An Experimentation with Statistical Testing

Hélène Waeselynck*
INRETS
20, rue Elisée Reclus
59650 Villeneuve d'Ascq - FRANCE
Tel: (+33) 20 43 83 23
Fax: (+33) 20 43 83 59
E-mail: waeselyn@inrets.fr

Pascale Thévenod-Fosse
LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex - FRANCE
Tel: (+33) 61 33 62 37
Fax: (+33) 61 33 64 11
E-mail: thevenod@laas.fr

* At LAAS-CNRS at the time of the study reported here.

Abstract

Statistical testing is based on a probabilistic generation of test data: classical 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 current criteria with software faults, and should not be confused with *random testing*, a "blind" approach that uses a uniform profile over the input domain.

This paper reports on experimental results obtained on a software component from the nuclear field:

- **unit testing of four functions** – statistical input sets were designed according to structural criteria; their efficiency was compared to the one of 1) deterministic sets derived from the same criteria and 2) uniform random sets; the comparison involved 2816 faults of mutation type seeded one by one in the source codes.

- **whole component testing** – statistical functional testing was designed from behaviour models of the component: finite state machines, decision tables, Statecharts; its efficiency was compared to the one of random testing, using two versions of the component: the real one, in which a minor fault was found, and a student version with 12 revealed faults.

The results showed the high fault revealing power of statistical testing, and its best efficiency in comparison to deterministic and random testing.

Prerequisite Key Words: None

Topic Descriptors: Structural and functional criteria, random and deterministic test patterns.

Introduction

Testing involves exercising the software by supplying it with input values. In practice, testing is partial as it is not feasible to exercise a piece of software with each possible data item from the input domain. When the focus of testing is **fault removal**, that is, bug-finding and not reliability assessment, the tester is faced with the problem of selecting a subset of the input domain that is well-suited for revealing the real, but unknown, faults; this issue being further compounded by the increasing complexity of real software systems. Then, the methods for generating test inputs proceed according to one of two principles: either deterministic or probabilistic.

The **deterministic methods** for generating test inputs usually take advantage of information on the target software in order to provide guides for selecting test cases, the information being depicted by means of test criteria. A lot of test criteria have been defined (see e.g. [Myers 79], [Howden 87], [Beizer 90]): each of them defines a specific set of elements to be exercised during testing; these elements being parts of either a model of the program structure or a model of its functionality. For example, the program control flow graph is a well-known structural model, and finite state machines are behaviour models that may be used to describe some software functions. State coverage (i.e. instruction testing in the case of a program flow graph) and transition coverage (i.e. branch testing in the case of a program flow graph) are two classical examples of test criteria associated with these models. Given a criterion, the deterministic principle consists in selecting a priori a set of test inputs such that each element defined by the criterion is exercised (at least) once; and this set is most often built so that each element is exercised *only once*, in order to minimise the test size. Unfortunately, an acute question still arises from the definition of test criteria: a strong limitation is due to the imperfect connection of the criteria with the real faults and, because of the (current) lack of an accurate model for software design faults, this problem is not likely to be solved soon. Hence, exercising only once, or very few times, each element defined by such imperfect criteria is likely to be far from being enough to ensure that the corresponding test set possesses a high fault exposure power. And this is the main reason why the efficiency of deterministic testing depends more on the particular test input values chosen than on the criterion retained [Hamlet 89].

On the contrary, the conventional probabilistic method for generating test inputs, called **random testing**, consists in generating random test data based on a uniform distribution over the input domain [Duran 84]: this is an extreme case of black box testing approach, no information related to the target piece of software being considered, except for the range of its input domain. The argument in favour of random testing is its low cost: large test sets can be generated cheaply, that is, without requiring any preliminary analysis of the software. Indeed, the fault revealing power of such a "blind" testing approach is questionable, even if large test sets are used, and previous work has shown that its effectiveness can be surprisingly high as well as surprisingly low, depending on both the target software and the particular input values randomly drawn [Ntafos 81, Duran 84].

Statistical testing is based on an unusual definition of random testing, with the purpose of removing the blind feature of the conventional probabilistic generation [Thévenod 89]. 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. And for

this, the method for generating statistical test inputs combines the information provided by a model of the target software, that is, by a test criterion, with a practical way of producing large test sets, that is, a random generation. Indeed, statistical testing aims to cope with current (imperfect but not irrelevant) criteria and compensate their weakness by requiring that each element be exercised several times. The statistical test sets are then 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. 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*.

The experimental investigation summarised in the paper aims to assess and compare the efficiency of the three methods for generating test data: deterministic, random and statistical. The experiments were conducted with programs taken from a real, industrial nuclear reactor safety shutdown system. Although the results got from a single application are not sufficient to draw general conclusions on the adequacy of the test methods, they have allowed us to confirm serious limitations of both the random and the deterministic approaches; the statistical method being likely to be a practical way of compensating for most of these limitations.

This paper proceeds as follows. *Section 1* outlines our experimental framework. Then, in the light of the results got from our real case study, *Sections 2, 3 and 4* concentrate on the analysis of the strengths and weaknesses of respectively random testing, deterministic testing and statistical testing.

1. Case Study for Safety Critical Software

1.1. Target Programs

The experiments involved a software component extracted from a **nuclear reactor safety shutdown system**, and belonging to the part of the system that periodically scans the position of the reactor's control rods. At each operating cycle, 19 rod positions are processed. The information is read through five 32-bit interface cards. After acquisition, the data are checked and filtered. Then, the measurements of the rod positions (in Gray code) are converted into a number of mechanical steps. *Degenerated operating modes* are obtained when one or more interface card is declared inoperational in the current cycle, so that the information it supplies is not taken into account. In the worst situation all cards are inoperational and a *minimal service* is delivered: no measure is provided, only routine checks are carried out. Reset operations can bring back the system from partial to full service.

The implementation of the component approximates a thousand lines of C language (without comments). It consists of a big controller and four small unit functions FCTi (i = 1, …, 4): FCT1 and FCT2 perform data acquisition, FCT3 is the filtering unit and FCT4 the conversion unit. Experimentation was carried out at various levels of integration: first at the **unit level** (functions FCTi); then at the **level of the whole component**. The latter experiments also involved a *second version of the component* developed by a student from the same requirements summarised above.

1.2. Overview of the Experiments

The experiments have been designed according to **two main investigation goals**:

(1) *Comparison of the three methods for generating test data*: deterministic, random and statistical;

(2) *Study of input profiles for statistical testing*; here, the aim was twofold: first, to analyse the impact of the test profile (uniform or designed) on the effectiveness of the probabilistic approach; and second, to show the feasibility of designing proper profiles from a model of the target software.

Goal (1) was addressed by the unit testing experiments. Classical structural criteria [Rapps 85] were used to design both statistical and deterministic test sets; random sets were also generated. The comparison of three types of sets involved 2816 faults of mutation type [DeMillo 78] seeded one by one in the source codes.

Goal (2) was addressed at both levels: structural profiles were first investigated (see above); then functional profiles were designed from behaviour models of the component: finite state machines, decision tables, Statecharts. At the component level, the comparison with the uniform profile involved 13 real faults, one minor fault related to the real version and 12 faults residing in the student version.

The next sections comment on the results supplied by each of the testing approaches – random, deterministic, statistical (see also [Thévenod 91-93], [Waeselynck 93]).

2. Random Testing

In [Ntafos 81], conventional random testing was experimented on five small programs. The author noticed that uniform testing 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 data. Our results corroborate his observations and clearly show that this testing method is likely to be a poor methodology, in most cases.

2.1. Experimental Results on Both Unit and Component Testing

As regards **unit testing**, the uniform distributions have been investigated for each FCTi through 1 to 5 test sets, depending on the function complexity; each set was the same size N as the larger statistical structural test set related to the same function (see 4.2). Figure 1 tabulates the test sets and displays the numbers of faults they do not reveal.

	#	N	faults not revealed
FCT1, 265 seeded faults	1	170	0
FCT2, 548 seeded faults	1	80	0
FCT3, 1416 seeded faults	5	405	229–626
FCT4, 587 seeded faults	5	850	49–61

Fig.1 Numbers of seeded faults not revealed by the random sets in unit testing. # denotes the number of test sets with N inputs per set.

For FCT1 and FCT2, the uniform distributions provide a 100% paths coverage of the programs: all the faults are revealed, which can be deemed very cost-effective if one considers that the test data generation requires little effort. Such efficiency is likely to be due to the comparative simplicity of both programs. Indeed, *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, for which hundreds of faults are not uncovered by the random sets. Besides, the results vary from one set to the other: from 229 up to 626 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.

The results supplied at the **component level** were even worse with respect to the 13 real faults uncovered during the whole set of our experiments. A single uniform test set was generated: it contains 5300 inputs, the test size being arbitrarily chosen without preliminary analysis of the component: this is in conformity with the foundation of uniform testing, that is, large test sets generated cheaply. Indeed, this set is an order of magnitude larger than the other (statistical) test sets experimented with (see 4.3). Nevertheless, it does not reveal 8 of the 13 faults. This is not a surprising result, since it was verified that the set poorly probes both component versions from both a structural and a functional viewpoint (with respect to the structural coverage, four blocks of instructions of the student version and one block of the real version are never exercised).

2.2. Explanation for the Inadequacy of the Uniform Distribution

To investigate the behaviour of random test patterns, a detailed analysis of the evolution of the fault exposure power 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: 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. For FCT1 and FCT2, all the faults are revealed within the first 25 executions; for FCT3 and FCT4, the final scores are obtained in the first half of the test sets. For the component, the five faults are revealed within the first 633 executions; the remaining 4667 executions being garbage. Indeed, the residual faults induce a very low failure probability under the uniform profile.

As a result, little improvement is expected if the tests are further pursued, unless a very large number of extra input data is generated. This suggest that *the inadequacy of the uniform distributions 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 data being unlikely to be uniformly distributed over the input domain, a uniform profile is not relevant to increase the program failure probability during testing.

3. Deterministic Testing

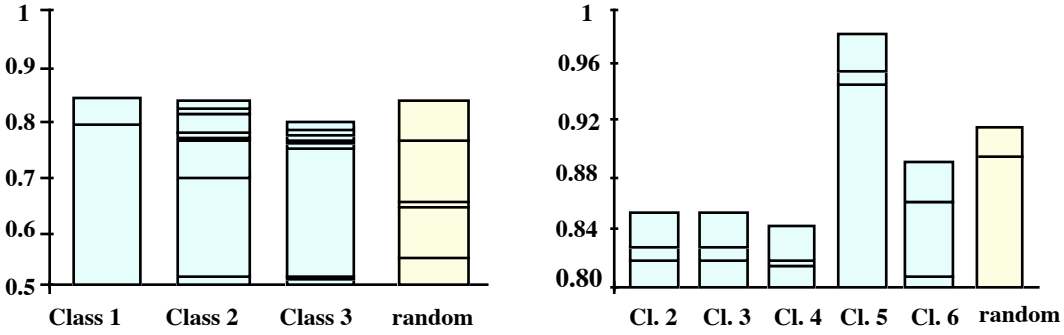
As mentioned at the beginning of the paper, the weakness of deterministic testing is likely to be due to the tricky link between the test criteria and the faults they aim to

track down. As regards *structural* criteria, this limitation was confirmed by the experimental results on unit testing.

3.1. Experimental Results on Deterministic Structural Testing

Figure 2 displays the proportion of seeded faults revealed by the deterministic structural testing of FCT3 and FCT4 (the entire set of results is detailed in [Thévenod 91]); for the purpose of comparison, the results of random testing are also recalled.

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.



(a) FCT3, 1416 seeded faults **(b) FCT4, 587 seeded faults**

Fig.2 Proportion of seeded faults revealed by the deterministic structural test sets.

As it 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 faults: 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.

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 data, 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.

3.2. Explanation for the Limitation of Deterministic Testing

The previous results may be analysed from a more general perspective, in relation to theoretical work on partition testing [Hamlet 90, Weyuker 91, Waeselynck 93].

Partition testing refers to any test data selection method which divides the input domain into subsets, and forces at least one test case to come from each subdomain. Most deterministic testing schemes belong to this family of strategies, including structural schemes: their subdomains group together inputs that exercise a same structural element.

To minimise the test size, a *single* input case is generally drawn from each subdomain, and all the authors agree that it is a poor strategy unless the subdomains are nearly homogeneous (or *revealing*), that is, they contain only failure-causing inputs or none. It is worth noting that a practical effect of the weakness of current criteria is that the subdomains they define are usually not revealing.

Although a structural fault model (mutations) was used for our unit testing experiments, many seeded faults were loosely connected to structural criteria, as were the three following ones:

(1) The *fault* f_1 (FCT3) disturbs a branch predicate. Yet, it was not revealed by the All-Paths test sets: owing to error masking in the function, executing the wrong branch does not always produce an incorrect output.

(2) The *fault* f_2 (FCT3) modifies the handling of a global variable, thereby turning the combinational behaviour of FCT3 into a sequential one. Revealing test sets must contain two consecutive patterns in a specific order: the order was not adequate in the Class 1 input sets, but some Class 2 and Class 3 sets happened to reveal the fault.

(3) The *fault* f_3 (FCT4) corrupts a pointer so that read/write operations are performed in an improper memory area. As was stated by repeating the experiments twice, the environmental context alters the faulty behaviour, making it unforeseeable: the outcome of a same test set (f_3 revealed or not) could vary from one run to the next.

Example (1) could be seen as a case for partition refinement: the paths with the faulty branch define subdomains that are too large relative to the set of failure-causing inputs. But without the knowledge of where the faults are, there is no way to insure that a refined partition performs significantly better than the original one; and current fine partitions (like All-Paths) have often proven to be unsuccessful.

Examples (2) and (3) invalidate the common assertion that exhaustive testing (the finest possible partition with single element subdomains) yields a correctness proof of the program. It would be true only if the program could be proven to exhibit a combinational behaviour, which is not granted even for simple functions: just as in some well-known cases of physical faults in hardware components, *design faults in software components may originate either a sequential behaviour or intermittent failures, or both*. Having analysed a large number of seeded faults, we claim that such cases are likely to occur.

Instead of improving current partition schemes, an alternative direction may be to cope with imperfect criteria and compensate their weakness through subdomain sampling. Statistical testing is a practical way to implement this strategy.

4. Statistical testing

4.1. Principle and Method

Any criterion specifies a set of elements (e.g. subdomains) to be exercised (sampled) during testing: given a criterion C, let S_C be the corresponding set of elements. When using a probabilistic method for generating test data, the number of times an element k of S_C is exercised is a random variable depending on two factors: the **input profile**, which determines the probability of exercising k; and the **test size** N. The notion of test quality with respect to a criterion [Thévenod 89] provides us with a theoretical framework to ensure that *on average* each k is exercised several times, whatever the particular test set generated according to the test profile and within a moderate test duration:

DEFINITION. A criterion C is covered with a probability Q_N if each element of S_C has a probability of at least Q_N of being exercised during N executions with random inputs. Q_N is **the test quality with respect to (wrt) C.**

The quality Q_N is a measure of the test coverage wrt C. Let P_C be the occurrence probability per execution of the *least likely* element under the chosen profile. Then the test quality and the test size N are linked by the relation: $(1-P_C)^N = 1-Q_N$, or equivalently:

$$N = \ln (1-Q_N) / \ln (1-P_C) \qquad (1)$$

There is 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, $n \cong 7$ for $Q_N = 0.999$, and $n \cong 9$ for $Q_N = 0.9999$.

Based on this, **the principle of the method for designing a statistical test set according to a given criterion C** involves two steps:

(i) *search for an input profile* which accommodates the highest possible P_C value;

(ii) *assessment of the test size N* required to reach a target test quality Q_N wrt C, given the value of P_C inferred from the first step; relation (1) yielding the test size.

Two different ways of deriving a proper profile are possible: either *analytical*, or *empirical*. 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 function of the input probabilities, facilitating the derivation of a profile that maximises the frequency of the least likely element. The second way consists in instrumenting the software in order to collect statistics on the number of activations of the elements: starting from a large number of input data 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.

Going back to the imperfect connection of the criteria with the actual faults, it is worth noting that the criterion does not influence data 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 data items. The

efficiency of the probabilistic approach relies on a single assumption: the information supplied by the criterion retained is relevant to derive a test profile that enhances the program failure probability.

4.2. Experimentation with Statistical Structural Testing

A total of 22 statistical test sets have been designed according to the structural analysis of the four functions FCTi (i=1, ..., 4).

First, **the most stringent achievable criteria have been adopted**: All-Paths for the first three functions, All-Uses for FCT4; and the proper profiles have been defined analytically. For a target $Q_N = 0.9999$, the test sizes N for each FCTi are respectively, from relation (1): 170, 80, 405, 850. Twelve test sets have been generated: one for FCT1 and for FCT2; five different sets for FCT3 and for FCT4. For these two functions, which are more complex than the others on both a structural and a functional viewpoint, we have also derived input profiles from **weak criteria**, respectively All-Defs (Class 3 of FCT3) and Branches (Class 6 of FCT4). Five sets of 152 (resp. 42) test patterns have been generated according to each profile, taking again $Q_N = 0.9999$. It is worth noting that the Class 3 profile of FCT3 let two paths have a null probability of being executed.

Figure 3 displays the scores supplied by the statistical sets versus the scores of (1) deterministic sets derived from the same criteria, and (2) random sets. For all functions, **the most efficient test data are the statistical ones**. High scores are *repeatedly* observed, whatever the particular set generated according to a same structural profile: the statistical testing approach allowed to increase significantly the failure probability, even as regards subtle faults loosely connected with the criteria (see 3.2).

	structural testing				
	stringent criteria		**weak criteria**		**random testing**
FCT1, 265 seeded faults	**statist. set** determin. sets	**100%** 99.6–100%	—		100%
FCT2, 548 seeded faults	**statist. set** determin. sets	**100%** 99.3–100%	—		100%
FCT3, 1416 seeded faults	**statist. sets** determin. sets	**100%** 79.2–84%	**statist. sets** determin. sets	**97.6–98.8%** 50–78.8%	55.8–83.8%
FCT4, 587 seeded faults	**statist. sets** determin. sets	**99–99.1%** 82.1–85.5%	**statist. sets** determin. sets	**97.4–99%** 80.7–88.9%	89.6–91.7%

Fig.3 Scores of the statistical structural test sets – comparison with deterministic and random testing.

FCT4 is the only case for which no statistical set supplies 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 data. This result confirms the efficacy of a *mixed test strategy* [Thévenod 89] combining: (1) a global probe by statistical testing and (2) deterministic testing of extremal values.

17 faults seeded in FCT3 are 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 are quite satisfactory (98.8–100% revealed). The Class 6 sets of FCT4 are also efficient: they reveal 98.5-100% of the faults not related to extremal values. 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).

4.3. Experimentation with Statistical Functional Testing

Statistical testing may also be based on *behaviour models* deduced from the specification of the whole component. Using a top-down modelling process, a hierarchy of models is issued; then retaining *weak criteria* and deriving *several input profiles* – each being focused on a subset of models – is the general approach that allows us to manage complexity.

Our experimentation was first based on a multi-level description combining finite state machines and decision tables [Thévenod 92]; then, we applied the approach to behaviour models produced in the STATEMATE™ environment [Thévenod 93].

4.3.1. Design of Statistical Test Sets from Finite State Machines and Decision Tables

Finite state machines (FSMs) and decision tables (DTs) – see e.g. [Davis 88, Beizer 90] – possess complementary features: FSMs are well-suited to describe sequential behaviours, while DTs focus on the combinational parts. The description of the component functionality involved three levels of model:

(1) the FSM M_0 (12 states, 88 transitions) identifies the current operating mode from full to minimal service. Each mode determines a number of measures to be processed.

(2) the FSM M_1 (12 states, 54 transitions) models the checks and filtering performed on one measure. Each transition induces the acceptance or rejection of the measure.

(3) the DT M_2 (8 rules) describes the rod position to deliver for each accepted measure.

The coverage of *DT rules* and of *FSM states in steady-state conditions*, i.e., after a number of initial executions large enough to ensure that the transients die down, are the criteria that we retained. Since M_1 and M_2 both relate to the processing of one measure, it was decided to cover them with the same input profile. Hence, two distinct profiles have been designed: the first one for the low level models (M_1, M_2); the second one for M_0. This was carried out by analysing the equations governing the state activation (transition matrix of the FSMs) and the DT rule selection [Waeselynck 93].

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

To reach the steady-state conditions with a precision of 10^{-6} and get $Q_N = 0.9999$, 85 inputs are required under the first profile, and 356 inputs under the second one. Five different statistical test sets of size N = 85 + 356 = 441 have been generated.

4.3.2. Design of Statistical Test Sets from a STATEMATE Specification

STATEMATE [Harel 90] is a tool for the specification of complex reactive systems. It supports a hierarchical modelling approach, each level of hierarchy consisting of both a functional and a behavioural view. The behaviour views are *Statecharts* [Harel 87], a graphical language that improves state diagrams by adding encapsulation, concurrency and broadcast communication. Simulation and instrumentation facilities are provided.

The specification developed with the aid of the tool involves two levels of hierarchy, the *top level function* and *6 subfunctions*, one for each of the five interface cards (checks and filtering of the data acquired), and one for the conversion of the data into rod positions:

- the Statechart of the top-level view has *33 basic states* (states without offspring);
- the six Statecharts of the low level involve a total amount of *299 basic states*.

It was decided to design two input profiles, one for each level of hierarchy, and to ensure the *coverage of the basic states*. Also, an improvement on the test sets previously defined (4.3.1) was to *stress the interactions between high and low level functions*. Since a major function of the top level is to modify the status of the card processing activities, it was focused on the transient period that follow the reset of low level functions (states covered during the first acquisition of data).

In the present state of the art, it is not possible to determine the equations governing the state activation of Statecharts. We thus had to proceed *empirically* to search for proper profiles, the models being instrumented so as to collect state coverage measures. Five test sets have been generated under the new profiles. Their size has been kept the same as previously (85 + 356 items): each basic state is exercised at least 11 times by any set, hence providing the target test quality of 0.9999.

4.3.3. Experimental Results on two Versions of the Component

Figure 4 shows the results supplied by the various test sets: *Functional 1* and *Functional 2* denotes respectively the statistical sets described in 4.3.1 and 4.3.2; *Random* denotes the large random set of 5300 data already commented on in Section 2. Thirteen faults have been identified in which 12, denoted A, B, ..., L, were found in the student program; the last one, Z, is a minor fault residing in the real version: in the real system, it would never lead to failure due to systematic hardware compensation. Faults A, G and J are *structural faults* directly linked to the coding activity. Faults B to F, and I, result from the *lack of understanding of the filtering check requirements* by the student, this function being at the heart of the component. The others are *initialisation faults*: either an improper initial value is assigned (K, L) or the initialisation is missing (H, Z).

	A	B	C	D	E	F	G	H	I	J	K	L	Z
Random	✔	—	—	✔	—	✔	✔	✔	—	—	—	—	—

Functional 1	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	—	4
Functional 2	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

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

Fig.4 Real faults found by the functional test sets

The use of behaviour models as guides for designing statistical testing is relevant to fault exposure. Since the experiments involved real (unknown) faults, it is not possible to measure the exhaustiveness of the sets with respect to them; but it can be seen that both types of statistical sets perform *repeatedly much better than the random set*, although they are one order of magnitude shorter. The two profiles defined in each case provide complementary results: the subsets of 85 and 356 data do not reveal the same faults. Moreover, each entire statistical set exercises 100% instructions of the two programs.

The fault Z was not revealed by one Functional 1 set: this motivated our choice to stress the reset interactions in the Functional 2 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 acquisitions of data after a reset. During the design of the Functional 2 profiles, only the *first* acquisition was taken into account for the coverage of the initial states; in spite of this, it allowed us to raise significantly the failure probability of the program. This result confirms once again the *efficiency of statistical testing with respect to faults that are loosely connected with the criteria retained*.

Finally, it is worth noting that the high fault exposure power of the Functional 2 sets, that were derived empirically, bodes well with regard to the feasibility of statistical functional testing for complex behaviour models.

Conclusion

Because the probabilistic approach is generally related to uniform test data, it is often deemed a poor methodology: our experimental studies remove this preconceived idea.

Applied to the verification of a non trivial software component from the nuclear field, statistical testing designed from structural or functional models supplied repeatedly a high fault exposure power within a realistic test duration. At the unit level, the efficiency of the method was assessed referring to 2816 faults of mutation type. The experiments at the whole component level involved a small sample of real faults that were a priori unknown. Note that complementary results reported elsewhere [Thévenod 94a] show that the statistical test sets designed at the component level are also efficient in revealing a large sample of seeded faults (6175 mutations).

A limitation of the statistical sets experimented on is 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 data specifically aimed at them. Hence, we support the adoption of a **mixed testing strategy** involving both statistical and deterministic test

sets; such a mixed strategy having already been successfully experimented with at the unit level [Thévenod 94b].

The results related to unit testing confirmed the insufficiency of deterministic test data derived from current – imperfect – criteria. For larger scale programs, this limitation is expected to get worse. However, the medium size of the component experimented on does not yet allow us to come to a conclusion about the feasibility of statistical testing for complex software. Two problems arise:

- The oracle problem, namely that of *how to determine the correct output results a (complex) program should return in response to given input data*. This problem arises with any dynamic verification technique; but it becomes crucial when numerous responses to probabilistic inputs are concerned.

- The complexity of the probabilistic analysis required by statistical testing.

Fortunately, an answer to these problems is likely to reside in the recent emergence of CASE tools that assist software development by supporting formal models for the specification of behaviour, and offering facilities to computerised simulation. Such tools provide us with an **oracle**; the possibility of instrumenting the models facilitates the **empirical derivation of test profiles** proper to ensure a rapid coverage of the software functionalities.

In Section 4, the benefit of using CASE tools for the design of statistical testing has been exemplified on the STATEMATE environment. The promising results obtained for the medium-scale component may constitute a first step along the road.

Acknowledgements

This work was supported in part by the CEC under ESPRIT Basic Research Actions no. 3092 & 6362: "Predictably Dependable Computing Systems" (PDCS & PDCS2). We wish to thank our colleague, Yves CROUZET, for his useful contribution to the experiments in unit testing.

References

[Beizer 90] B. Beizer, *Software testing techniques*, Van Nostrand Reinhold, New York, Second Edition, 1990.

[Davis 88] A. M. Davis. "A comparison of techniques for the specification of external system behaviour," *Communications of the ACM*, vol. 31, no. 9, pp. 1098-1115, 1988.

[DeMillo 78] 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.

[Duran 84] 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.

[Hamlet 89] R. Hamlet, "Theoretical comparison of testing methods," in *Proc. 3rd IEEE Symposium on Software Testing, Analysis and Verification (TAV-3)*, Key West, USA, pp. 28-37, December 1989.

[Hamlet 90] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Trans. on Software Engineering*, vol. 16, no. 12, pp. 1402-1411, December 1990.

[Harel 87] D. Harel, "Statecharts : a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231-274, 1987.

[Harel 90] 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.

[Howden 87] W. E. Howden, *Functional program testing and analysis*, Computer Science Series, McGraw-Hill Book Company, 1987.

[Myers 79] G. J. Myers, *The art of software testing*, Wiley, New York, 1979.

[Ntafos 81] S. C. Ntafos, "On testing with required elements," in *Proc. COMPSAC'81*, pp. 132-139, November 1981.

[Rapps 85] 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, 1985.

[Thévenod 89] P. Thévenod-Fosse, "Software validation by means of statistical testing: retrospect and future direction," *Preprints 1st IEEE Working Conference on Dependable Computing for Critical Applications (DCCA-1)*, Santa Barbara, USA, pp. 15-22, August 1989. Published in *Dependable Computing and Fault-Tolerant Systems*, vol. 4 (Eds. A. Avizienis, J-C. Laprie), Springer-Verlag, pp. 23-50, 1991.

[Thévenod 91] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet, "An experimental study on software structural testing: deterministic versus random input generation," in *Proc. 21st IEEE Symposium on Fault-Tolerant Computing (FTCS-21)*, Montréal, Canada, June 1991, pp. 410-417.

[Thévenod 92] P. Thévenod-Fosse, H. Waeselynck, "On functional statistical testing designed from software behaviour models," *Preprints 3rd IEEE Working Conference on Dependable Computing for Critical Applications (DCCA-3)*, Palerme, Italy, pp. 3-12, September 1992.

[Thévenod 93] P. Thévenod-Fosse, H. Waeselynck, "STATEMATE applied to statistical software testing", *Proc. International Symposium on Software Testing and Analysis (ISSTA 93)*, Cambridge, Massachussets, June 1993.

[Thévenod 94a] P. Thévenod-Fosse, Y. Crouzet, "On the Adequacy of Functional Test Criteria based on Software Behaviour Models", LAAS report no. 94.192; To appear in *PDCS2 (Predictably Dependable Computing Systems, Esprit Basic Research Action no. 6362) 2nd Year Report*, September 1994.

[Thévenod 94b] P. Thévenod-Fosse, C. Mazuet and Y. Crouzet, "On statistical structural testing of synchronous data flow programs," LAAS report no. 93.282; To appear in *Proc. 1st European Dependable Computing Conference (EDCC-1)*, Berlin, Germany, October 1994.

[Waeselynck 93] H. Waeselynck, "Vérification de logiciels critiques par le test statistique", Doctoral Thesis, Institut National Polytechnique de Toulouse, LAAS report no. 93.006, January 1993.

[Weyuker 91] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. on Software Engineering*, vol. 17, no. 7, pp. 703-711, July 1991.