Software Statistical Testing Based on Structural and Functional Criteria

P. Thévenod-Fosse, H. Waeselynck
LAAS – CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4 - FRANCE

Tel. +(33) 5 61 33 62 37
Fax +(33) 5 61 33 64 11
E-mail: {thevenod, waeselyn}@laas.fr

Abstract

Statistical testing is based on a probabilistic generation of test patterns: structural or functional criteria serve as guides for defining an input profile and a test size. The method is intended to compensate for the imperfect connection of criteria with software faults, and should not be confused with random testing, a blind approach that uses a uniform profile over the input domain. After a brief description of the approach, the paper is focused on experimental results involving safety-critical software from various application domains: avionics, civil and military nuclear field. In most of the experiments, mutation analysis was used to assess the error detection power of various test patterns. First, we give results related to procedural programs: they show the best effectiveness of statistical testing in comparison to deterministic and random testing. Yet, a limitation of the statistical patterns experimented on was their lack of adequacy with respect to faults related to extremal/special cases, thus justifying the use of a mixed test strategy involving both statistical and deterministic test sets. Such a mixed strategy has been defined for synchronous data flow programs. It may be applied at either the unit or integration testing levels and has been experimented with Lustre programs.

Keywords: Software testing, probabilistic generation, industrial programs, experiments.

1 Introduction

Testing involves exercising the software by supplying it with input values. Since exhaustive testing is not tractable, the tester is faced with the problem of selecting a subset of the input domain that is well-suited for revealing the (unknown) faults. The selection is guided by *test criteria* that relate either to a model of the program structure or a model of its functionality, and that specify a set of elements to be exercised during testing (see e.g. [1]). For example, the control flow graph is a classical structural model, and branch testing is an example of a criterion related to this model. Given a criterion, the methods for generating test inputs proceed according to one of two principles: either deterministic or probabilistic.

The **deterministic principle** consists in selecting a priori a set of test inputs such that each element is exercised at least once; and this set is most often built so that each element is exercised *only once*, in order to minimize the test size. But a major *limitation* is due to the imperfect connection of the criteria with the real faults. Because of the lack of an accurate model for software design faults, this problem is not likely to be solved soon. Exercising only once, or very few times, each element defined by such imperfect criteria is not enough to ensure a high fault exposure power.

To make an attempt at improving current testing techniques, one can cope with imperfect criteria and compensate their weakness by requiring that *each element be exercised several times*. This involves larger sets of test inputs that may be tedious to derive manually; hence the need for an automatic generation of test inputs. This is the motivation of **statistical testing** designed according to a criterion: combine the information provided by imperfect criteria with a practical way of producing numerous input patterns, that is, a random generation. In this approach, the probability distribution from which the test inputs are randomly drawn is derived from the criterion retained. Then it may have little connection with actual usage (i.e. operational profile): the focus is bug-finding, not reliability assessment. Also, the approach should not be confused with blind random testing, which uses a uniform profile over the input domain [5].

After a brief description of statistical testing in Section 2, the paper will be focused on experimental results involving safety-critical software from various application domains. Section 3 is devoted to results for procedural programs: they show the best effectiveness of statistical testing in comparison to deterministic and random testing. Yet, a limitation of the statistical patterns experimented on is their lack of adequacy with respect to faults related to extremal/special cases, thus justifying the use of a mixed test strategy involving both statistical and deterministic test sets. In Section 4, such a mixed strategy is implemented for synchronous data flow programs at unit and integration testing levels. Section 5 presents some concluding remarks, and introduces on-going work addressing the statistical testing of object oriented programs.

2 Statistical testing: principle and method

Statistical testing is based on an unusual definition of random testing, with the purpose of removing the blind feature of the conventional probabilistic generation. It aims at providing a "balanced" coverage of a model of the target software, no part of the model being seldom or never exercised during testing. As in the case of deterministic testing, test criteria may be related to a model of either the program structure, which defines statistical *structural* testing, or of its functionality, which defines statistical *functional* testing. In both cases, any criterion specifies a set of elements to be exercised during testing. Given a criterion C, let S_C be the corresponding set of elements. To comply with finite test sets, S_C must contain a finite number of elements that can be exercised by at least one input pattern. For example, the structural criterion "All-Branches" requires that each program branch be executed: C = "Branches" $\Rightarrow$ S_C = {executable program edges}.

The statistical test sets are defined by two parameters, which have to be determined according to the test criterion retained: (i) the test profile, or input distribution, from which the inputs are randomly drawn and, (ii) the test size, or equivalently the number of inputs (i.e. of program executions) that are generated.

The determination of the **test profile** is the corner stone of the method. The aim is to search for an input probability distribution that is proper to rapidly exercise each element defined by the criterion. In particular, the profile must accommodate the highest possible P_c value, where P_c is the occurrence probability per execution of the *least likely* element of S_c. Depending on the complexity of this optimization problem, the determination of the profile may proceed either in an analytical (see e.g. [12]) or an empirical [13] way. The first way supposes that the activation conditions of the elements can be expressed as a function of the input parameters: then their probabilities of occurrence are a function of the input probabilities, facilitating the derivation of a profile that maximizes the frequency of the least likely element. The empirical way consists in instrumenting the software in order to collect statistics on the numbers of activation of the elements: starting from a large number of inputs drawn from an initial distribution (e.g. the uniform one), the test profile is progressively refined until the frequency of each element is deemed sufficiently high.

Then the **test size N** must be large enough to ensure that the least likely element is exercised several times under the test profile inferred from the previous step. The notion of test quality q_N provides us with a theoretical framework to assess a minimum test size, using Relation (1) which can be explained as follows: $(1-P_c)^N$ is an upper bound of the probability of never exercising some element during N executions with random inputs. Then, for a required upper bound of $1-q_N$, where the target test quality q_N will be typically taken close to 1.0, a minimum test size is derived.

$$N_{min} = \ln(1-q_N) / \ln(1-P_c) \qquad (1)$$

It is worth noting that Relation (1) establishes a link between q_N and the expected number of times, denoted n, the least likely element is exercised: $n \cong -\ln(1-q_N)$. For example, $n \cong 9$ for $q_N = 0.9999$.

Returning to the imperfect connection of the criteria with real faults, it must be understood that the criterion does not influence random inputs generation in the same way as in the deterministic approach: it serves as a guide for defining an input profile and a test size, but does not allow for the a priori selection of a (small) subset of input patterns. The efficiency of the probabilistic approach relies on the assumption that the information supplied by the criterion retained is relevant to derive a test profile that enhances the program failure probability. The experimental results shown in the next sections support this assumption.

3 Experimental results for procedural programs

Statistical testing was first investigated for classical procedural programs written in Pascal, C or assembly language, coming from various critical application domains: avionics, civil and military nuclear field. Depending on the complexity of the program, from tens to thousands lines of code, the design of statistical test sets was based either on structural (Section 3.1) or functional

(Section 3.2) criteria. As a general rule, we adopted the following approach to assess the error detection power of a given test selection method: design several different test input sets according to this method, so that the reproducibility of results from one set to the other can be studied. In most cases, the effectiveness of test sets was assessed using mutation analysis [4]: it has been shown in [3] that, for procedural programs, simple syntactic mutations have the ability to produce complex run-time errors representative of the ones that would have been caused by real faults.

3.1 Statistical structural testing

Referring to classical control-flow and data-flow criteria (see e.g. [10]), statistical structural testing was investigated on small software components (about one hundred lines of code or less). Mutation analysis was used to compare the error detection power of statistical structural test sets with the one of: 1) deterministic test sets designed according to the same structural criteria; 2) random test sets drawn from a uniform profile; 3) actual test sets performed by the industrial companies (manual selection of functional test inputs, supplemented by additional ones to achieve a target structural coverage).

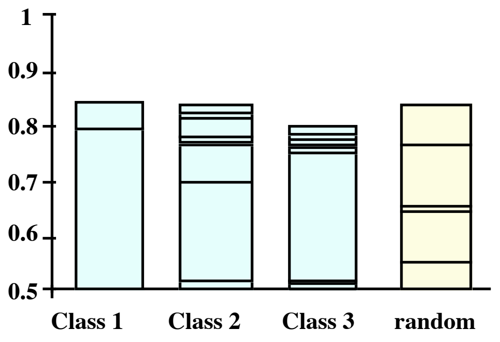
The results were in favor of statistical structural testing for the following reasons:

- Random uniform testing turned out to be a poor strategy in most cases, except for some very simple software components.
- There was no empirical evidence that deterministic structural test sets were more effective than pure random ones, and so whatever the stringency of the adopted criterion: this confirms the tricky link between these test criteria and the faults they aim to track down.
- On the contrary, high mutation scores were repeatedly observed for statistical structural testing, even in the case of weak criteria: this approach allowed us to increase significantly the failure probability of programs, even as regards subtle faults loosely connected with the criteria.
- The comparison with industrial test sets, which could be performed for some software units coming from different companies, showed the best efficiency of statistical testing. In some cases, the industrial test sets were as effective as the statistical ones, but longer. In the other cases, the industrial test sets were shorter but supplied a lower mutation score than the statistical test sets truncated to the same length.

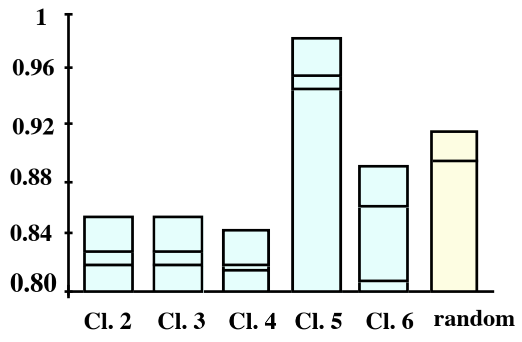
These various issues are more thoroughly commented on below, using examples extracted from our experimental results (Fig. 1 to 3). Figures 1 and 2 refer to an experimentation with safety-critical software units coming from the nuclear field. Additional material concerning these units may be found in [12]. Figure 3 refers to unpublished (confidential) experiments involving other software units from critical applications.

		Uniform profile	Structural profiles weak criteria	Structural profiles stringent criteria
Functions with simple "uniform" structure	FCT1	N = 170: 100%	—	N = 170: 100%
	FCT2	N = 80: 100%	—	N = 80: 100%
Functions with "non uniform" structure	FCT3	N = 405: 55.8 – 83.8%	N = 152: 97.6 – 98.8%	N = 405: 100%
	FCT4	N = 850: 89.6 – 91.7%	N = 42: 97.4 – 99%	N = 850: 99 – 99.1%

Fig. 1. Mutation scores for 4 unit functions – comparison of uniform and structural profiles

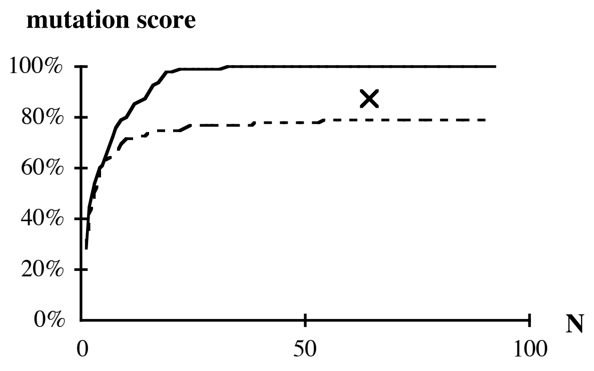


(a) `FCT3`, 1416 mutations

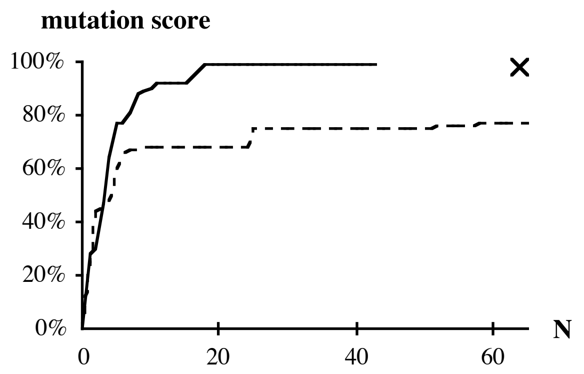


(b) `FCT4`, 587 mutations

Fig. 2. Functions with "non uniform" structure – mutation scores of deterministic structural test sets.



(a) `Prog1`, 6150 mutations



(b) `Prog2`, 118 mutations

Fig. 3. Comparison with industrial test sets

———	**statistical structural set**
- - - -	**uniform random set**
×	**industrial set**

Inadequacy of random uniform testing

In [9], random uniform testing was experimented on five small programs. The author noticed that the approach was surprisingly effective for some programs but very ineffective for others, and related the effectiveness to the degree of code coverage achieved by the random inputs. Our results corroborate his observations and clearly show that this testing method is likely to be a poor methodology, in most cases.

An example of this may be found in Figure 1, where the uniform sets have the same size as the statistical sets designed from stringent criteria. For `FCT1` and `FCT2`, the uniform profiles provide a 100% paths coverage of the programs: all the mutants are killed, which can be deemed very cost-effective if one considers that the test input generation requires little effort. This effectiveness is likely to be due to the comparative simplicity of both units. Indeed, for these units, the statistical structural profiles designed from the All-Paths criterion were close to the uniform ones. But *the performance of uniform testing falls off heavily as soon as the program's structure no longer lends itself to a uniform stimulation*. This appears strikingly in the case of `FCT3`, detailed in Figure 2a: hundreds of mutations were not uncovered by the 5 random uniform sets generated. Besides, the results vary from one set to the other: from 229 (16.2%) up to 626 (44.2%) seeded faults are not revealed depending on the set. And real disparities are observed between the subsets of faults uncovered: 66 faults revealed by the least efficient set are not revealed by the most efficient one.

To investigate the behavior of random test patterns, a detailed analysis of the evolution of the mutation score as a function of the number of test inputs was conducted. Whatever the test set and the program, the evolution of the growth of the number of faults revealed was quite similar (see e.g. Fig. 3): the test patterns rapidly uncover the faults during a first phase; then the incremental gain exhibits a sharp slowing down, that is, the slope of the growth becomes almost null. As a result, little improvement is expected if the tests are further pursued, unless a very large number of extra inputs is generated. This suggests that *the inadequacy of the uniform profiles may not be compensated by a reasonable increase of the test sizes:* it is unlikely that uniform test patterns will exhibit a good efficiency, since generally they properly exercise neither the functionalities of a program, nor its structure. Revealing input patterns being unlikely to be uniformly distributed over the input domain, a uniform profile is not relevant to increase the program failure probability during testing.

Limitation of deterministic structural testing

As mentioned at the beginning of the paper, the weakness of deterministic testing is likely to be due to the tricky link between the test criteria and the faults they aim to track down. As regards *structural* criteria, this limitation was confirmed by our experimental results.

Figure 1 shows that a blind testing approach is not adequate for functions `FCT3` and `FCT4`. Then, it may be deemed that using structural information to guide the selection of test inputs would allow us to significantly improve the test results. However, it can be seen in Figure 2 that such is not the case when the structural test sets are designed according to the deterministic principle (covering once each element defined by the criterion retained). Figure 2 displays the proportion of seeded faults revealed by the deterministic structural testing of FCT3 and FCT4 and its comparison with random uniform testing. A column identifies a class of test experiments: the horizontal lines stacked in the column give the scores of the various test sets that have been designed according to the same criterion. For FCT3, Classes 1 to 3 correspond respectively to the structural criteria All-Paths, All-Uses and All-Defs; for FCT4, All-Paths testing being not feasible, Classes 2 to 6 correspond to All-Uses, All-C-Uses, All-Defs, All-P-Uses and Branches.

As can be seen in the figure, *given a function and a criterion, the test sets exhibit scores that may be quite unrelated*: for FCT3, the scores of the Class 2 sets range from 0.52 to 0.833. Even when the range is narrower, it does not mean that the test sets reveal the same mutations: both Class 1 sets provide similar scores (0.79 versus 0.84), but 101 faults found by the least efficient set are not found by the most efficient one. Worse, the FCT4 results show that *the most stringent criteria do not necessarily supply the highest scores*: Class 2 (All-Uses) subsumes all other criterion classes, but the best results are supplied by the Class 5 experiments (All-P-Uses). Finally, there is no empirical evidence that deterministic structural inputs are more effective than pure random ones.

The previous results may be analyzed from a more general perspective: the loose connection of the test criteria with software design faults. Although a structural fault model (mutations) was used for our experiments, many seeded faults were loosely connected to structural criteria, and managed to induce complex behavior schemes of error masking, sequential erroneous behavior, or intermittent failures. Then deterministic structural testing, and in fact deterministic testing as a whole, suffers from the fact that it involves the selective choice of a small number of test inputs, which may or may not turn out to be adequate. No guarantee is provided in regard to fault exposure, even if a stringent criterion is adopted.

High error detection power of statistical testing

All our experimental results consistently showed that statistical testing was a practical way to compensate the imperfection of structural criteria.

Let us go back to the example of Figure 1, focusing on the results obtained for FCT3 and FCT4. In each case, two different profiles were designed. First, the most stringent achievable criteria were adopted: All-Paths for FCT3, All-Uses for FCT4; and the proper profiles were defined analytically. Then weaker criteria were also investigated, respectively All-Defs (Class 3 of FCT3) and Branches (Class 6 of FCT4), leading other profiles to be derived. It is worth noting that the Class 3 profile of FCT3 let two paths have a null probability of being executed.

Five different test sets were generated according to each profile, taking a target test quality of $q_N = 0.9999$. Figure 1 shows that high scores were *repeatedly* observed, whatever the particular set generated according to a same structural profile: the failure probability under the structural profiles was significantly higher than the one under a uniform profile. Moreover, the comparison with the results supplied by deterministic test sets designed from the same criteria shows that statistical testing was effective even as regards subtle faults loosely connected with those criteria.

FCT4 was the only case for which no statistical set supplied a score of 100%: 6 faults lead to failure only for some extremal input values. Under both structural profiles, to reach a probability 0.9 of generating such values would require more than 600,000 test inputs. This result complies with the decreasing of the incremental gain also observed for uniform test sets: mutants not killed at the end of a test set are difficult to track down because they induce a low failure probability under the corresponding profile. In the case of the statistical structural sets, the final stabilized scores are high, but the remaining mutants show the need for a *mixed test strategy* combining: (1) a global probe by statistical testing and (2) separate coverage of extremal values.

17 faults seeded in FCT3 were never revealed by any Class 3 statistical set: they affect the paths that have a null probability under this profile. As regards the 1399 other faults, the results were quite satisfactory (98.8% – 100% revealed). The Class 6 sets of FCT4 were also effective: they revealed 98.5% – 100% of the faults not related to extremal values. Hence, for the first step of the mixed test strategy, the most cost-effective approach seems to *retain weak criteria* facilitating the search for a test profile, and to *require a high test quality* (0.9999) with respect to them. But one must be careful the designed profile does not exclude items of the input domain (as for FCT3).

Comparison with industrial practice

The efficiency of statistical testing must also be assessed in comparison with industrial practice. Indeed, two objections could be raised:

- Given a criterion, statistical testing requires larger test sets than the deterministic approach. For example, let us recall than a test quality of $q_N = 0.9999$ implies that the least likely element of the criterion is exercised 9 times on average.
- The comparison with deterministic testing involved test sets built only from structural criteria, whereas industrial practice typically consists in building deterministic test sets from both structural and functional criteria.

The comparison with industrial test sets was performed for some critical software units coming from different companies. The industrial sets were designed by manual selection of functional test inputs, supplemented by additional ones to achieve a target structural coverage. Mutation analysis gave two categories of results, exemplified by Figures 3a and 3b. The most common case was the one observed for program Prog1: the industrial sets were shorter but supplied a lower mutation score than the statistical test sets truncated to the same length. Surprisingly, a second category of results was also observed in some cases, like in the case of program Prog2: the industrial test sets were indeed larger than the statistical ones, while supplying similar high mutation scores. Both categories of results need to be further confirmed by experimentation, but they bode well with regard to the efficiency of statistical structural testing.

To conclude on this set of experiments, it is worth recalling that the statistical structural testing approaches that were carried out are mainly intended for small pieces of software at a unit level. For larger pieces of software, higher level functional approaches have to be investigated.

3.2 Statistical functional testing

Statistical functional testing makes use of black box information on the program under test to derive a test profile and a test size: the aim is to ensure that the software functions, and their interactions, are properly scanned. Since it must be possible to study the influence of the input profile on the supplied coverage, our probabilistic approach requires a behavioral specification that is sufficiently formal to allow precise functional criteria to be defined. Fortunately, many software development environments provide technical support for behavioral modeling. For example, CASE tools implementing SA/RT techniques provide support for modeling with finite state machines and decision tables. Such models may serve as a basis for the probabilistic generation of test patterns.

As far as complex components are concerned, a detailed specification analysis is likely to determine a large number of functions that may not be described by a single model of reasonable complexity. Then, a top-down modeling process is usually undertaken, thereby involving a hierarchy of models. Given a hierarchy of models, the design of statistical testing must induce an analysis of reasonable complexity. The general guidelines outlined below aim at managing this complexity.

For a hierarchical specification, the detailed analysis of the models according to stringent criteria is presumably intractable, thus compelling us to use only weak criteria (e.g. state coverage in cases of finite state machines). Furthermore, even if weak criteria are adopted, it would not be realistic to attempt intensive coverage of all the models at the same time. This is because of:

- the *component complexity*, which makes the assessment of the element probabilities difficult as numerous correlated factors are involved;
- the *explosion of the test size;* even if the assessments are feasible in order to derive an input distribution, a prohibitive test size will probably be required to reach a high test quality, as the probability of the least likely element remains very low due to the large number of elements.

Hence, retaining *weak criteria* and deriving *several input profiles* – each one being focused on a subset of models – is the general approach that allows us to manage complexity. The approach has been exemplified by an industrial case study for which two different behavioral modeling techniques were investigated. The design of statistical test sets from the corresponding models, and their effectiveness with respect to both real faults and mutations, are presented below.

Case study: design of statistical functional test sets

Statistical functional testing was investigated for a *safety-critical software component from the nuclear field* (see e.g. [13]). The component is extracted from a nuclear reactor safety shutdown system, and belongs to the part of the system that periodically scans the position of the 19 reactor's control rods. Its implementation involves about thousand lines of C code, including the four functions `FCTi` mentioned in Figures 1 and 2. The design of statistical testing was based on two sets of behavior models deduced from the component specification:

- One description combining finite state machines and decision tables, defining a hierarchy of three functional levels. The retained criteria were state coverage for the finite state machines and rule coverage for the decision tables.
- Another description involving two levels of functionality, developed with the aid of the STATEMATE™ tool. The behavioral models supported by the tool are statecharts, a graphical language that improve state diagrams by adding the notion of state hierarchy, orthogonality and broadcast communication [7]. A retained criterion was the coverage of the basic states of each statechart (that is, states having no offspring). Also, it was required to probe a particular interaction between high and low level functions: the reset of low-level functions due to a change in operational mode (coverage of all possible low-level basic states immediately following the high-level transitions inducing a reset).

In both cases, it was not possible to derive a single test profile for the coverage of all models of the description: each of the corresponding test sets involved inputs drawn from two different profiles.

In the case of the first 3-level description, the search for the input profiles proceeded as follows. First, the set of equations governing state or rule activation was derived from the target subset of models. Then, it was noticed that the input configurations required for the balanced coverage of the topmost functional level clashed with the ones required by the two lowest levels of the hierarchy; hence the decision to derive two different input profiles. An approximation of the optimal profiles was computed using a special purpose program. Under these two profiles, 85 + 356 = 441 test inputs were required for the coverage of the various levels with a test quality of $q_N = 0.9999$.

In the case of the second description, it was also decided to design two input profiles, one for each functional level. But we had to proceed empirically: in the present state of the art, it is not possible to determine the equations governing the coverage of statecharts. We used the simulation and instrumentation facilities provided by STATEMATE: the statecharts were supplied with large input sets, whose profiles were progressively refined according to measures of coverage collected on the models. In order to compute a test size for the resulting profiles, we generated a sample of test sets having the same length as previously (85 + 356 inputs). It turned out that each basic state was exercised at least 11 times by any set, hence providing the target test quality of 0.9999.

Case study: error detection power of functional statistical test sets

Five different test sets were generated according to each of the cases described above: five input sets (denoted F-Sets) corresponding to the first probabilistic analysis, and five input sets (denoted ST-Sets) corresponding to the empirical analysis conducted in the STATEMATE environment. For the purpose of comparison, one uniform test set (denoted U-Set) was also generated. It contains

™ STATEMATE is a registered trademark of i-Logix, Inc.

5300 inputs drawn from a uniform profile without any preliminary analysis: this is in conformity with the foundation of random testing, that is, large test sets generated cheaply.

Two versions of the component were available: the real one, and a student version developed from the same specification. The experimental results, summarized in Figure 4, involved both genuine (unknown) faults (see e.g. [13]) and mutations [15] in each versions. Mutation analysis allowed us to experiment with a larger sample of faults than the 13 ones found by testing: 12 faults (denoted A, B, ..., L) in the student version, and one minor residual fault (denoted Z) in the real version. Faults A, G and J are *structural faults* directly linked to the coding activity. Faults B to F, and I, result from some *misunderstanding of the requirements* by the student. The others are *initialization faults*: either an improper initial value is assigned (K, L) or the initialization is missing (H, Z). It is worth noting that in the real system fault Z would never lead to failure due to systematic hardware compensation.

Once again, the uniform distribution turned out not to be an efficient profile for generating random test patterns. For example, only 5 genuine faults were revealed by the U-Set; and these 5 faults were all revealed within the first 633 executions, the remaining 4667 executions being garbage. A more detailed analysis of the 5300 uniform inputs showed that they poorly probe each version of the component from both a structural and a functional viewpoint.

✔	always revealed
i	revealed by i sets out of 5
—	not revealed

	Genuine faults													Mutation scores	
	A	B	C	D	E	F	G	H	I	J	K	L	Z	student version 2419 mutants	real version 3756 mutants
U-set (N = 5300)	✔	—	—	✔	—	✔	✔	✔	—	—	—	—	—	0.743	0.582
F-Sets (N = 441)	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	—	4	0.994 - 0.997	0.945 - 0.959
ST-Sets (N = 441)	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	0.998 - 1.0	0.930 - 0.961

Fig. 4. Experimental results of functional test sets

The F-Sets and ST-Sets performed repeatedly much better than the uniform one, although they are one order of magnitude shorter: the functional analysis allowed us to significantly increase the efficiency of the generated inputs. This is all the more satisfactory as weak criteria had to be retained to manage complexity. Also, the good results support the adequacy of deriving several profiles from a given multi-level description: whatever the modeling technique, the subsets of 85 and 356 inputs exhibited complementary features with respect to the exposure of both genuine and seeded faults; moreover, each entire set exercised 100% instructions of both versions of the component.

The fault Z was not revealed by one F-Set: this motivated our choice to stress the reset interactions in the ST-Sets. Then a new fault, L, was uncovered in the student program. This fault is rather subtle, because its exposure depends on specific conditions involving *five* successive data acquisition after a reset. During the design of the ST-profiles, only the *first* acquisition was taken into account for the coverage of the initial states; still, L was exposed by all ST-Sets. This result confirms the effectiveness of statistical testing with respect to faults that are loosely connected with the criteria retained. However, in this specific case, mutation analysis shows that the reset is not sufficiently probed. Whatever the F-Set or ST-Set, and whatever the program version, most or all mutants remaining alive are related to the initialization process, in spite of the fact that this process was implemented quite differently in each version. In our mind, testing the initialization process is a typical case where deterministic testing is more appropriate: this process is well-defined and it involves a limited number of possible input patterns.

3.3 First conclusions for procedural programs

The results reported in this section shows the high error detection power of statistical testing for procedural programs, referring either to structural or functional criteria. However, a limitation of the statistical sets experimented on was their lack of adequacy with respect to faults related to extremal/special cases. Such faults induce, in essence, a very low probability of failure under the profiles defined to ensure a *global* probe of the target programs: they require test inputs specifically aimed at them. Hence, the idea of a **mixed test strategy** involving both statistical and deterministic test sets.

Such a mixed strategy has been successfully implemented for an application from the spatial domain, not detailed in this paper. Statistical functional test sets were derived from the SA/RT specification of an R&D prototype, and supplemented by deterministic inputs covering extremal values and specific operation modes. The approach revealed several faults in the prototype.

The next section further investigates a mixed strategy in the case of Lustre programs.

4 Mixed test strategy for synchronous data flow programs

Synchronous data flow languages are designed for programming reactive and real-time systems, and especially those having safety-critical requirements [2]. For these languages, the two standard models of program structure, that is, the control flow graph and the data flow graph, do not fit any more; and this makes obsolete the use of classical path selection criteria as guides for deriving test inputs from a structural analysis of the program. Hence, the need for structural testing strategies proper to synchronous data flow programs. Through the example of the language Lustre [6], this issue has been investigated and a mixed strategy which combines statistical testing with deterministic testing of special values has been defined and exemplified by two industrial case studies. A salient feature of the strategy is that it may be applied at either the unit or integration levels of a gradual testing process. Let us first introduce the principle of the strategy before reporting on results.

4.1 Principle of the test strategy based on Lustre automata

To define a strategy which may be applied at either the unit or integration testing levels, two issues must be addressed: (i) how to design test patterns and, (ii) how to define the testing levels.

Method for designing test patterns

A Lustre program has a cyclic behavior. Any variable or expression denotes a flow which is a pair made of a possibly infinite sequence of values, and a sequence of times related to the cyclic behavior of the program. A program is structured into nodes (subroutines). A node contains a set of equations and any variable which is not an input parameter has to be defined by a single equation. For example, the equation "X = 0 –> pre(X) +1;" defines a counter of basic clock cycles: X is 0 initially, and then is incremented by one at each clock cycle.

The Lustre compiler produces sequential code. First, it expands recursively all the nodes called by the program so that the code generation step starts with a node which does not call any node. Then it may perform an automaton-like compilation, based on an analysis of the internal states of the program. The result is a finite state automaton in which: (i) each state is characterized by a combination of values of state variables; (ii) each transition is labeled by a combination of input or internal values which defines the condition that triggers it; a transition consists in executing a linear piece of code and corresponds to an elementary reaction of the program. This automaton is expressed in a format common with other synchronous data flow languages. A code generator takes this format as input and produces a program in the C language.

Since the Lustre automaton carries the control structure of the program, we have used it as a guide for designing test inputs, the criterion retained being the coverage of the automaton transitions [14]. Based on this criterion, complementary statistical and deterministic test inputs are designed according to the method briefly recalled below.

As regards *statistical testing*, the influence of the input distribution on the transition probabilities can be studied on the stochastic graph obtained by replacing the input conditions that label the transitions with their occurrence probabilities in the distribution. But, when initial values appear in the equations defining the variables, the Lustre compiler introduces an initial state which corresponds to the very first execution of the program during which the variables are set to their initial values. Since the experiments reported in Section 3.2 led us to conclude that the initialization process is a typical case where deterministic testing is more appropriate, statistical testing may be focused on all the program states and transitions, except the initial state and its output transitions. As a result, the probabilistic analysis is conducted on the stochastic graph obtained after deletion of the initial state. Then the minimum test size to reach a target test quality q_N is deduced from Relation (1). These statistical test patterns supply a stringent coverage of the automaton, yet without tracking any special values inferred from the automaton.

Additional *deterministic test patterns* are selected to ensure the coverage of special values which include: (i) the automaton initial state – if any – and its output transitions; (ii) boundary values related to the conditions that trigger the transitions. For example, if two transition conditions are labeled "$X < Y$" and "$X \geq Y$", where X and Y are input variables, then $X = Y$ is a boundary value to be verified through deterministic testing: its occurrence probability is likely to remain low in the statistical patterns, since the probabilistic analysis does not differentiate between $X = Y$ and $X > Y$.

Definition of unit and integration testing levels

The method proposed for selecting the nodes to be tested at the various levels aims to minimize the test effort by: (i) gathering as many Lustre nodes as possible to test them all together and, (ii) avoiding redundant testing of nodes called by different nodes. Based on both guidelines, an algorithm to define the unit testing level has first been formulated, and then generalized to deal with the definition of the integration levels [8, 16]. These algorithms are based on the bottom-up exploration of the program call graph from its terminal nodes.

The stopping criterion of the bottom-up exploration used to define the nodes to be tested at the unit level is the automata complexity (number of transitions): testing of the nodes whose structure is no longer tractable with respect to the probabilistic analysis for designing statistical patterns, is postponed to the integration phase(s). In essence, integration testing disregards the structure of nodes previously tested to focus on their interactions. This reducing principle leads us to progressively decrease the automata complexity by removing part of the called nodes previously tested (only the nodes which do not influence significantly the control structure of the calling node may be disregarded[1]), so that the bottom-up process can go on. As a result, the algorithm for defining the unit level can be generalized to deal with the definition of successive integration levels by referring to simplified automata.

To conclude, it is worth noting that since the test strategy is based on a white box analysis of the Lustre nodes, its feasibility is restricted to programs whose structural complexity remains tractable. Here, a node complexity is expressed in terms of the number of transitions involved in the Lustre automaton (either the complete automaton or a simplified automaton, depending on the testing level). In practice, when an automaton contains more than about 30 transitions and may not be simplified any more, the bottom-up exploration is stopped: for the associated node and all its calling nodes, the design of test input patterns has to be based on functional models such as those presented in Section 3.2. The two case studies presented below illustrate both alternatives: either the successive integration levels allow us to reach the root node of the program, or not.

4.2 Application to two industrial case studies

The first case study, *SWITCH*, is extracted from a fault-tolerant monitoring system which regulates the outputs of eight gates or burners in a thermal power station; the second one, *SRIC* (*S*ource *R*ange *I*nstrumentation *C*hannel), is extracted from a monitoring system of a nuclear reactor.

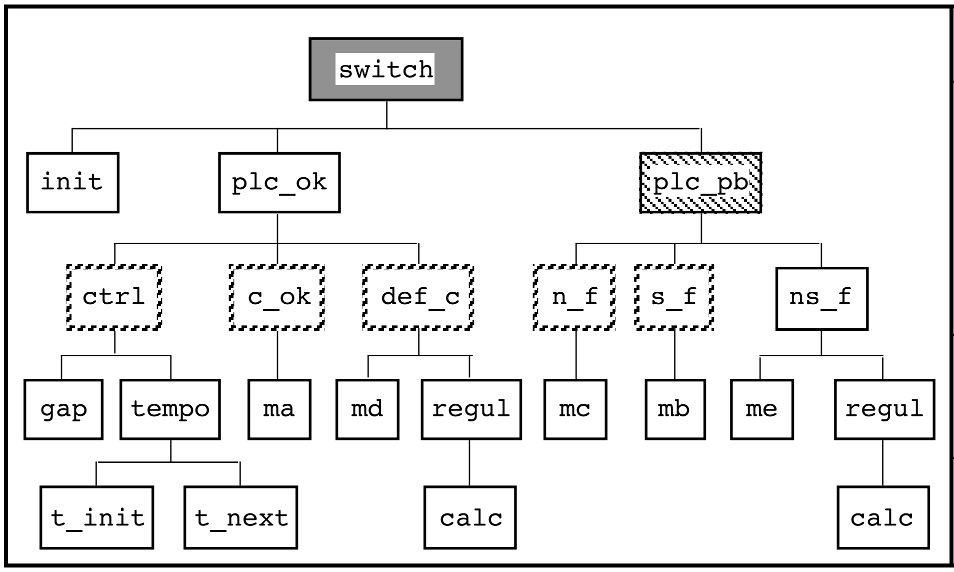
[1] Formal rules to decide whether or not called nodes may be disregarded are given in [8].

The program *SWITCH*

SWITCH was used as an experimental support to the definition of the test strategy [8]. It belongs to the subsystem which controls analogical signals coming from two Programmable Logic Controllers in standby redundancy. It is made up of 21 different Lustre nodes. Figure 5a shows its call graph which contains 23 nodes, two nodes – `calc`, `regul` – being called twice. Three testing levels were needed to reach the root node (see Fig. 5a):

- At the *unit testing* level, 5 nodes were retained: `ctrl`, `c_ok` , `def_c`, `n_f` and `s_f`; the result of this is that a total of 15 different nodes are tested at this level. For example, the automaton of `def_c` carries the control structure of the 3 called nodes – `md`, `regul` and `calc`; hence, the test patterns designed from this automaton allow us to test these 3 nodes through `def_c`.
- At the *first integration* level, the node `plc_pb` was retained; the simplified automaton of `plc_pb` was got by disregarding the nodes tested at the unit level, that is all the nodes below `plc_pb` except `ns_f` and `me`: it has only 7 transitions instead of 111 without the simplifications; the nodes `ns_f` and `me` are tested through `plc_pb`.
- The *second integration* level allowed us to reach the root node; the simplified automaton of `switch` was got by disregarding all the nodes except `init` and `plc_ok` which have not been tested yet, and `ctrl` which influences significantly the control structure of `plc_ok` (and thus, it may not be disregarded according to the simplification rules [8]); it has 20 transitions, instead of more than 200 without the simplifications allowed by the definition of first two testing levels.

The whole program is then tested through the seven nodes retained. Figure 5b tabulates the number of test patterns generated according to the mixed test strategy. The sizes of the statistical test sets correspond to a test quality of $q_N = 0.9999$. Deterministic patterns have been added only when special values were identified on the automata. The error detection power of the test patterns have been assessed using mutation analysis. Main experimental results are reported in Section 4.3.



Node	**Ns**	**Nd**
Unit testing:		
`ctrl`	146	17
`c_ok`	69	—
`def_c`	223	39
`n_f`	69	—
`s_f`	69	—
Integration 1:		
`plc_pb`	60	—
Integration 2:		
`switch`	287	39

(a) Call graph and testing levels:

⬚ **unit testing** ▨ **1st integration testing** ■ **2nd integration testing**

(b) Numbers of statistical (Ns) and deterministic (Nd) test patterns

Fig. 5. The Lustre program *SWITCH*

The program *SRIC*

SRIC is extracted from a software system whose goal is mainly to generate actions to protect a nuclear reactor when some physical values are beyond given thresholds. The program has been developed in the SCADE™ environment: it is made up of 36 different Lustre nodes and the C code automatically generated by SCADE approximates 2600 lines.

For this program, four testing levels were defined [11, 18]: the unit level selects ten different nodes, the first integration level selects six nodes, the second and the third integration levels select one node each. A total of 30 nodes are tested through the 18 nodes retained, which means that six nodes are not manageable according to the structural test strategy. This is due to the fact that two automata contain more than 130 states each, and they may not be simplified. These two nodes compel us to stop the bottom-up exploration of the call graph leaving four other nodes not reached. For these six nodes, a functional testing approach will be used.

Test patterns have been generated for the 18 nodes retained: the sizes of the statistical test sets range from 23 to 725 input patterns (for $q_N = 0.9999$), and the maximum number of deterministic patterns for a node amounts to 40.

™ SCADE is a registered trademark of VERILOG SA.

4.3 Experimental results

The experimental assessment of the effectiveness of the test patterns generated for the program *SRIC* is still under investigation, so that no final results may be reported here for this program. Nevertheless, let us note that the automata analyses performed to design the test patterns have allowed us to identify two minor faults, which may not produce failures in the real system due to hardware compensation.

For the program *SWITCH*, the error detection power of the test patterns generated was assessed against mutations that take into account the peculiarities of Lustre [14]. The experiments aimed at: (i) comparing the effectiveness of statistical inputs with random (uniform) ones, (ii) analyzing the complementary features of statistical and deterministic inputs, and (iii) analyzing the relevance of the successive testing levels. As regards issue (i), the results confirmed the best efficiency of the statistical approach as already shown in the experiments conducted on procedural programs (Section 3). We now concentrate on the other two issues.

Summary of the experimental results

For each node retained at a testing level – called target nodes below – five different statistical test sets of size Ns (see Fig. 5b) were generated; they are denoted Sk. Figure 6 summarizes the results of the mutation analysis performed on the whole program. For each target node, it tabulates:

- the called nodes which are tested through the target node;
- the number of mutants produced by faults seeded in the target node and in the called nodes tested through the target one;
- the lowest and the highest numbers of mutants remaining alive after completion of (i) one of the 5 different statistical test sets Sk and, (ii) the mixed test strategy got by one Sk supplemented by the set D of Nd deterministic inputs – if any (see Fig. 5b).

Complementary features of statistical and deterministic test inputs

Deterministic test inputs were added for three nodes, and in all cases they allowed us to increase the final mutation score. As regards `ctrl`, 4 mutations were never revealed by the sets Sk; but they are typical cases of boundary values, and thus, they are revealed by the deterministic patterns D.

For `def_c`, from 3 to 8 mutants were left alive by the sets Sk. Their analysis shows that: (i) 3 mutants are never killed whatever the set Sk; they are cases of boundary values which are the target of the deterministic test set D; (ii) 7 mutants are killed only by some of the sets Sk; they correspond to the insertion of the operator "pre" which turns the memoryless behavior of the program into a history-sensitive behavior; revealing these faults requires that the mutants be supplied with two successive input patterns in a specific order; nevertheless, 5 of the 7 mutants are killed by the deterministic patterns D. More generally, the analysis of the mutants alive has shown that the insertion of the operator "pre" can induce very subtle faulty behaviors.

			# Mutants alive	
Target node	Nodes tested through the target node	# Mutants	Sk	Sk + D
`ctrl`	`gap, tempo, t_init, t_next`	311	4	0
`c_ok`	`ma`	503	0	—
`def_c`	`md, regul, calc`	493	3 – 8	0 – 2
`n_f`	`mc`	506	0 – 1	—
`s_f`	`mb`	504	0 – 1	—
`plc_pb`	`ns_f, me`	317	0 – 10	—
`switch`	`init, plc_ok`	173	7 – 8	0 – 3

Fig. 6. Results of mutation analysis conducted on the program *SWITCH*

For `switch`, the mutations not revealed by the sets Sk are faults seeded in the node `init` which performs the initialization process of the whole program: here the deterministic inputs were not stringent enough to reveal subtle faults related to the initial state and its output transitions.

These results support the high error detection power of the statistical test sets designed according to the transition coverage criterion: the minimum mutation scores reached by the sets Sk range from 0.968 to 1.0 depending on the nodes. They show how deterministic testing of special values deduced from the Lustre automaton provides an additional effectiveness, in particular for boundary values related to the conditions that trigger the transitions.

Nevertheless, deterministic testing was not able to provide a sound probe of the initialization process at the top level, this process involving a significant number of elementary operations. These results show (once again) that the deterministic approach suffers from severe limitations as soon as the tested function is not trivial. Hence, the solution we propose for non-trivial initialization functions is to return to the statistical approach and design a specific test profile to stress the initialization process.

Relevance of the successive testing levels

Additional experiments were conducted to answer the question of whether or not the test sets defined at the different testing levels are redundant with each others. If the answer is Yes, it means that some of the test sets listed in Figure 5b may be needless. For example, if the statistical inputs used to test `plc_pb` at the first integration level provide a sound test of its called node(s) retained at the unit level (`n_f` or `s_f` or both) then unit testing of the called node(s) would become useless. The experimental analysis of potential redundancies was based on the comparison of the mutation scores for a given node tested through different levels. Figure 7 illustrates the two different cases that were observed (see [8] for the whole set of experimental results):

- A large number of mutations in the nodes tested at level k are not revealed by the test sets defined at level k+1, thus justifying the two testing levels (see nodes `s_f` and `mb` in Fig.7).
- The test sets at level k and k+1 exhibit similar performance and thus, should be redundant; this case is exemplified by the nodes `ctrl`, `gap`, `tempo`, `t_init` and `t_next` in Fig. 7, for which the results were not surprising since: (i) the simplification rules did not allow us to disregard `ctrl` to get the simplified automaton of the root node `switch` and (ii) under the test profile and the test size defined for `switch`, the node `ctrl` is called 251 times on average by each set Sk, thus providing a stringent coverage of the `ctrl` automaton transitions.

# Mutants	**# Mutants alive**	
`s_f`, `mb`: 504 mutants	Unit testing of `s_f`: 0 - 1	Integration testing from `plc_pb`: 54 - 149
`ctrl`, `gap`, `tempo`, `t_init`, `t_next`: 311 mutants	Unit testing of `ctrl`: 0	Integration testing from `switch`: 0 - 1

Fig. 7. Comparison of the error detection powers of the test sets defined at different levels

More generally, the whole set of experiments has shown that testing of a node `nx` retained at a given testing level k may be skipped when the following conditions hold: (i) there is a node `ny` retained at level k+1 whose simplified automaton does not disregard `nx` and, (ii) under the test profile and the test size defined for `ny`, the average number of calls to the node `nx` is sufficient to reach the target test quality with respect to the transitions of the `nx` automaton. These conditions have to be checked before the generation of the test patterns, in order to minimize the test effort. For the program *SWITCH*, unit testing of two nodes – `ctrl` and `c_ok` – may be skipped.

5 Conclusion and future direction

The results reported in the paper synthesize about ten years of research work on the probabilistic generation of test inputs. Because the probabilistic method for generating test inputs was generally related to uniform test patterns, it was often deemed a poor methodology: our experimental studies deny this preconceived idea. Indeed, the basic motivation of our work was to remove the blind feature of the conventional probabilistic generation, by using information on the program under test as a guide for determining a test profile and a test size. In the deterministic testing approaches, such information is provided by test criteria related to a model of either the program structure or of its functionality. We have defined a rigorous method for designing probabilistic test cases based on such test criteria, and called it *statistical (structural or functional) testing* to avoid any confusion with the conventional random (uniform) approach.

The method consists in deriving test profiles from structural or functional test criteria. Then the test profile may have little connection with actual usage (operational profile): the focus is bug-finding, not reliability assessment. As a result, our approach should not be confused with (statistical) *operational testing* which suffers from *severe limitations* when testing aims at finding faults, and in particular in cases of safety-critical software [15]: when the failure severity is much more crucial than the occurrence rates, an operational profile may be quite inadequate since there are scenarios where the system dependability is of utmost importance, but which are expected to occur only very rarely during real operation (e.g. pressure drop in a nuclear plant).

A number of experiments conducted on safety-critical programs, a sample of which is reported in the paper, have confirmed the soundness of statistical testing which has repeatedly exhibited a higher error detection power when compared with deterministic testing approaches (including the current industrial practice) and random testing. Nevertheless, a limitation of the statistical test patterns experimented on was their lack of adequacy with respect to faults related to extremal/special cases. Such faults induce, in essence, a very low probability of failure under the profiles defined to ensure a global probe of the target programs: they require test inputs specifically aimed at them. Hence, we support the adoption of the *mixed test strategy* involving both statistical and deterministic test patterns, as illustrated in the paper on synchronous data-flow programs written in the language Lustre.

The main conclusion arising from this work is that statistical testing is a suitable means to compensate for the tricky link between test criteria and software design faults. Because of the lack of an accurate model for software design faults, this imperfect connection is not likely to be solved soon, and it is expected to get even worse in the case of *object-oriented (OO) programs*. The OO paradigm raises a number of challenging issues from a tester's viewpoint and, as a first step, a feasibility study of statistical structural testing in cases of small OO programs was performed [17]. The idea was to combine path selection techniques with the consideration of one specific OO concept: inheritance. This preliminary study allowed us to identify a number of problems, and on-going research work is now concentrated on two fundamental questions: (i) how to define the unit and integration testing levels for OO software systems? and, (ii) which software models and associated test criteria should be used as guides for designing statistical test patterns?

Acknowledgements

This work was partially supported by the European Community (ESPRIT Projects n° 3092 and n° 6362: PDCS and PDCS2. It is currently partially supported by the European Community (ESPRIT Project n° 20072: DeVa), and by the French Government (DSP-STTC/CNRS/MENRT Project : FORMA). We wish to thank our colleagues Yves Crouzet and Christine Mazuet very much: all the experiments involving mutation analysis were conducted using the SESAME tool designed and implemented in our laboratory by Yves Crouzet; the test strategy for Lustre programs was firstly defined by Christine Mazuet within the framework of her PhD research work.

References

[1] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, 2nd Edition, 1990.

[2] A. Benveniste and G. Berry, “The Synchronous Approach to Reactive and Real-Time Systems”, *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270-1282, Sept. 1991.

[3] M. Daran and P. Thévenod-Fosse, “Software Error Analysis: A Real Case Study Involving Real Faults and Mutations”, in *Proc. ACM Int. Symp. Software Testing and Analysis (ISSTA'96)*, San Diego,USA,pp.158-171, 1996.

[4] R. A. DeMillo, R. J. Lipton and F. G. Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer”, *IEEE Computer Magazine*, vol. 11, no. 4, pp. 34-41, 1978.

[5] J. W. Duran and S. C. Ntafos, “An Evaluation of Random Testing”, *IEEE Trans. on Software Engineering*, vol. SE-10, no. 4, pp. 438-444, July 1984.

[6] N. Halbwachs *et al.*, “The Synchronous Data Flow Programming Language Lustre”, *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305-1320, Sept. 1991.

[7] D. Harel *et al.*, “STATEMATE: A Working Environment for the Development of Complex Reactive Systems”, *IEEE Trans. on Software Engineering*, vol. SE-16 no. 4, pp. 403-414, April 1990.

[8] C. Mazuet, “Stratégies de test pour des programmes synchrones : application au langage Lustre”, Doctoral Dissertation, INPT, Toulouse, LAAS Report 94508, Dec. 1994.

[9] S. C. Ntafos, “On Testing with Required Elements”, in *Proc. COMPSAC'81,* pp. 132-139, Nov. 1981.

[10] S. Rapps and E. J. Weyuker, “Selecting Software Test Data Using Data Flow Information”, *IEEE Trans. on Software Engineering*, vol. SE-11, no. 4, pp. 367-375, April 1985.

[11] A. Robisson, “Test de programmes Lustre”, LAAS Report 97249, June 1997.

[12] P. Thévenod-Fosse, H. Waeselynck and Y. Crouzet, “An Experimental Study on Software Structural Testing: Deterministic Versus Random Input Generation”, in *Proc. 21st IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, Montréal, Canada, pp. 410-417, June 1991.

[13] P. Thévenod-Fosse and H. Waeselynck, “STATEMATE Applied to Statistical Software Testing”, in *Proc. Int. Symp. on Software Testing and Analysis (ISSTA 93)*, Cambridge, Massachusetts, pp. 99-109, June 1993.

[14] P. Thévenod-Fosse, C. Mazuet and Y. Crouzet, “On Statistical Structural Testing of Synchronous Data Flow Programs”, in *Proc. 1st European Dependable Computing Conf. (EDCC-1)*, Berlin, Germany, pp. 250-267, Oct. 1994.

[15] P. Thévenod-Fosse and Y. Crouzet, “On the Adequacy of Functional Test Criteria Based on Software Behaviour Models”, in *Proc. 5th IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-5)*, Urbana-Champaign, USA, pp. 176-187, Sept. 1995.

[16] P. Thévenod-Fosse, C. Mazuet and Y. Crouzet, “Defining the Unit Testing Level of Synchronous Data Flow Programs”, in *Proc. 15th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP'96)*, Vienna, Austria, pp. 115-125, Oct. 1996.

[17] P. Thévenod-Fosse and H. Waeselynck, “Towards a Statistical Approach to Testing Object-Oriented Programs”, in *Proc. 27th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-27)*, Seattle, USA, pp.99-108, June 1997.

[18] P. Thévenod-Fosse, “Unit and Integration Testing of Lustre Programs: A Case Study from the Nuclear Industry”, LAAS Report 98078, Feb. 1998 (Submitted for publication).