FROM RANDOM TESTING OF HARDWARE TO STATISTICAL TESTING OF SOFTWARE

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

ABSTRACT

Random or statistical testing consists in exercising a system by supplying to it valued inputs randomly selected according to a defined probability distribution on the input domain. Its efficiency with respect to physical fault exposure in hardware components has been evidenced in theoretical and experimental work. So far, the literature on software validation has shown little regard for statistical testing although – by essence – it should also exhibit good performances as a means for dependability with respect to design fault. The paper points out the specific role one can expect from statistical testing in a software validation process. The notion of test quality with respect to the experiment goal allows to adjust the testing time to a target test quality. Numerical results illustrate the strengths and limits of statistical testing as a software validation tool, in the present state of the art.

1. INTRODUCTION

A computer system is made up of hardware and software components and any residual fault in some component can cause errors which are liable to lead to system failures. To deal with the issue of revealing faults, several verification techniques have been proposed in work related to either hardware or software. Faced with the high and always increasing complexity of actual components, none of them allows to guarantee that a component – or a set of components, or finally a whole system – is fault-free after thorough verification phases. Hence, system validation calls for two complementary notions: 1) fault removal which involves verification, diagnosis and correction; 2) fault forecasting which concentrates on dependability assessment.

Testing is a dynamic verification technique in current use: the component under test is exercised by supplying to it valued inputs. The large input domains of actual components prohibit exhaustive testing: the test data have to be confined to small subsets of the whole input domains. Various criteria provide guidelines to determine test patterns. Concerning hardware testing, most studies focus on physical faults: criteria are fault models deduced from known possible physical changes according to the technology. For design faults, either in hardware or in software, the definition of accurate fault models is still a challenge as – ideally – it should comply with a model of human failures. Concerning software testing whose topic is only design faults, criteria relate to either the structure or the function of the program, few of them referring to (hypothetical) fault models.

Whatever the criteria, the methods of generating the test inputs proceed from two main principles: either deterministic, or probabilistic. In the first case, called deterministic testing, test inputs are predetermined by a selective choice according to the adopted criteria. In the second case, called random or statistical testing, test inputs are selected according to a defined probability distribution on the input domain, and both the distribution and the number of input data are determined according to the adopted criteria. The paper focuses on the probabilistic generation method, our goal being to point up its specific role in software validation.

Historically, random testing seems to have hold hardware community's attention first. Since the early 1970s, theoretical and experimental work has evidenced the efficiency of random patterns with respect to physical fault exposure. The rapid growth of unit complexity was the main reason of the intensifying interest shown in random testing, as the fault revealing power of (small) deterministic test sets decreases when one are faced with both a less accurate knowledge of physical changes and larger input domains. Concerning software, the same trend becomes now apparent: the increasing complexity of programs has given rise to a renewed interest in the probabilistic generation method since a decade, although few theoretical studies have already been performed in that direction. In our mind, a particular emphasis has to be placed on statistical testing in future work on software validation as it becomes a dual-purpose tool with regard to design faults*: depending on the input probability distribution, it can be turned into either a verification method, or a fault forecasting method to assess the future operational dependability. In the sequel, we use the term random testing for probabilistic verification with respect to physical faults, as usual in the hardware literature. To involve the complementary notion of dependability prediction, statistical testing will designate probabilistic validation with respect to (here software) design faults.

The paper is organized as follows. Section 2 briefly recalls work on random testing. It leads us to conclude in section 3 that, although design faults set additional problems, random inputs should also exhibit a high fault

* The same remark applies to hardware validation with respect to design faults. But it is beyond the scope of the present paper.
revealing power in regard to such faults. Then section 4 outlines a software validation strategy mixing statistical and deterministic test inputs. It is strengthened by first results on software verification in section 5, and software evaluation in section 6. The directions for future work proposed in section 7 aim at pushing back the present limits of statistical testing as a software validation tool.

2. ON RANDOM TESTING OF HARDWARE

Random testing with respect to physical faults has given rise to work whose application field ranges from small scale integrated circuits to logical boards7-15. Its fault revealing power has been shown which has led, for example, to use it in design for testability techniques2,16. Our purpose is not a survey of the related work. We only recall basic principles, illustrated with some results, in order to introduce the subsequent sections.

2.1. Principle

Hardware pseudo-random generators, in particular based on linear feedback shift registers, are easy to realize so that uniform as well as weighted random generation of input vectors is a simple and attractive process to solve the problem of test pattern generation. Figure 1a gives the principle of random testing. The reference unit is assumed to be free of physical faults. A simulator can act as a reference. Other implementation schemes are possible. For example, Figure 1b shows compact random testing: only a compact function of the output signals – called the signature of the unit2,17 – is observed and compared with the valid expected signature one has got off-line, from simulation or from a reference unit. Hence, with regard to physical faults, the output checking to state whether or not the produced results are correct does not set a practical problem.

Figure 1. Random testing of hardware units.

The question facing the user is how to assess the efficiency of random inputs in revealing faults? Depending on both the unit complexity and the fault model matching the technology, different methods are proposed to determine the number of input patterns N (test size) required to reach a given probability QN that a faulty unit fails during a test experiment. (1-QN) is the probability that an experiment wrongly states that a unit under test is fault-free. Both implementations in Figure 1 lead to the same calculations.

2.2. Test size assessment

For a given unit, a set F of faults which possibly affect the unit is deduced from a physical fault model. To each fault fi ∈ F corresponds a test size ni needed to reach the probability QN that fi causes at least one failure. The methods relate the value of N to the worst-case fault by taking N = max {ni}. Hence, ∀ fi ∈ F: either ni = N ⇒ fi has the probability QN to be revealed, or ni < N ⇒ fi has a probability higher than QN to be revealed. Nevertheless, it is not necessary to calculate the whole set of ni, which should be unrealistic due to the large sets F one has to tackle: in practice, the methods allow to simply find a small subset F0 ⊂ F containing the worst-case fault(s) so that the study is limited to F0.

For a given fault fi, ni depends on pi which denotes the failure probability per test pattern induced by fi. This leads us to distinguish two classes of methods. The first one groups methods proper to analyse units and fault sets so that each pi has – or is soundly approximated by – a constant value each time a test pattern is applied. In the second one, the failure probabilities depend on both the inputs and the unit internal state so that approximations by constant values are disallowed.

Class 1.

If the failure probability per test pattern of the unit under test has a constant value p for the fixed input probability distribution, the three parameters p, N and QN are linked by relation (1) which is easy to justify as (1-p)N is the probability of N successive correct outputs.

\[
(1 - p)^N = 1 - Q_N
\]

Hence relation (2) gives the minimum test size required to reach a probability QN of revealing the actual fault.

\[
N = \ln(1 - Q_N) / \ln(1 - p)
\]

Then, for a fault set F, relation (2) with p = min {pi} ∀ fi ∈ F: F0 ⊂ F provides the required test size with respect to F. It is the basic principle used to evaluate N for a wide range of components, from Small and Middle Scale Integrated7 (SSI, MSI) circuits to Large12 and Very Large9,10 Scale Integrated (LSI, VLSI) circuits.

Class 2.

In case of Random Access Memories13 (RAM) and logical boards14,15, a given input pattern may or may not lead to failure depending on the internal state of the unit when applying the pattern, i.e. on the preceding test inputs. Instead of evaluating an average (meaningless) failure probability pij for a fault fi, nj is directly got from an absorbing Markov chain Mij related to the faulty behavior induced by fi, then N = max {ni} ∀ fi ∈ F0 ⊂ F.
The absorbing state $D$ of $M_i$ is reached upon occurrence of the first failure due to $f_i$, and $n_i$ is the lowest number of inputs so that $D$ has a probability at least $Q_N$. For boards, results focus on microprocessor ($\mu$P) boards containing a Read Only Memory (ROM) for program storage. Starting with a large Markov model of the detailed behavior of a board, a reduction algorithm\textsuperscript{15} allows to create a simple chain from which $n_i$ is assessed.

### 2.3. Some results

Figure 2 briefly recalls some main features of the theoretical results according to the unit complexity. It shows that the more complex a component, the more remote the fault model (column 2) and the analysis method (column 3) from the physical level: referring to the detailed circuit implementation becomes unfeasible. For example\textsuperscript{10}, the faults in the register decoding function of the $\mu$P Motorola MC 6800 are of the type "one or more internal registers are accessed instead of (or in addition to) a given register"; the information available in the MC 6800 user's book is sufficient to provide a proper graph model of the data flow in the $\mu$P. For RAMS\textsuperscript{13}, the power of random testing is especially impressive: for complex faults (influences between three or more cells), the test sizes remain a linear function $0(n)$ of the cell number $n$ while deterministic testing algorithms either take higher test times, or have not yet been found due to the fault set complexity.

<table>
<thead>
<tr>
<th>UNIT TYPE</th>
<th>F = ${f_i}$ DEFINED AT (1)</th>
<th>ASSESSMENT OF TEST SIZE FROM</th>
<th>TEST TIME (2) FOR QN = 0.999</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinational SSI</td>
<td>line and gate levels</td>
<td>network design</td>
<td>&lt; 0.1 second</td>
</tr>
<tr>
<td>Sequential SSI, MSI</td>
<td>line and gate levels</td>
<td>fault-free &amp; faulty flow tables</td>
<td>&lt; 0.1 second</td>
</tr>
<tr>
<td>LSI, VLSI</td>
<td>register-transfer level</td>
<td>graph model of the data flow</td>
<td>MC 6800: 25 seconds</td>
</tr>
<tr>
<td>RAM $n$ words x 1 bit</td>
<td>cell level (influences)</td>
<td>Markov chains (faulty behaviors)</td>
<td>Example (3): 1 minute</td>
</tr>
<tr>
<td>Board ($\mu$P+ROM)</td>
<td>cell and register-transfer levels</td>
<td>Markov chains (faulty behaviors)</td>
<td>Example (3): 1 minute</td>
</tr>
</tbody>
</table>

Figure 2. A general picture of theoretical work.

(1) Identical fault sets are used in deterministic testing approaches.
(2) With a uniform distribution on the input domain.
(3) A real board with a $\mu$P MC 6800 and a 1K words by 8 bits ROM.

Experiments\textsuperscript{7,11,12} performed on specific random testers as well as on a universal Tektronix tester, confirm the theoretical results and the fault revealing power of random patterns. The efficiency is particularly strong for the $\mu$P MC 6800 (250 chips were tested\textsuperscript{11}) and for LSI circuits (1100 chips\textsuperscript{12}); in practice, the actual faults were revealed within short testing times, less than 0.3 second for the MC 6800. Some of these faults, not involved in the presumed fault set $F$, were not revealed by the – obviously less numerous – deterministic patterns. Any fault uncovered by the deterministic data was also exposed by the random inputs.

### 3. FROM PHYSICAL TO DESIGN FAULTS

As explained in subsection 3.1, the main justification for the use of the probabilistic approach in a validation process with respect to design faults is the imperfect knowledge of actual faults. But theoretical and practical problems related to such faults prevent from directly applying methods used for physical faults (§3.2).

#### 3.1. Statistical testing adequacy

From the results on random testing of physical faults, it arises that the more complex the component, the more effective – and then needed – the random inputs for two main reasons:

- a) the knowledge of all physical changes is less accurate so that fault sets $F$ from which test patterns are defined do not include all the possible actual faults (see e.g. MC 6800 and LSI circuits);
- b) moreover, the known physical changes provide larger sets $F$ which are not exhaustively tractable as required in deterministic testing (see e.g. RAMS), while the probabilistic approach allow to limit the detailed study to reasonable subsets $F_0 \subset F$.

Indeed, the deterministic approach involves a selective choice of test patterns proper to reveal the presumed faults $e_\in F$, and such inputs may or may not turn out to be adequate to expose actual – but not presumed – faults. On the contrary, random testing does not ensure that each fault from $F$ is exposed. But its practical efficiency is less dependent on the fault hypotheses from which only a test size is deduced, instead of specific inputs: any actual fault – presumed or not – is revealed with at least the required probability $Q_N$ provided that the failure probability it induces is not lower than the worst-case defined from $F$. Hence, the two methods of generation – deterministic and probabilistic – are to be thought of as complementary rather than competing verification techniques\textsuperscript{18-20}.

Concerning design faults, no accurate fault model is known, in the present state of the art, and any model of human failures is expected to involve large fault sets. Then, the points a and b above clearly argue in favor of a high efficiency of random inputs in revealing design faults. Another issue of this imperfect knowledge is that no verification technique allows to certify that a system is free of design faults. Hence, fault forecasting is needed to decide whether or not the dependability requirement is achieved, and the probabilistic method provides a scientific basis for making inferences from the testing to operational environments: test samples randomly drawn from an input distribution representative of the future usage allow to assess the system dependability with respect to design faults, before the system release and with a sound confidence level\textsuperscript{21-24}. It is not the case with the deterministic techniques as they tend towards a subjective selection of input data. This analysis leads us to support the software validation strategy proposed in section 4, in spite of the arising issues discussed below.
3.2. Problems

A look through section 2 evidences three main problems concerning software design faults: 1) the absence of a trustworthy reference in Figure 1, 2) the lack of sound fault models to relate the test size N to realistic – even if only presumed – worst-case faults, and 3) the large testing times which may correspond to N program executions according to the execution duration.

The first issue is the well-known oracle problem, namely how to determine the correct output results the program has to return in response to the test input data? This arises when executing testing assuming an average of 10 seconds per execution. The problem is even more acute when numerous responses to random inputs are concerned. Few tools are available for this challenge and current solutions are partial (imperfect) oracles.

The most used scheme is back-to-back testing similar to Figure 1a. Two (or more) program variants replace the unit under test and the reference unit, assuming that they have a high probability to be free of related faults. A discrepancy between the outputs indicates that one or several variants have failed, and the analysis is managed manually. A prototype can act as a variant. Obviously, it is harder to plan solutions referring to compact testing (Figure 1b). The definition of a compaction technique comes up against the question of the calculus precision so that a large – or even infinite – number of reference signatures should be involved. Hence, the probability that a program with failures provides an acceptable signature should drastically be increased.

To cope with the second issue, the "worst-case" principle can be generalized to relate to any fault-based or not – test-based or set of criteria as explained in section 5. qN, called the test quality with respect to the adopted criteria, is the probability to exercise the less likely (worst-case) element from the element set S = {S_i} to be exercised according to the adopted criteria. If n_i is the execution number needed to reach the probability qN to exercise S_i ∈ S, N = max{n_i} ∀ S_i ∈ S. Then, two classes of methods are to be investigated depending on whether or not each program execution involves a constant probability to exercise an element. It is noteworthy that a meaningful link with the actual fault revealing power of random test inputs thus defined exists yet: any fault uncovered with a probability at least qN provided that the failure probability it induces for the chosen input distribution is not lower than the probability to exercise the worst-case element deduced from S.

The last issue is the prohibitive testing times one may get from the test sizes. For example (Figure 2), 25 seconds for the MC 6800 corresponds to N = 6 300 000 instructions (25.10^4 instructions/second). For software, if the same number of N executions is required, each execution involving many instructions, one gets two year testing assuming an average of 10 seconds per execution! Hence, emphasis must be placed on the definition of an input distribution proper to the adopted criteria, i.e. which maximizes the worst-case element frequency.

4. A SOFTWARE VALIDATION STRATEGY

To exploit the distinct aptitudes of the deterministic and probabilistic methods, the validation strategy combines both types of test inputs, with the double aim of removing faults and assessing the dependability in operation. Three steps are necessary: the first two ones aim at fault removal, and the third one involves fault forecasting. For fault removal, different mix strategies have been suggested. Most of them, deterministic inputs are first applied and then enhanced by additional runs with random data. In others, random data precede deterministic ones. In our mind, the second ordering is most cost effective provided that an appropriate input distribution is used to generate random data. Then the three steps are the following ones:

Step 1: Statistical testing for fault removal.
Test data are randomly drawn from an input probability distribution which lowers the number N of executions required to achieve a target test quality with respect to the adopted criteria. A large number of faults is expected to be rapidly revealed.

Step 2: Deterministic testing for fault removal.
Additional deterministic inputs are selected to ensure the coverage of some extremal/special values whose occurrence probabilities remain insufficient in step 1.

Step 3: Statistical testing for fault forecasting.
Random test data are selected from an operational input profile which can vary depending on the user in case of a population of disparate users. Some residual faults may be uncovered and fixed. Then a quantitative assessment of a dependability measure – as it will be perceived by the user – is deduced.

The three steps can be used in different phases of a validation process, from unit level to system level, each phase focusing on a specific type of faults. For example, at the unit level each part of the software is separately exercised to verify whether or not it satisfies its detailed specification. Then, assuming that unit testing has been thoroughly performed, integration testing emphasizes on the way the different programs interact, ... Finally, at the system level, one checks whether or not the system works in accordance with the high level specifications.

Concerning fault removal in the first two steps, the criteria adopted to select test inputs must be proper to the validation phase, i.e. to the fault type the phase aims at tracking down. The deterministic inputs in step 2 are deduced from the analysis required to determine the random inputs in step 1 (input distribution and number of data). Section 5 gives an efficient way of designing the steps in a unit testing phase, by referring to structural test criteria. The third step, either at the unit level or at the system level (acceptance testing), is dual-purpose: a) uncovering residual faults whose failure probabilities are amplified by the user's profile; then b) evaluating the user-perceived dependability. Section 6 faces the issue of the quantitative assessment in an acceptance testing phase.
5. VERIFICATION

Few previous work refers to statistical testing as a verification tool. Its review has enhanced our confidence that statistical testing has great potential, and has evidenced that its efficiency is linked to the adequacy of the input probability distribution with regard to the test goal. Hence, our theoretical study - illustrated with experiments on four real life programs - aims at finding practical approaches to determine proper input distributions and test sizes according to the test phase. This section mainly concerns a unit testing phase. Integration testing is currently under investigation (§7).

5.1. Test size assessment

The test criteria defined in the literature relate to the structure or the function of the program, involving or not hypothetical fault sets. Each criterion specifies a set of elements to be exercised during testing, where elements and exercised have different meanings depending on the criterion. For example: branch testing means that each program branch has to be executed; mutation testing requires that each mutant is killed, i.e. each mutation is revealed. Let Si be the element set to be exercised according to a criterion Aj. Definitions 1 and 2 generalize the “worst-case” principle by linking the test size N to a criterion or a set of criteria.

DEFINITION 1.
A criterion Aj is covered with a probability qN if each element from Si has a probability at least qN to be exercised during N executions with random inputs. qN is the test quality with respect to Aj.

DEFINITION 2.
A criteria set A = {Aj} is covered with a probability qN if each criterion Aj from A is covered with a probability at least qN during N executions with random inputs. qN is the test quality with respect to A.

If nI is the test size needed to reach the probability qN of exercising sI ∈ Si, then Nj = max [nI] ∨ sI ∈ Si is the size required to get the test quality qN with respect to Aj. For a set A, N = max {Nj} ∨ A ∈ A. Obviously, the worst-case element sk (such as sk = max {ni}) and the value nk depend on the input distribution. Hence, a proper distribution must be determined according to the adopted criterion or criteria set in order to lower the test size.

The notion of test quality with respect to a criterion allows us to compare various criteria in regard to random data, by means of the R-subsumption relation.

DEFINITION 3.
Aj R-subsumes Ah if any set of random inputs which covers Aj with the probability qN also covers Ah with a probability at least qN, i.e. if at a same quality level qN: Nj ≥ Nh for any program.

A mutation is a simple change in the program code. A mutant is a copy of the program with a single mutation (seeded fault).

Aj is more stringent than Ah if it R-subsumes Ah, and the criteria stringency is coherently connected with the test size: the more stringent the criterion, the higher the test size, and the better the probability that a test experiment reveals actual fault(s).

5.2. Structural statistical testing

The preceding definitions are the basic framework of a study on the adequacy of current structural criteria to define random inputs. Such criteria apply to programs for which a structural analysis remains tractable which implies program control and data flow graphs of reasonable complexity. Then the results mainly apply to unit testing. Moreover, they are related to structural statistical testing, the input distribution and the test size being determined from the program structure. Many structural criteria are defined in deterministic testing approaches. Each of them defines a specific set of paths in the program graph to be executed.

In case of memoriless programs, each path i has a constant probability pi to be exercised by an input pattern (i.e. per execution). We are in the case of the methods of class 1 (§2.2). Relation (2) gives the test size for a target qN with respect to the adopted criterion Aj: QN is replaced by qN and p = min {pi} ∨ i ∈ Sj. The {pi} values are obtained from the analysis of the program design, by analogy with the network design analysis performed for combinational circuits (Figure 2). As for hardware components, the analysis allows to identify a subset S0 ⊂ Sj containing the worst-case element(s), and the assessment of the {pi} values is limited to S0.

For programs which retain data from one execution to the next - programs with memory - the execution probabilities pI cannot be soundly approximated by constant values. The assessment of the test sizes nI must take into account both the input data and the program internal state. Absorbing Markov chains, used in the hardware methods of class 2, should provide a relevant modeling tool by associating to a path i a chain with an absorbing state D reached upon occurrence of the first execution of i; then ni is the lowest number of executions so that the probability of D is at least qN. Applications to various programs are still necessary to confirm the practical use of this analysis method.

Fifteen criteria, defined in various papers, have been partially ordered according to the R-subsumption relation (Figure 3). The ordering is demonstrated in case of constant probabilities pi. But the proof can easily be generalized to provide the same result whatever the execution probabilities, constant or not. Figure 3 means, for example, that if a set of random inputs provides a test quality qN with respect to “All-Uses”, it provides a test quality at least qN with respect to the eight criteria below “All-Uses”. Obviously “All-Paths” is the most stringent criterion. The less stringent ones are “All-C-Uses”, “All-Defs” and “Instructions”, which are not comparable according to the R-subsumption relation. It is noteworthy
that the ordering is similar to the one induced by the subsumption relation\textsuperscript{34-36} defined as follows, in regard to deterministic inputs: $A_i$ subsumes $A_h$ if any test set which satisfies $A_i$ for a given program also satisfies $A_h$.

![Diagram of partial ordering of structural test criteria]

Figure 3. Partial ordering of structural test criteria. "$A_i \rightarrow A_h$" denotes the relation "$A_i$ R-subsumes $A_h$".

5.3. Some experimental results

Experiments achieved on four small programs from the nuclear industry – written in C language – confirm the high fault revealing power of structural statistical testing, and the need for additional deterministic test data to ensure the coverage of some special input values\textsuperscript{37}. The study aims at comparing: 1) the two methods of structural data generation, deterministic and random; 2) random data selected from an input distribution proper to structural criteria with the ones drawn from a uniform distribution (referred in other work\textsuperscript{18,28-30}).

To assess and compare the efficiency of various data sets with respect to fault exposure, mutation analysis\textsuperscript{28} is the chosen approach. A amount of 2816 non-equivalent mutants\textsuperscript{*} were created: each mutant contains a single-point, syntactically correct change. The seeded faults are representative of the most likely faults committed by programmers using the C language, involving the following types: constant replacement; symbol (scalar variable, array and structure reference, ...) replacement; operator replacement. Although these faults are not real faults, they should form a target fault set consistent with the goal of structural testing in a unit testing phase which should aim at tracking down faults directly linked to the program coding.

Mutation scores, defined as the fraction of the mutants which are killed, were assessed for various input sets. Figure 4 outlines results for three types of data.

1) Structural statistical inputs.

Proper input distributions are determined according to the most stringent achievable criteria, namely: "All-Paths" for three programs and "All-Uses" for the other one as it contains several millions of complete paths. The test sizes are defined for $q_k = 0.9999$: $N = 170, 100, 405$ or $850$ depending on the program.

2) Uniform statistical inputs.

Random data are selected from a uniform distribution on the valid input domain. For each program, the test size is chosen to be identical to the one of the structural statistical testing.

3) Structural deterministic inputs.

For each program several deterministic data sets are used. Each set contains the same number of inputs ($N = 17, 9, 19$ or $9$ depending on the program) and ensures the execution of all the elements to be exercised according to the most stringent achievable criteria. For a given program the different test sets exhibit various mutation scores. Figure 4 gives the highest and the lowest observed scores.

<table>
<thead>
<tr>
<th>TYPE OF TEST INPUTS</th>
<th>MUTATION SCORE</th>
<th># ALIVE MUTANTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural statistical</td>
<td>99.8%</td>
<td>6</td>
</tr>
<tr>
<td>Uniform statistical</td>
<td>75.7%</td>
<td>685</td>
</tr>
<tr>
<td>Structural deterministic:</td>
<td>88.9%</td>
<td>312</td>
</tr>
<tr>
<td></td>
<td>lowest efficiency</td>
<td>85.6%</td>
</tr>
</tbody>
</table>

Figure 4. Summary of experimental results involving 2816 mutants related to four real life programs.

5.4. Conclusions

The whole set of experiments focus on ten criteria from Figure 3: All-Paths, All-DU-Paths, All-Uses, All-C-Uses/Some-P-Uses, All-P-Uses/Some-C-Uses, All-C-Uses, All-Defs, All-P-Uses, Branches, Instructions. Two main conclusions arise from the detailed results\textsuperscript{33,37}.

a) Best efficiency of structural statistical testing.

Whatever the program, structural statistical testing provides the highest mutation scores. The more complex the program, the greater the difference between structural statistical and deterministic scores.

b) Weakness of structural deterministic testing.

No direct relation is observed between the stringency of a criterion and the mutation score. For example, for one of the four programs, the scores of eight data sets derived from "All-Uses" range from 0.52 to 0.83.
Furthermore, the most stringent criteria do not ensure the highest score: for a program, the best "All-Uses" data set reveals 84% mutants while the worst "All-Uses" data set reveals 95.4% mutants. As in the case of hardware testing (§3.1), the deterministic approach 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 with respect to fault exposure, even if a stringent criterion is adopted.

As the results evidence the efficiency of structural statistical testing with respect to mutation fault exposure, they argue in favor of its highest revealing power with respect to real faults in program codes. The six mutants not killed by structural statistical testing correspond to changes in an array index. Five of them can be revealed only by specific input patterns that have a probability 3.10^{-3} to occur in the whole data set. Hence, the need for a separate deterministic testing of extremal values (§4).

A practical way to process is to derive from the analysis of the adopted input distribution a deterministic data set ensuring the coverage of particular values of interest.

6. EVALUATION

When test inputs are randomly selected from an input distribution representative of the user profile, statistical testing is an experimental way to determine whether or not a software is ready to be used, i.e. whether or not it meets the dependability requirements. Most of the evaluation methods refer to dependability growth models which provide predictions from databases of failure reports related to several software versions owing to successive corrections. This section focuses on a complementary issue: the evaluation of a version from a test experiment with no information on the previous versions. Then, one must decide whether or not a product may be released without more fault fixing.

For reliability evaluation, two types of inference, related to the classical theory of hypothesis testing, have been proposed: either from a test experiment without observed failures or from failure records during testing. They supply a lower bound of the failure rate. A detailed analysis confirms the sound confidence level of such quantitative assessments, but at the expense of high required testing times. The second type applies only to systems one may release without fixing all known faults, provided that the reliability requirement is met. In particular, it excludes critical software. Furthermore, testing times are an increasing function of the number of observed failures. Hence evaluation from zero-failure experiments seems to us more promising in spite of its limits, recalled below, in the present state of the art.

When no failure occurs, the test size N required to accept the hypothesis that the actual failure probability p is less than or equal to a target value p_0, with a risk E_0 that the hypothesis is false (i.e. that p > p_0), is given by relation (2) with p = p_0 and 1-Q_0 = E_0. Q_0 becomes the confidence level in the assessed value. Assuming that the failure process is governed by a constant hazard rate, the evaluation in continuous time calls for a similar formula. The testing time T required to ensure that the software meets a failure rate \lambda_0 with a confidence level (1-E_0) is: T \geq \ln(E_0)/\lambda_0. It leads to results equivalent to those from relation (2), although expressed in a different unit of time, since for the practical values p_0: \ln(1-p_0) \approx p_0. For \lambda_0 = 10^{-3}/hour, T goes from 3 months (E_0 = 10^{-1}) up to 9.5 months (E_0 = 10^{-3}) and these times are multiplied by 10^x for \lambda_0 = 10^{-3} \times \text{hour}. Although the approach should be ideal to evaluate critical software, their requirements (10^{-6}, ..., 10^{-9}) clearly remain out of the scope of the current results. The previous formula can also be stated from Markov models – both in discrete and continuous time – of the software behavior during the test experiment. The models provide a tool well-suited to describe various situations, and extensions to the evaluation of two measures – reliability with respect to specific failures, and availability – have been formulated. They should be appropriate for further studies, outlined in section 7, which focus on testing time reduction.

7. RESEARCH DIRECTIONS

The key to statistical testing effectiveness is the determination of an input distribution relevant to the test objective. In particular, random inputs generated from a uniform distribution over the input domain are not expected to be an efficient way of designing statistical testing experiments, whatever their goal: as regards verification the revealing input data are unlikely to be uniformly distributed over the input domain, and as regards dependability assessment uniform testing does not allow to infer an operational behavior.

For verification, input distributions derived from current structural criteria properly scan the program structure, and have been shown to rapidly expose most of the faults in program codes during a unit testing phase. Making use of the same theoretical framework (§5.1), emphasis will be placed on the definition of functional criteria in order to design functional statistical test data for integration testing. Such criteria, possibly related to a behavior model derived from the software specifications – as in case of VLSI circuits –, must allow to deduce an input distribution and a test size ensuring that the various functions and their interactions are well probed.

Research work is needed to push back the limit of software evaluation from a "perfect working", today 10^{-3}/hour for reliability assessment within a reasonable testing time. Possible solutions involve: 1) several software copies exercised in parallel by statistically independent data, setting more acute problems for programs with memory; 2) test acceleration by modifying the input distribution without disturbing the confidence level of the evaluation.

The oracle problem must not be put aside. Future work will be directed towards formal specifications of the software behavior to investigate whether or not feasible solutions could emerge.
ACKNOWLEDGMENTS

I wish to thank many colleagues which have significantly helped me all along my research work in both hardware and software fields, especially Alain COSTES, René DAVID and Jean-Claude LAPRIE.

This work was supported in part by the CEC under ESPRIT Basic Research Action no. 3092: "Predictably Dependable Computing Systems (PDCS)".

REFERENCES


