. We preprocess all predicates on a given (short) time frame by checking them for satisfiability. In the brute-force method, this time frame is one cycle, we extend it to 2 and 3 cycles. This preprocessing eliminates a *large* number of unfeasible predicates, and consequently this limits the possibility of drawing an infeasible path.

Pairs 2-by-2 FLAs (only in the non-initial case) consist in choosing two non-initial FLAs, defining their conjunction as a new predicate, and checking

the satisfiability of this predicate. The predicates that are not unsatisfiable are kept. Then a tree of 2-by-2 FLAs is built on the same pattern as the tree of FLAs.

The construction of pairs of FLAs is costly nevertheless it removes a lot of local insatisfiability and the pre-processing is done once.

Suppose the FLA “co4xru3”, the conjunction of the predicate co4 and a predicate for the flow running is drawn (this is indeed a FLA since the conjunction of the two predicates is satisfiable) at the non-initial cycle k and the FLA “co2xru2” at the cycle k+1 . Both FLAs are feasible (by definition of the FLA) nevertheless their conjunction is not satisfiable: At cycle k, **running** is false and at cycle k+1, **running** has the value of the flow at the previous cycle (since **running=pre(running)** in ru2) but furthermore **running** is false (in co2).

- Cycle k : co4:={running=false, duration>0, cooking=false};
- Cycle k : ru3:={open=false, abort=true, running=false};
- Cycle k+1 : co2:={running=true, duration<0, cooking=false};
- Cycle k+1 : ru2:={open=false, abort=false, start=false, running=pre(running)}.

To avoid these kind of insatisfiabilities on a two-cycle time frame, all the pairs of FLAs on consecutive cycles are preprocessed and insatisfiable pairs of FLAs on this time frame are removed. The remaining predicates are the 2-by-2 FLAs.

Triples The 3-by-3 FLAs are built on the same pattern as 2-by-2 FLAs but the time frame is 3 cycles. A triple of FLAs is composed and checked for satisfiabilty in a preprocessing step. If the predicate is satisfiable, it is kept as a 3-by-3 FLA. Then a structure based on these triples is constructed and a path is drawn within this structure.

Building the set of triples of FLA is extremely time- and memory-consuming and in most cases it does not remove many insatisfiabilities (most were already removed by the 2-by-2 FLAs).

5.2 Optimizations for the resolution

The brute-force method utilizes the naïve resolution mode: A path of the required length is drawn during the generation step then the predicate associated to the path is checked for satisfiability. This is costly since the checked predicate is long. (Its length is related to the number of cycles and to the number of flows defined within the node.) We offer two possible optimized modes for the resolution: *reject* and *incremental.* Reject means checking for satisfiability predicates related to the prefix of the full path incrementally until either a prefix is in the database of rejected prefixes, a prefix is unsatisfiable, or the full path predicate has been checked. Incremental means checking for satisfiability predicates related to the prefix of the full path incrementally, until either a prefix is unsatisfiable, or the full-length predicate has been checked.

Reject A path (which length is given by the user) is generated uniformly among all paths. The predicate associated to the full-length path is built incrementally as the conjunction of the predicate built on the prefix of the path and the predicate associated to the current edge. A *database of rejected prefixes* is built alongside the resolution. At first the database is empty. The database is queried to check whether the first predicate has already been rejected. If the query gives no answer then the current prefix predicate is checked for satisfiability. If the predicate fails the satisfiability check then it is added to the database of rejected prefixes. Otherwise the predicate is prolonged (a new FLA, or 2-by-2 FLA, or 3-by-3 FLA is added), the database is queried for the new predicate and so on until a path of the required length has been found satisfiable.

With this mode, there is no need to store the full predicate in memory. The length of the predicate is the number of cycles asked for by the user, hence the storing can be cumbersome. The database reject is fast since it is a simple look-up.

Incremental A path (which length is given by the user) is generated uniformly among all paths. The predicate associated to the full-length path is built incrementally as the conjunction of the predicate built on the prefix of the path and the predicate associated to the current edge. The incremental resolution mode *incrementally* checks the satisfiability of the predicate i.e., it checks the satisfiability alongside the predicate construction. The method first checks the satisfiability of the predicate associated to the first edge (i.e., the label associated to the edge drawn for the initial cycle), if the predicate is satisfiable, then the predicate for the second edge (corresponding to cycle 1) is added to the path predicate and checked for satisfiability, and so on. If, at some cycle, an unsatisfiability is detected, the resolution is abandoned and a new path is generated. The process stops when the full-length predicate is successfully checked for satisfiability.

In the incremental resolution mode, the length of the predicate under consideration is increasing incrementally from 1 to the given number of cycles.

Hence the first predicates under consideration are of small sizes allowing efficient satisfiability checking, unlike the naïve mode. If the satisfiability check is unsuccessful at one cycle then the longer predicates are not even looked at, avoiding the dealing with longer predicates.

5.3 Optimization for re-drawing

The brute-force method's mode for re-drawing is reset. At one point the resolution of a predicate may fail (the predicate corresponds either to the full-length path, or to a prefix of the path). Two modes to re-draw another path are available: either (*reset* mode), a new path is drawn from scratch (i.e., from the initial cycle), or (*backtrack* mode) one backtracks to the longest satisfiable predicate, then the generation and resolution steps continue from this "healthy" prefix.

The reset mode is slower since there is a new path to generate each time the resolution of a predicate fails. But the generation of paths is uniform among all paths from the structure and it gives the ability to quantify the coverage of the set of test scenarios. The backtrack mode takes advantage of the previous draws since it does not need to re-draw a satisfiable predicate. Hence the generation of the desired number of test scenarios is faster, nevertheless the uniformity of the drawing and thus the possibility of quantifying the coverage are lost.

6. sALLUSTe

sALLUSTe is an implementation of the method explained in the previous two sections. It has been developed by Bruno Marre and Julien Fayolle, using the logical programming language Prolog. The constraint-solving step is performed by a module from GATeL. Our tool generates test scenarios according to some strategy defined by the user. The user may also specify the unfolding of the Lustre operators.

The user provides sALLUSTe a Lustre description, along with the length of the scenarios and their numbers. The following optional modes presented in the previous sections are available:

- Generation of the path: FLAs, 2-by-2 FLAs,or 3-by-3 FLAs;
- Resolution: naïve, incremental, or reject;
- Re-drawing: backtrack or reset.

The generation mode is set by the flag UPT (**U**nique FLA, **P**air of FLAs, or **T**riple of FLAs). The value zero corresponds to FLAs, value 1 to 2-by-2 FLAs, and value 2 to 3-by-3 FLAs. The resolution mode is set by the flag NIR (Naïve, Incremental, or Reject). The value zero corresponds to naïve, value 1 to incremental, and value 2 to reject. The backtrack/reset mode is set to backtrack if the value of the parameter is zero, to reset if the value is one.

7. Experimental results

Experiments have been run on two examples : firstly the description of a microwave oven with an environment and secondly a description provided by MBDA, an industrial partner. The benchmarks have been run on GNU/Linux 2.6.20-16 with an Intel Pentium IV 2.6 Ghz (512 MB of RAM). For each of the four tables, the CPU time is averaged on 5 experiments.

The Lustre description of the microwave oven is terse (63 lines of code and 4 flows), but possesses a variety of Lustre operators and a large number of temporal dependencies. Nevertheless the number of FLA is limited (5 in the initial case and 13 in the non-initial cases). This microwave is the same as the one in Marre and Blanc [9] (see their Appendix for the full description).

Table 1 shows the CPU time taken by sALLUSTe to generate 1000 test scenarios using the brute-force method. Obtaining 1000 test scenarios of length 10 already takes about 13 hours. Hence the brute-force method is prohibitive even for small lengths.

Table 1

Length	*CPU time*
5	130.21
10	47638

Table 2 shows the CPU time taken for the generation of 1000 test scenarios using the modes reject, FLA 2-by-2, and reset. The preprocessing step, i.e., computing the FLAs 2-by-2, is of little influence in itself, but this optimization it provides large gains on the costly resolution step.

Table 2

Length	*CPU time*
5	4.24
10	8.9
15	13.7
20	18.66
25	23.66
30	28.8
35	33.5
40	39.8
45	45.27
50	50.2
100	112.5
150	195.7
200	295.6

The Lustre description of the second example has 388 lines of code and 19 flows are defined within the node. There are 74 FLAs at the initial node and 744 at the non-initial nodes (for the standard unfolding of operators).

Table 3 shows the time taken by sALLUSTe to generate a single test scenario of length 4 to 6. The brute method is used i.e., naïve resolution, AFLs and reset.

Table 3

Length	*CPU time*
4	4.2
5	21.15
6	324.7

It can be noted that the computing time increases very fast and obtaining even a single test scenario of length 6 takes roughly 5 minutes. Once again the brute force method is prohibitive even for test scenarios on a small number of cycles.

The test generation using the above-mentioned optimizations is far more satisfactory. Table 4 sums up the results for the generation of 100 test scenarios of length varying from 5 to 20. The method uses the building of FLAs 2-by-2, the resolution mode is reject and redrawing mode is reset.

For this example, the computation of the FLAs 2-by-2, and 3-by-3 is lengthy (roughly 10 minutes to compute 57,196 FLAs 2-by-2 from 553,536 couples of FLA) given the large number of FLAs. Nevertheless this preprocessing step is done once for the description and can be stored. The time in the table measure the generation, resolution and reset steps but not the preprocessing.

The preprocessing reduces the combinatorial size of the problem by 90%. It is possible to generate 100 test scenarios of length 20 in about 10 minutes. A scenario of length 20 already exhibits some interesting behaviors to the tester.

Table 4

Length	*CPU time*
5	5.47
10	26.85
15	130.
20	584.2

Overall these benchmarks indicate that the brute-force method is ineffective. The backtrack mode exhibits faster generation times but the uniformity on the drawing is lost, and hence the ability to quantify the coverage of the structure. Nevertheless with the optimizations we devised, sALLUSTe is able to generate test scenarios in large number and of expressive length, in reasonable time.

8. Conclusion

Several improvements on the method are on-going. First of all a refinement of the combinatorial structure representing the Lustre description is under consideration (currently it is the trivial concatenation of FLAs, 2-by-2 FLAs, or 3-by-3 FLAs). We also study the use of some learning process from the detected insatisfiabilities (currently there is a database of failures). An intelligent backtracking strategy keeping the drawing uniform is considered. Finally we investigate the use of program slicing techniques in our approach since it may limit the number of FLAs.

In this paper we do not consider the Lustre temporal operators *when* and *current*, nor do we deal with multiple clocks. This has to be taken care of in later works.

The approach we devised is dedicated to Lustre descriptions. Nevertheless, we are considering the extension of the principle to others synchronous languages like Esterel or Signal.

9. Acknowledgements

This work is part of the Software Factory project (www.usine-logicielle.org) of the Cluster System@tic Paris-Région.
The authors thank P. Baufreton from Hispano-Suiza, and D. Pariente from Dassault Aviation for their on-going evaluation of sALLUSTe and their feedback. We also thank J.-S. Cruz from MBDA for providing us a Lustre description (the one used in this article) and M. Nakhlé and J. Courtilleux, both from C-S, for sending us a very interesting Lustre description to work on.

10. References

[1] Bruno Marre, Agnès Arnould: *Test Sequence Generation from Lustre Descriptions: GATeL*. In Fifteenth International Conference on Automated Software Engineering, Grenoble, 2000.

[2] Alain Denise, Marie-Claude Gaudel, and Sandrine-Dominique Gouraud: *A Generic Method for Statistical Testing,* In ISRRE 2004.

[3] Abdesselam Lakehal and Ioannis Parissis: *Lustructu* : *A tool for the automatic coverage assessment of LUSTRE programs*. In Sixteenth IEEE International Symposium on Software Reliability Engineering, pages 301–310, 2005.

[4] Pascal Raymond, Daniel Weber, Xavier Nicollin, and Nicolas Halbwachs: *Automatic testing of reactive systems*. In Nineteenth IEEE Real-Time Systems Symposium, 1998.

[5] Lydie du Bousquet and Nicolas Zuanon: *An Overview of Lutess, A Specification-based Tool for Testing Synchronous Software*. In 14th IEEE International Conference on Automated Software Engineering, pages 208-215, 1999.

[6] Pascale Thévenod-Fosse and Hélène Waeselynck. *An investigation of software statistical testing*. The Journal of Software Testing, Verification and Reliability, 1(2):5–26, july-september 1991.

[7] Sandrine-Dominique Gouraud. *Utilisation des Structures Combinatoires pour le Test.* Thèse de doctorat. Université Paris-Sud. Juin 2004.

[8] Bruno Marre. *Toward automatic test data set selection using algebraic specifications and logic programming.* ICLP'91 Eighth International Conference on Logic Programming, pages 25-28, MIT Press, 1991.

[9] Bruno Marre, Benjamin Blanc: *Test Selection Strategies for Lustre Description in GATeL.* Electronic Notes in Theoretical Computer Science 111 (2005) 93—111.

[10] Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. *A calculus for the random generation of labelled combinatorial structures.* Theoretical Computer Science. 132:1-35, 1994.

[11] Albert Nijenhuis and Herbert Wilf. *Combinatorial Algorithms.* Academic Press, Harcourt, Brace, and Jovanovich, 1975.

[12] Ioannis Parissis and Farid Ouabdesselam. *Specification-based testing of synchronous software.* ACM SIGSOFT Fourth Symposium on the Foundation of Software Engineering, 1996.

[13] Guillaume Lussier and Helene Waeselynck, *Deriving test sets from partial proofs*, Fifteenth Int. Symposium on Software Reliability Engineering (ISSRE'04), pages 14-24, November 2004.

[14] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Didier Pilaud. *The synchronous Data Flow Programming Language Lustre.* Proceedings of the IEEE, 79(9) :1305-1320, september 1991.