ON FUNCTIONAL STATISTICAL TESTING DESIGNED FROM SOFTWARE BEHAVIOR MODELS

Pascale THÉVENOD-FOSSE, Hélène WAESELYNCK
Laboratoire d'Automatique et d'Analyse des Systèmes du C.N.R.S.
7, Avenue du Colonel Roche, 31077 Toulouse Cedex - France

Abstract

Statistical testing involves exercising a piece of software by supplying it with input values that are randomly selected according to a defined probability distribution over its input domain. This paper focuses on **functional statistical testing**, that is, when an input distribution and a number of random inputs are determined according to criteria relating to software functionality. The criteria based on **models of behavior** deduced from specification, i.e., finite-state machines and decision tables, are defined. The modeling approach involves a hierarchical decomposition of software functionality. It is applied to a module from the **nuclear field**. Functional statistical test sets are designed and applied to two versions of the module: the real version, and that developed by a student. **Twelve residual faults** are revealed, eleven of which affect the student's version. The other fault is quite subtle, since it resides in the driver that we have developed for the real version in our experimental test harness. Two other input distributions are experimented with: the uniform distribution over the input domain and a structural distribution determined so as to rapidly exercise all the instructions of the student's version. The results show that the functional statistical test sets have the highest fault revealing power and are the most cost-effective.

Keywords: Software testing, functional criteria, random test inputs, theory, experiments.

1. Introduction

Testing involves exercising the software by supplying it with input values. In practice, testing is partial as it is not possible to exercise a piece of software with each possible data item from the input domain. Hence, the problem of selecting a subset of the domain that is well-suited for revealing the actual but unknown faults; this issue is further compounded by the increasing complexity of real software systems. Many test criteria, relating either to program structure or software functionality, have been proposed as guides for determining test cases (see e.g. [1, 10, 12]).

Using these criteria, the methods for generating the test inputs proceed according to one of two principles [11]: either deterministic or probabilistic [5, 7, 16]. In the first case, which defines **deterministic testing**, test data are predetermined by selection in accordance with the criteria retained. In the second case, which defines **statistical** (or **random**) **testing**, test data are generated according to a defined probability distribution over the input domain; both distribution and number of input data items being determined in accordance with the criteria retained.

Some previous work focused on **structural statistical testing** [17], in which input distributions were determined according to structural criteria defined in the deterministic approaches [13, 15]. These distributions, called *structural distributions*, aim at ensuring that the program structure is properly scanned during a test experiment. Structural statistical testing has been shown to be an efficient way of designing test data during a unit testing phase, i.e. for programs with a known structure which remains tractable [18]. As a result, the *functional* statistical testing approach investigated in this paper is mainly concerned with but not confined to larger software components, that is, modules integrating several unit programs.

Functional statistical testing consists of determining an input distribution as well as a number of random test cases according to criteria based on software functionality. The criteria must facilitate the determination of input distributions, referred to as *functional distributions*, which will ensure that the different software functions are well probed within a reasonable testing time. This paper presents a **rigourous method** for designing functional statistical testing, based on **criteria related to behavior models**, as deduced from software specification.

Section 2 recalls the notion of *test quality with respect to a criterion*, from which the method for designing statistical test sets according to a given criterion is stated. **Section 3** deals with *behavior models* that facilitate the definition of *functional criteria*. It lays the foundation of a structured method for determining functional test data. The approach is exemplified by a case study in **Section 4**: statistical test data are defined from behavior models deduced from the specification of a *safety critical module* from the nuclear field. **Section 5** gives the *experimental results* that support the efficiency of these test data in revealing faults. *New areas of research* in functional statistical testing are described in **Section 6**.

2. Background

Previous work has shown that statistical testing is a practical verification tool. Indeed, the key to its effectiveness is the derivation of a probability distribution over the input domain that is appropriate to the test objective. The theoretical framework recalled below induces a rigourous method for determining these distributions.

2.1. Basic framework

Test criteria take advantage of information on the program under test in order to provide guides for selecting test cases. This information relates either to program structure or its functionality. In both cases, any criterion specifies a set of elements to be exercised during testing. Given a criterion A_i, let S_{Ai} be the corresponding set of elements. (To comply with finite test sets, S_{Ai} must contain a finite number of elements that can be exercised by at least one input item.) For example, the structural testing approach called "branch" testing requires that each program branch be executed: A_i = "branches" $\Rightarrow$ S_{Ai} = {executable program edges}. The notion of *test quality with respect to a criterion,* firstly defined for random inputs only [16], has been generalized as follows to be applicable to any test set irrespective of whether or not the inputs are deterministic or random [20].

Definition. A set T of N input data items covers a test criterion A_i with a probability q_N if each element of S_{Ai} has at least a q_N probability of being exercised during the N executions supplied by T. $\mathbf{q_N}$ is called the **test quality with respect to $\mathbf{A_i}$**.

In the case of **deterministic testing**, the tester selects a priori a number N of inputs such that each element of S_{Ai} is exercised at least once thereby providing in essence a "perfect" test quality ($q_N = 1$) with respect to A_i. It is worth noting that deterministic sets are often built so that each element is exercised *only once*, in order to minimize the test size (number of input items). In the case of **statistical testing**, a finite number of random inputs can never ensure that each element of S_{Ai} is exercised at least once, since no data specifically aimed at exercising these elements have been intentionally included in the test set; hence, $q_N < 1$. The test quality q_N provided by a statistical test set of size N is deduced from the following theorem [16].

Theorem. In the case of statistical testing, the test quality q_N with respect to a criterion A_i and the number N of input data items are linked by the relation:

$$(1-P_i)^N = 1-q_N \qquad \text{with } P_i = \min \{p_k, k \in S_{Ai}\} \qquad (1)$$

p_k being the probability that a random input exercises the element k of S_{Ai}.

Relation (1) is easy to justify: since P_i is the probability per input case of exercising the least likely element, each element has a probability of at least $1 - (1-P_i)^N$ of being exercised by a set of N random input cases. The result of this is that on average each element of S_{Ai} is exercised several times. More precisely, relation (1) establishes a link between the test quality and the number of times, denoted n, the least likely element is expected to be exercised: $n \cong - \ln(1-q_N)$, whatever the value of P_i. For example, $n \cong 7$ for $q_N = 0.999$, and $n \cong 9$ for $q_N = 0.9999$.

The **method for determining a statistical test set** according to a criterion A_i is based on the preceding theorem. It involves two steps, the first of which is the corner stone of the method. These steps are the following:

(i) *search for an input distribution* which is well-suited to rapidly exercise each element of S_{Ai} to decrease the test size; or equivalently, the distribution must accommodate the highest possible P_i value;

(ii) *assessment of the test size N* required to reach a target test quality q_N with respect to A_i, given the value of P_i inferred from the previous step; relation (2) deduced from relation (1) yields the minimum test size:

$$N = \log (1-q_N) / \log (1-P_i) \qquad (2)$$

An acute question arises from the **definition of test criteria**: a real limitation is due to the imperfect connection of the criteria with the actual faults. Due to the current lack of an accurate model for software design faults, this problem is not likely to be solved soon. Nevertheless, the criterion does not influence random data generation in the same way as in the deterministic approach: it serves as a guide for defining an input distribution and a test size, but does not allow for the a priori selection of a (small) subset of input data. Hence, one can expect that the criterion adequacy with respect to faults should lead to a less acute problem in the statistical approach; and all the more so as several test cases are involved per element to be exercised and as these test cases are unbiased by human choice. Indeed, there is a meaningful link between fault exposure and random data: from relation (1), *any fault involving a failure probability* $p \geq P_i$ *per execution according to the chosen input distribution has a probability of at least* q_N *of being revealed by a set of N random inputs*. No such link is foreseeable as regards deterministic data; and this link should carry more weight than the warrant of a *perfect* test quality with respect to *questionable* criteria. Previous work on unit testing has already supported this assumption: the main conclusions recalled below justify our present investigation of functional statistical testing for larger software components.

2.2. On the fault revealing power of structural statistical testing

The first investigations focused on current structural criteria and the theoretical results were confirmed by experiments relating to the unit testing of four real programs from the nuclear field [18-20]. Path selection criteria were used [13, 15], each of them defining a proper set of (sub)paths to be executed. Given a program and a criterion A_i, the **method for determining an input distribution** according to A_i was applied as follows. The program flow graph analysis provides the set S_{Ai} of (sub)paths, together with their execution probability p_k according to probabilities of input values. Then, an input distribution that lets $P_i = \min\{p_k\}$ be as high as possible is inferred by solving the equation set $\{p_k\}$ (see examples in [16, 18]). The structural statistical test sets thus designed ensure that the program structure is soundly probed, the level of probing being an increasing function of the criterion stringency (from "instruction" to "path" level): the more stringent the criterion, the larger the test size.

In the **experiments**, the target faults were mutations [4], seeded in the source code. The four programs were subjected to several structural deterministic, structural statistical and uniform statistical test sets; the latter sets are the

'conventional' random test sets, i.e., when data are drawn from a uniform distribution over the input domain. The efficiency of the sets was assessed in terms of percentage of mutation faults revealed, called mutation score. A total of 2816 mutations was involved, and the results were in favor of structural statistical testing for two reasons:

(i) *structural statistical test data rapidly uncovered 99.8% of the seeded faults;* both structural deterministic data and uniform random data were far from exhibiting so good fault revealing powers, failing to reveal several hundreds of mutations;

(ii) *the mutation scores provided by the structural statistical test sets were repeatedly observed,* whatever the particular test sets generated according to a same structural input distribution [20]. On the contrary, in the case of deterministic testing, the test sets related to the same criterion exhibited quite disparate mutation scores and the most stringent criteria did not always supply the highest scores; similarly, the uniform random test sets provided various scores.

This work confirmed the fact that the effectiveness of deterministic testing and uniform testing depends heavily on the particular input values chosen [7], while the effectiveness of structural statistical testing does not; the two former approaches were never more efficient than the latter one. This supports the high fault exposure power of statistical testing, as expected at the end of Section 2.1. The comparison between the uniform and structural statistical sets showed that *the structural analysis of a program does provide a relevant information.* The weakness of the deterministic sets resulted more from a limited number of data that failed to compensate for the imperfect connection between structural criteria and faults, than to a strong inadequacy of the criteria.

Indeed, in the light of the experimental results, the impact of the criteria stringency was deemed not critical in the case of statistical testing: the conclusion was that the *most cost-effective approach* is to retain weak criteria facilitating the structural analysis needed to determine an input distribution, and to require a high test quality (say, 0.9999) with respect to them [20]. Finally, the efficacy of a *mixed test strategy* combining structural statistical testing and deterministic testing of special/extremal input values was fully confirmed: the six mutations not uncovered after completion of the structural statistical tests were changes

affecting the bounds of an array; and these are typical cases for extremal value testing, that are poorly catered for by statistical testing within short testing time.

As structural testing is only applicable to programs that lend themselves to a tractable analysis, this work has to include larger **modules integrating several unit programs**. This is the aim of the following sections.

3. Functional statistical testing

"The goal of functional testing of a software system is to find discrepancies between the actual behavior of the implemented system's functions and the desired behavior as described in the system's functional specification." [14]. Accordingly, relevant functional test criteria must facilitate the selection of an input distribution and a test size ensuring that the software functions, and their interactions, are properly scanned.

3.1. Functional test criteria

Functional testing approaches refer to different levels of software description: external specification [12, 14, 21], internal design [9], program code [9], or a combination of levels [6, 22]. Some of them are rather informal, based on a careful review of the documents relating to the chosen level. They facilitate the derivation of deterministic test cases that are assumed to be functionally sensible. Examples of such approaches are "equivalence partitioning", "boundary-value analysis" and "cause-effect graphing" [12]. Keeping in mind that the probabilistic approach calls for the study of the influence of the input distribution on the coverage of the chosen criterion, informal approaches are not convenient for our purpose.

On the contrary, **finite-state machines (FSM)** used for describing software behavior – see e.g. [1, 3] – possess properties that are well-suited for a probabilistic analysis. An FSM can be depicted by a graph having a finite number of states and a finite number of transitions. The principle consists in:

(i) associating one *state* with each mode of behavior;

(ii) weighting each *transition* with the input conditions that trigger it, and eventually with the action caused when the transition is made.

Different criteria A_i may be defined from an FSM depending on the stringency of the graph coverage: state coverage, transition coverage, or sequence of transitions coverage [21]. A criterion defines a finite set S_{Ai} of elements to be exercised: S_{Ai} = {graph states}, {graph transitions}, {sequences of graph transitions}, for the aforementioned examples. The influence of the input distribution on the value $P_i = \min \{p_k, k \in S_{Ai}\}$ is studied by replacing the input conditions that weight the transitions with their probabilities of occurrence in the distribution.

Although finite state machines are helpful to describe a large number of functions and their interactions, there are behaviors for which the FSM approach makes no sense [3]. A typical example is when actions depend on a combination of conditions, thereby causing an explosion of the number of states and/or transitions. Then, **decision tables (DT)** are another modeling tool that is well-suited for describing such behaviors – see e.g. [1, 3] – and for a probabilistic analysis. A DT defines a finite set of rules, each rule specifying the actions that take place when a specific combination of input conditions is met. A natural criterion A_i related to a DT is the rule coverage, that is: S_{Ai} = {rules}, and $P_i = \min \{p_k, k \in S_{Ai}\}$ where p_k is the probability that the rule k be activated. For a rule k, the replacement of the specified combination of conditions with its probability of occurrence provides the expression of p_k.

FSMs and DTs are **complementary modeling tools**: in essence, FSMs are appropriate to describe sequential behaviors while combinational functions are easier to translate into DTs . They form the basis of our modeling approach, that involves a hierarchical decomposition of software specifications.

3.2. Behavior modeling approach

In keeping with the goal of functional testing presented above, we opted for functions deduced from module **specification**, rather than implemented functions deduced from module design. Thus, the test cases should be more likely to expose design faults and can be defined early in the development process. Since a detailed specification analysis should determine a large number of functions that may not be described by any FSM or DT of reasonable size, the modeling approach is based on a **hierarchical decomposition** of the functionalities that proceeds from a **top-down** approach; thereby involving a sequence of models M_i ($i \geq 0$), each M_i being either an FSM or a DT.

First, high-level functions and their interactions are identified, providing M_0; then, the high-level functions are refined through other models M_1, M_2, etc. For example [1]: in telephony, two-level models are common, three- and four-level models are not unusual. At a given level i, the actions caused by an FSM transition or a DT rule may be the production of output results or the transfer of control to lower level models. Hence, the set of models forms a tree network, M_0 being the root. The decomposition stops when the functions are considered elementary with respect to the outputs they supply. The function granularity at level i is the result of a compromise between the complexity of M_i and the required number of further refinements, i.e., of models M_k, $k > i$. A single limitation applies to the models: the FSM graphs must be strongly connected so that no state becomes unreachable when increasing the size of the statistical test sets. Note that, although the decomposition suggested here is independent of the module implementation, design information could easily be taken into account.

The top-down approach is well-known and used in most current specification and design methods. But the definition of statistical test sets from a hierarchical decomposition of software functionality has never been investigated.

3.3. Design of statistical testing

The coverages 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. Hence, p+1 models $\{M_0, \ldots, M_p\}$ provide p+1 sets of elements (rules or states) that have to be exercised. Starting with the models M_i, the first step consists of replacing the input conditions that weight the FSM transitions and that enter the DT rules with their probabilities of occurrence as function of the input probabilities. The next two steps, described below, are those identified in the method for determining a statistical test set, namely (i) *search for an input distribution* and, (ii) *assessment of the test size*.

3.3.1. Search for an input distribution

A lot of DT rules and FSM states are likely to be derived from the p+1 models. Hence it would not be realistic to attempt intensive coverage of all of them at the same time. This is because of:

(i) *the module complexity;* when all the models are encompassed, it is difficult to assess the probabilities of the elements as numerous correlated factors are involved;

(ii) *the explosion of the test size;* even if these assessments are feasible and tractable 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.

To address this issue several distinct test sets are designed each one focusing on the coverage of a subset of models. To do this, one defines a **partition of the models M_i** into s+1 ($s \leq p$) disjoint subsets PS_j ($j = 0, \ldots, s$), each PS_j gathering one or several models of consecutive levels. For each PS_j, a specific input distribution can reasonably be derived, that maximises the stationary probability of the least likely element related to the models M_i grouped in PS_j. One gets s+1 input profiles, and a proper test size N_j will be assessed for each of them (§3.3.2).

Since only a subset of elements is taken into account to determine the input distribution specific to a given PS_j, some input variables may not be involved and as a result no probability is obtained for them. Hence, the information deduced from other partition subsets must be included to define a complete input profile. To accomplish this, **the inputs that are not involved at a given partition level j** are classified according to three types:

(i) *upper level inputs* (except for $j = 0$), conditioning the transfer of control to a model $M_i \in PS_j$ from the upper level models; their probabilities must provide the most likely transfers to M_i;

(ii) *lower level inputs* (except for $j = s$), taken into account in lower level models; their probabilities are set as defined from the corresponding PS_k ($k > j$);

(iii) *unrelated inputs*, for which a uniform distribution may be used.

In practice, the determination of the input distributions uses a **bottom-up** approach (from PS_s to PS_0) since, from (ii), the input distribution specific to a level j may be partly defined at lower levels.

3.3.2. Assessment of a test size N

A complete test set is composed of s+1 suites of test cases, involving $N = N_0 + \ldots + N_s$ test cases. It must provide a test quality q_N with respect to the selected criteria, i.e., FSM states and DT rules. Let P_j be the stationary probability of

exercising the least likely state or rule related to a partition subset PS_j (P_j is inferred from the input profile derived for PS_j). Since P_j is a probability in steady-state conditions, the assessment of the test size N_j specific to PS_j involves two factors:

(i) first, equation (2) yields the test size in *steady-state conditions*;

(ii) and second, this test size must be augmented with the number N_{j1} of initial state transitions needed to *ensure that the transients die down*, that is:

$$N_j = N_{j1} + \log (1-q_N) / \log (1-P_j)$$

4. Case study: a safety critical application

The real case study reported in the succeeding sections illustrates the feasibility of the proposed approach. The models M_i are derived from the high level specifications of the target module, whose main requirements are summarized below.

4.1. High level requirements of the target module

The module is extracted from a **nuclear reactor safety shutdown system**. It belongs to that part of the system which 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. Cards 1 to 4 each deliver data about four rod positions; these cards are all created in the same way and are hereafter referred to as *generic* cards. The 5th card delivers data about the three remaining rod positions as well as monitoring data; this card which is therefore processed differently is called the *specific* card.

At each operating cycle, one or more interface card may be declared inoperational: the information it supplies is not taken into account. This corresponds to a *degenerated operating mode:* only part of the inputs are processed. A card identified as inoperational remains in that state until the next reset of the system. In the worst situation all cards are inoperational and the module delivers a *minimal service:* no measure is provided and only routine checks are carried out.

Extensive hardware self-checking is used so that errors when reading a card are unlikely. Nevertheless, for defensive programming concerns, this case is specified: the application is stopped and has to be restarted.

After acquisition, the data are checked and filtered. Three checks are carried out: the corresponding rod sensor is connected, the parity bit is correct and the data is stable (several identical values must be read before acceptance). The stringency of the third check (required number of identical values) depends on the outcome of the preceding checks of the same rod. After filtering, the measurements of the rod positions (in Gray code) are converted into a number of mechanical *steps*. The result of data conversion may be a valid number of mechanical steps or an invalid number or two special limit values.

4.2. Functional decomposition of the specification

The hierarchical decomposition involves two FSMs (M_0, M_1), and one DT (M_2). In the simplest case an FSM transition condition is the occurrence of a specific input value, e.g. a 'read error'. It may also include more complicated expressions, e.g. the current value of a rod position has to be identical to the previous one. The action resulting from a transition is either an output result or a transfer of control to lower level models until an output result be determined.

4.2.1. First level of decomposition: finite state machine M_0

The first level of decomposition consists in describing the **various operating modes**, and in identifying the conditions that make the system switch on them. Twelve operating modes are identified (Figure 1). The transition conditions relate to four factors:

- A(i) : i generic cards switch from 'operational' to 'inoperational';
- B : the specific card switches from 'operational' to 'inoperational';
- C : an error has occurred when reading an operational card (initiating a restart);
- D : a reset is forced while all the cards are inoperational.

operating mode		**state label**
full service	all cards operational	1
partial service	1 generic card inoperational	2
	2 generic cards inoperational	3
	3 generic cards inoperational	4
	all generic cards inoperational	5
	specific card inoperational	6
	specific and 1 generic cards inoperational	7
	specific and 2 generic cards inoperational	8
	specific and 3 generic cards inoperational	9
minimal service	all cards inoperational	10
initialization	reset	11
initialization	restart following a read error	12

Figure 1. States of the finite state machine M_0.

M_0 involves 88 transitions. By way of example, Figure 2 shows the outcoming transitions of state 8.

The information considered at this first level is not sufficient to determine an output result: when at least one card is operational and read without error, we have to proceed with the decomposition and study the processing of the measures acquired.

A(i) : i generic cards switch from 'operational' to 'inoperational'.

C : read error.

D : reset forced while all the cards are inoperational.

• : logical connective AND

+ : logical connective OR

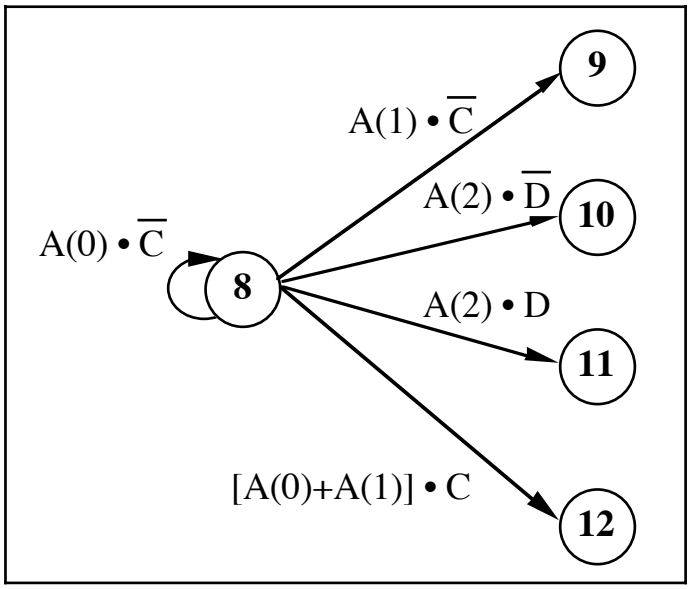

Figure 2. Outcoming transitions of state 8.

4.2.2. Second level of decomposition: finite state machine M_1

The second level of decomposition M_1 models the **checks and filtering performed on one measure**. As the number of measures acquired depends on the operational cards, the number of FSM M_1 running in parallel is determined by the state occupied in M_0. The machines M_1 are created, deleted, or maintained according to the transition made at the top level. For example (Figure 2), taking the transition '8 → 9' means that one generic card less is operational: as this card contains four measures, the four corresponding machines M_1 are deleted.

M_1 has twelve states: four of them are related to the stringent filtering mode, and the eight others to the normal filtering mode. The 54 possible transitions depend on three conditions:

- the sensor is connected;
- the parity bit of the measure is correct;
- the value of the measure is identical to the one read at the preceding operating cycle for the same rod.

It is worth noting that the latter condition will imply that successive test cases will not be selected independently within a statistical test set.

Each transition induces the acceptance or rejection of the measure acquired: twelve of them correspond to the case where a value passes the checks and has to be converted.

4.2.3. Third level of decomposition: decision table M_2

The **conversion** function, invoked by twelve transitions of M_1, is described by a DT with eight rules defining the rod position to deliver. The rule conditions involve:

- the value of the measure (valid or invalid number of mechanical steps, special values);
- a boolean input value specifying whether or not a special value is expected.

4.3. Behavior analysis

The previous analysis has identified attributes of input values that are significant for exercising the module functions and their interactions. The next step **assigns input probabilities**, in order to study the dynamic behavior of the module when subjected to a test profile.

The FSMs M_0 and M_1 weighted with symbols become **stochastic graphs**, i.e. graphs weighted with probabilities. For example, consider the condition for transition '8 → 9' (Figure 2): one generic card becomes inoperational (A(1)), and the remaining card is read without error ($\overline{C}$). Let:

- q be the probability that a generic card switches to inoperational,
- r be the probability that a read error occurs.

Then, the probability weighting transition '8 → 9' is: $\binom{2}{1}$ q (1-q) (1-r).

Once the transformation is completed for all transitions, the dynamic behavior is studied through operations on the transition matrices.

As regards the DT M_2, the probability of exercising each rule is directly obtained from the probability of its condition of activation.

4.4. Design of statistical test sets

Here a partition of the three models into two subsets PS_j is suitable: PS_0 contains M_0; PS_1 groups M_1 and M_2, that may easily be studied at the same time since both relate to the processing of one measure (while M_0 relates to the acquisition of a bundle of measures). Then, two distinct test sets must be designed: the first one will ensure the coverage of the filtering/conversion functions ($M_1 + M_2$); the

second one will probe the operating modes (M_0). Each set involves its proper input distribution and test size, informations deduced from the other level (upper or lower level input parameters) being incorporated when required.

The test sizes given below are assessed for a target test quality of 0.9999 (q_N, in steady state conditions) and for the requirement that the asymptotic state probabilities are reached with a precision of 10^{-6}.

4.4.1. Input distribution and test size to cover M_1 and M_2

Upper level parameters are forced to their activating values: a full service is delivered with no read error. Hence, the maximum number of measures are processed in parallel. The probabilities of the values appearing in the rule conditions of M_2 are determined so as to ensure a good balance between the rules. Then, an input distribution appropriate to cover the FSM M_1, that is, a distribution that maximizes the asymptotic probability of the least likely state, is investigated. By considering the small number of parameters that govern the process, an approximated solution may be obtained by **simulation, by sampling the relevant probabilities over [0...1]**.

The distribution obtained supplies a sufficiently high probability for the twelve transitions invoking M_2: each rule is exercised with the same (or higher) probability as the least likely state of M_1, i.e., approximately 0.0087.

For a given q_N, the test size in steady state conditions is derived from relation (2), and divided by the number of measures processed in parallel: for nineteen measures, 1055/19 = 56 inputs are required. This size is augmented with the number of initial transitions needed to ensure that the transients die down, namely 29 inputs, leading to: N_1 = 85 test inputs.

4.4.2. Input distribution and test size to cover M_0

As previously stated, the study of the dynamic state coverage allows us to assign optimum probabilities on the subset of inputs related to M_0. State coverage of M_0 in steady-state conditions requires N_0 = 356 test inputs (302 for state coverage, and 54 to reach the asymptotic state probabilities). To complete the input profile, the probabilities of lower level parameters, e.g. the values of the measures, are the same as in the preceding distribution.

5. Experiments and results

The experiments involve two versions of the module, providing us with a back-to-back testing scheme. REAL is the real version, and STU a version developed by a student from the same high-level specification; both versions are written in C language. The size of their object code approximates 20 K-bytes (a thousand lines of source code without comments). The experiment proceeds as follows: apply a test set to REAL and STU; examine the first output result for which a discrepancy is observed; identify and fix the corresponding fault(s). The process is iterated until REAL and STU agree on the whole test set.

5.1. Overview of the statistical test sets

5.1.1. Functional test sets

Section 4 aimed to show the practicality of deriving statistical test inputs from the behavior models of a non-trivial application. We now investigate the ability of these test data to expose actual faults, repeatedly whatever the particular values drawn from the defined input distributions: it is pointless to define a testing method whose efficiency depends heavily on the particular input values selected, rather than on adequate properties of the test data related to the method. Hence, in order to expose eventual disparities, **five different functional test sets F-Set$_i$** (i = 1, ..., 5) have been generated, each being composed of 85 inputs ensuring the coverage of M_1 and M_2 followed by 356 inputs ensuring the coverage of M_0.

5.1.2. Structural test sets

Although structural statistical testing is highly efficient in a unit testing phase, its relevance for larger scale programs, that is, when the complexity of the source code forces us to use only weak criteria such as branch or instruction coverage, may be questioned. One can wonder whether these criteria are sufficient to distinguish relevant input cases for a target module involving the aggregation of several functions. Another drawback of structural testing is that the selection of test data is driven by the source code, rather than by the specification: in particular, if two different versions of a same application have been designed, each of them requires its own test profile and test size.

The test sets used in the experiments are derived from the structure of the **STU version** only since few, if any faults are expected to reside in the REAL one. The complexity of the source code forces us to choose the weakest criterion, namely **instruction testing**, and to proceed empirically to derive an input distribution.

Starting from a large number of input data uniformly drawn from their valid range, we progressively refine the test profile until the frequency of an instruction is deemed sufficiently high (the C-compiler supports the automatic insertion of code to count the number of times each basic block of instructions is executed). At each step, the analysis focuses on a few blocks (the "hidden" ones) and aims at determining the input conditions that force their activation. The final input distribution is very different from the uniform one. A crude estimate of the probability of the least likely block is derived; given a high test quality requirement of 0.9999, an upper bound N = 500 on the test size is drawn from relation (2). **Five structural test sets S-Set$_i$** (i = 1, ..., 5) have been generated. It has been verified a posteriori that they provide a good coverage of STU (14 executions on average for the least likely blocks).

5.1.3. Uniform test set

As the notion of random patterns is often connected to a **uniform distribution over the input domain**, it was also used experimentally for comparison purposes. Selecting uniform patterns is a *black-box strategy*, the definition of the valid input domain being derived from the program specification. Actually, "blind" uniform testing is probably the poorest test strategy, since it does not take into account information relative to the target piece of software. In [18], the unit testing of four low level functions of REAL convincingly showed that a uniform distribution was far from adequate in most cases. The results are expected to be even worse when testing the whole module. A **single uniform test set**, denoted **U-Set**, has been generated involving a large number of test data, namely 5300 inputs: this is in conformity with the foundation of uniform testing, that is, large test sets generated cheaply.

5.2. Overview of the faults uncovered

Twelve faults have been identified (Figure 3) in which 11, denoted A, B, ..., K, were found in STU; the last one, Z, resides in our test driver that provides the interface between REAL and the files containing the test sets. Faults A, G and J are **structural faults**, directly linked to the coding of STU. 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 module. The others are **initialization faults**: either an improper initial value is assigned (K) or the initialization is missing (H, Z). The initialization faults are most subtle since their activation depends on the states that follow the wrong initialization. For example,

revealing H requires that G be removed and that the specific card be inoperational immediately after a restart or a reset. Finally, although Z resides in the test driver that we have developed for REAL, it has a ripple effect on the module: the simulation of hardware restart/reset fails to restore the correct initial context of REAL.

A	wrong operator used in the processing of an output value
B, C, D, E, F, I	the filtering checks, as implemented, do not comply with the specification
G	wrong control flow when the specific card is inoperational
H	initialization missing for variables related to the specific card
J	a variable in a loop is initialized out of loop instead of at each iteration
K	wrong initial state for the filtering process
Z	initialization missing for a variable of our test driver

Figure 3. List of the twelve faults uncovered.

Figure 4 summarizes the results supplied by the various test sets. The succeeding sections provide the main comments and conclusions related to each type of statistical test sets: uniform, structural and functional.

	A	B	C	D	E	F	G	H	I	J	K	Z
U-Set N = 5300	✔	—	—	✔	—	✔	✔	✔	—	—	—	—
S-Sets N = 500	✔	✔	✔	✔	✔	✔	2/5	1/5	✔	✔	✔	3/5
F-Sets N = 441	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	4/5

Figure 4. Results supplied by the test sets.

✔	always revealed
i/j	revealed by i sets out of j
—	not revealed

5.3. Inadequacy of the uniform distribution

As anticipated, uniform testing provides the **poorest results**, since it reveals only five of the twelve faults identified. The U-Set poorly probes STU and REAL, from both a structural and a functional viewpoint. With respect to the structural coverage, four blocks of instructions of STU and one of REAL are never exercised; some are seldom executed (less than three times). As regards the functional coverage, some M_1 states are seldom or never reached: most faults related to the filtering checks are not revealed.

In conclusion, a test data generation based on a uniform distribution is **definitely not efficient** to design a statistical test experiment. It is often argued that uniform testing gives a best return on investment than other approaches, since a large number of test cases can be generated cheaply. But, such data are unlikely to exhibit a good fault revealing power, so that little improvement is to be expected from a *realistic* increase of the test size. Here, the uniform set is an order of magnitude larger than the other sets: the five faults are found within the first 633 executions; the remaining 4667 executions being garbage.

5.4. Weakness of instruction testing

Nine faults are repeatedly revealed by all the structural sets; as they induce a high failure rate under the structural profile, the first quarter of the sets generally suffices to expose them. The other faults (G, H, and Z) are occasionally revealed, usually late in the test experiment. The case of Z is special, because the test sets have been designed to cover STU and Z corrupts REAL from its test driver. However, the S-Sets also provide a good structural coverage of REAL (14 executions on average for the least likely blocks, as for STU).

An interesting property observed for the five S-Sets is that the high instruction coverage is preserved throughout the debugging process, despite the fact that the structure of STU is modified by successive corrections. The problem with structural criteria is that the test data are program dependent: strictly speaking, a new analysis should be conducted for each intermediate version of the program, in order to adapt the test inputs to the evolution of the source code. It seems here that the probabilistic approach allows us to circumvent the problem.

As the structural profile ensures a **suitable probe of the models M_1 and M_2**, the S-Sets are very efficient for faults in the filtering checks. On the other hand, they provide **poor coverage of M_0**: most of the time the system resides in state

1 (full service delivered); the theoretical probability of ever reaching state 10 (minimal service) is only 4.10^{-4} for N = 500. A fault linked to the degeneration of the operating mode would not be uncovered. The specificity of the fifth card is not identified in this distribution. This accounts for the poor results for G and H, while both faults are rapidly revealed by all functional sets. With 500 structural data, the probability of revealing G is about 0.49, and the exposure of H requires that G be fixed. It can be said that some major module features are hidden in the implementation structure: actually, they correspond to family of subpaths in the control flow graph that are invisible at the instruction level. This problem did not occur when testing involved small units of REAL [20]: then, selecting weak criteria was deemed as the most cost-effective approach for the design of structural statistical testing.

In conclusion, structural testing is particularly **well-suited in unit testing**, but its effectiveness diminishes as the source code under test grows. The functions supported by the software move away from the instruction level, while finer examination of the structure becomes intractable. Indeed the size of the module under test is a limit above which instruction testing itself is no longer tractable. Moreover, this module belongs to the broad class of reactive systems, and whether or not the static graph of control is a relevant model for such systems is debatable.

5.5. Promising features of functional testing

The five functional test sets yield the best results, despite the fact that they involve the smallest number of inputs (441 versus 500 and 5300). Every fault in STU exposed by some S-Set is repeatedly found by all the F-Sets. The exposure of Z is less accurate, since Z is not revealed by one functional test set, but the result is still better than the one observed for the structural sets.

The **two input distributions involved in the F-Sets exhibit complementary features**: the first subsets of 85 test data reveal the faults related to the bad processing of the measures, notably during the filtering checks, while the second subsets are more appropriate for faults related to the specific card and for initialization faults. Hence, the behavior models constructed from the specification appear to be meaningful with respect to the faults. Full instruction coverage is provided by each F-set and for each intermediate version supplied by the fixes, although slower than during structural testing.

In the light of our experiments, functional statistical testing seems to exhibit promising features that justify the continuation of the investigation. Now, the question arises as to whether the F-Sets, as designed, ensure a suitable probe of the **interactions** between high and low level functions. When defining an input distribution for the coverage of M_0, we have included probabilities deduced from the lower behavior models. But as Z is not revealed by one set this may be insufficient: the exposure of Z depends on conditions involving states of both M_0 and M_1, and these conditions are never fulfilled in F-Set$_2$. Hence, the testing of the interactions will be addressed in our new research work.

6. Conclusion

Because statistical testing is generally related to uniform test data, it is often deemed inadequate for the exposure of faults. Our study removes this preconceived idea. The random inputs have to be designed by using some model of the target program, whether structural or functional, as investigated in this paper.

For large or even medium scale programs, the model complexity forces us to use weak criteria, e.g. instruction or state coverage. The results reported here suggest that, as soon as one shifts from unit to integrated module testing, the functional approach is the most efficient: it allows us to distinguish the important features of the module early in the development process, while still providing a good coverage of the implemented code. The models used to depict the expected behavior are conventional: finite state machines and decision tables are available for any program specified according to current SA/RT techniques. Hence the approach is consistent with modern trends in software development, these trends being reflected by the increasing popularity of the CASE tools that support these techniques.

As a result the CASE tool **Statemate** [8] will be used as it offers interesting facilities that will help us to refine the approach: the behavior description is more powerful than FSMs; simulations can be programmed according to a chosen input distribution, so that statistics are easily gathered on the models; and these statistics should provide us with a significant assistance during the basic step of our approach, that is the search for a proper input distribution. Moreover, the high simulation power of Statemate should address the *oracle issue*, namely how

to determine the correct output which a program should give in response to a given input [2, 23].

Emphasis will be placed on the study of the interactions between the various levels of decomposition. We will also look at practical ways to identify extremal/special values from the models. In essence such values have a low probability of occurrence during statistical testing and are more efficiently covered by deterministic inputs specifically aimed at them: the ultimate goal of the study is to define a **mixed test strategy** combining random and deterministic inputs. In our investigation, the efficiency of the test sets thus designed will be assessed relative to real faults (the twelve already found and maybe other residual ones), and a larger sample of seeded faults (mutations).

Acknowledgments

We wish to thank our colleagues Alain COSTES, Yves CROUZET, Jean-Claude LAPRIE and David POWELL for their helpful comments during the preparation of this paper. This work was supported in part by the CEC under ESPRIT Basic Research Action no. 3092: "Predictably Dependable Computing Systems (PDCS)".

References

[1] B. Beizer, *Software testing techniques*, Van Nostrand Reinhold, New York, 1983. Second Edition, 1990.

[2] D. B. Brown et al, "An automated oracle for software testing", *IEEE Transactions on Reliability*, Vol. 41, No. 2, June 1992, pp. 272-280.

[3] A. M. Davis, "A comparison of techniques for the specification of external system behavior", *Communications of the ACM*, Vol. 31, No. 9, Sept. 1988, pp. 1098-1115.

[4] R. A. DeMillo, R. J. Lipton, F. G. Sayward, "Hints on test data selection: help for the practicing programmer", *IEEE Computer Magazine*, Vol. 11, No. 4, April 1978, pp. 34-41.

[5] J. W. Duran, S. C. Ntafos, "An evaluation of random testing", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, July 1984, pp. 438-444.

[6] B. Goodenough, S. L. Gerhart, "Toward a theory of test data selection", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 156-173.

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

[8] D. Harel et al, "STATEMATE: a working environment for the development of complex reactive systems", *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 4, April 1990, pp. 403-414.

[9] W. E. Howden, "A functional approach to program testing and analysis", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 10, Oct. 1986, pp. 997-1005.

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

[11] J-C. Laprie (Ed.), *Dependability: Basic Concepts and Terminology*, Vol. 5 in the Series on Dependable Computing and Fault-Tolerant Systems, Springer-Verlag, Austria, 1992.

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

[13] S. C. Ntafos, "A comparison of some structural testing strategies", *IEEE Transactions on Software Engineering*, Vol. SE-14, No. 6, June 1988, pp. 868-874.

[14] T. J. Ostrand, M. J. Balcer, "The category-partition method for specifying and generating functional tests", *Communications of the ACM*, Vol. 31, No. 6, June 1988, pp. 676-686.

[15] S. Rapps, E. J. Weyuker, "Selecting software test data using data flow information", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, April 1985, pp. 367-375.

[16] P. Thévenod-Fosse, "Software validation by means of statistical testing: retrospect and future direction", *Proc. 1st IEEE Working Conference on Dependable Computing for Critical Applications*, Santa Barbara, USA, August 1989, pp. 15-22. Published in *Dependable Computing and Fault-Tolerant Systems*, Vol. 4, Springer-Verlag, 1991, pp. 23-50.

[17] P. Thévenod-Fosse, "On the efficiency of statistical testing with respect to software structural test criteria", *Proc. IFIP Working Conference on Approving Software Products*, Garmisch, Germany, Elsevier Science Publishers B.V., North-Holland, 1990, pp. 29-42.

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

[19] P. Thévenod-Fosse, H. Waeselynck, "An investigation of statistical software testing", *Journal of Software Testing, Verification and Reliability*, Vol. 1, No. 2, July-September 1991, pp. 5-25.

[20] P. Thévenod-Fosse, H. Waeselynck, Y. Crouzet, "Software structural testing: an evaluation of the efficiency of deterministic and random test data", LAAS Report 91.389, December 1991.

[21] H. Ural, "Formal methods for test sequence generation", *Computer Communications*, Vol. 15, No. 5, June 1992, pp. 311-325.

[22] E. J. Weyuker, T. J. Ostrand, "Theories of program testing and the application of revealing subdomains", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 3, May 1980, pp. 236-246.

[23] E. J. Weyuker, "On testing non-testable programs", *The Computer Journal*, Vol. 25, No. 4, 1982, pp. 465-470.