An Evaluation of Random Testing

JOE W. DURAN, MEMBER, IEEE, AND SIMEON C. NTAFOS, MEMBER, IEEE

Abstract—Random testing of programs has usually (but not always) been viewed as a worst case of program testing. Testing strategies that take into account the program structure are generally preferred. Path testing is an often proposed ideal for structural testing. Path testing is treated here as an instance of partition testing, where by partition testing is meant any testing scheme which forces execution of at least one test case from each subset of a partition of the input domain. Simulation results are presented which suggest that random testing may often be more cost effective than partition testing schemes. Also, results of actual random testing experiments are presented which confirm the viability of random testing as a useful validation tool.

Index Terms—Partition testing, path testing, random testing, software testing experiments.

I. Introduction

In [10], Howden remarks that "... much has been written about structural testing, but little about black box testing, the method it was supposed to replace and over which it was supposed to be an improvement." He then reports on a successful refinement of black box functional testing, in which a detailed design "structure" is used to develop test cases. Another black box testing method which has not yet been adequately studied is random testing. There has been strong disagreement about its value. Myers [13] says "probably the poorest [test case design] methodology is random input testing . . .." On the other hand, Thayer *et al.* [14] recommend that the final testing of a program should be done with input cases chosen at random from its operational profile—the expected run time distribution of inputs to the program. Additionally, they showed how quantitative estimates of a program's operational reliability can be inferred from random testing. Girard and Rault [5] have also proposed random testing as a valuable test case generation scheme. Further, recent results [3] show that path testing, a popular paradigm for structural testing, can lead to less satisfactory reliability estimates than a corresponding number of random test case executions.

In this paper we present some simulation and experimental studies of random testing in order to investigate its effectiveness. In Section III we report on some simulation comparisons of random versus partition testing. In Section IV we investigate the error detection ability of random testing by applying it to a set of programs with known errors. Finally, in Section V we evaluate how well a set of programs is tested by random testing by using a variety of coverage measures such as the proportion of segments and branches executed by randomly generated sets of test cases.

Manuscript received October 1, 1982; revised September 30, 1983. This work was supported in part by the National Science Foundation under Grant MCS-8003322.

J. W. Duran was with the Southwest Research Institute, San Antonio, TX 78284. He is now with the Division of Mathematics, Computer Science, and Systems Design, University of Texas, San Antonio, TX 78285.

S. C. Ntafos is with the Computer Science Program, University of Texas at Dallas, Richardson, TX 75080.

II. Estimates of Error Finding Ability

Let θ be the probability that a program will fail to execute correctly on an input case chosen from a given input distribution. If the program is used for a long period of time with input from a particular operational profile (input distribution), then the failure rate actually experienced will converge toward θ. We thus refer to θ as the failure rate, which is a valuable measure of the operational reliability of the program.

Suppose the input domain D is partitioned into k subsets, $D_1, D_2, \cdots, D_k$, and that a randomly chosen input case has probability p_i of being chosen from D_i. Then $\theta = \sum_{i=1}^{k} p_i\theta_i$, where θ_i is the failure rate for D_i. The term "partition testing" refers to any test data generation method which partitions the input domain and forces at least one test case to come from each subset. Path testing is a special case of partition testing. D could also be partitioned according to alternative functions a program is supposed to perform. For example, a checking account program performs alternative functions depending upon whether the transaction being processed is a deposit, a withdrawal with sufficient funds, or a withdrawal with insufficient funds. Thus, partition testing contains instances of both black box and structural testing.

Thayer *et al.* [14] showed that if n random tests are carried out and x program failures are discovered, then θ^*, the $1 - \alpha$ upper confidence bound on θ, is the largest θ value such that

$$\sum_{i=1}^{x} \binom{n}{i}\theta^i(1-\theta)^{n-i} > \alpha.$$

(That is, there is a probability of $1 - \alpha$ that the calculated bound θ^* exceeds the actual θ value of the program. The θ^* value is, of course, valid only with respect to the input distribution used for test data selection.)

If $x = 0$, then $\theta^* = 1 - \alpha^{1/n}$. It has been shown [3] that if n test runs are made according to a partition testing scheme, and no failures are discovered, then, if there is no prior knowledge about the distribution of θ_i values, the resulting value of θ^* is bounded below by $1 - \alpha^{1/n}$. It is this result that led us to investigate the error finding ability of random testing experimentally.

Ideally, one might wish to characterize this error finding ability in terms of errors found per dollar, perhaps weighted by error severity. Being at present unable to characterize errors and their effects well enough to prepare any formal models for making "errors found per dollar" calculations, we pose two related questions.

1) What is the probability that a set of test cases will reveal one or more errors? (Finding a single failure instance is assumed to be equivalent to finding evidence of an error.)

2) What is the expected number of errors that a set of test cases will discover?

For random testing, the probability P_r of finding at least one error in n tests is $1 - (1 - \theta)^n$. For a partition test method in which n_i test cases are chosen randomly from each D_i, the probability of finding at least one error is given by

$$P_p = 1 - \prod_{i=1}^{k} (1 - \theta_i)^{n_i}.$$

With respect to the same partitioning,

$$P_r = 1 - (1 - \theta)^n = 1 - \left(1 - \sum_{i=1}^{k} p_i \theta_i\right)^n$$

where

$$n = \sum_{i=1}^{k} n_i.$$

In order to investigate the expected number of errors which will be discovered by partition testing and by random testing, we need a partition model which is both intuitively reasonable and mathematically tractable. For mathematical tractability, it is convenient to consider programs in which the domains of errors do not intersect. Certainly this covers a large class of programs, perhaps even the majority of well designed and constructed programs. For such programs, an ideal partitioning scheme would provide subdomains which contain at most one error each. Therefore we consider a partition model with the assumption that each subdomain D_i contains at most one error. If one random test case is chosen from each D_i, then the expected number of errors discovered is given by

$$E_p(k) = \sum_{i=1}^{k} \theta_i$$

where θ_i is the failure rate for D_i. Various θ_i distributions are discussed in the next section.

Let the expected number of errors found by n random tests be denoted by $E_r(k, n)$. If n random test cases are used, some actual set of values $\boldsymbol{n} = \{n_1, n_2, \cdots, n_k\}$ will occur, where n_i is the number of test cases from D_i. If $\boldsymbol{n}$ were known, then the expected number of errors found would be

$$E(\boldsymbol{n}, k, n) = \sum_{i=1}^{k} [1 - (1 - \theta_i)^{n_i}].$$

Since $\boldsymbol{n}$ is not known, our expected value must be computed by summing the product of the probability of occurrence of each $\boldsymbol{n}$ value times $E(\boldsymbol{n}, k, n)$ for that occurrence. Thus

$$E_r(k, n) = \sum_{\boldsymbol{n}} E(\boldsymbol{n}, k, n) * n! * \prod_{i=1}^{k} p_i^{n_i} \Big/ \prod_{i=1}^{k} n_i$$

which reduces to the expression

$$E_r(k, n) = k - \sum_{i=1}^{k} [1 - p_i * \theta_i]^n.$$

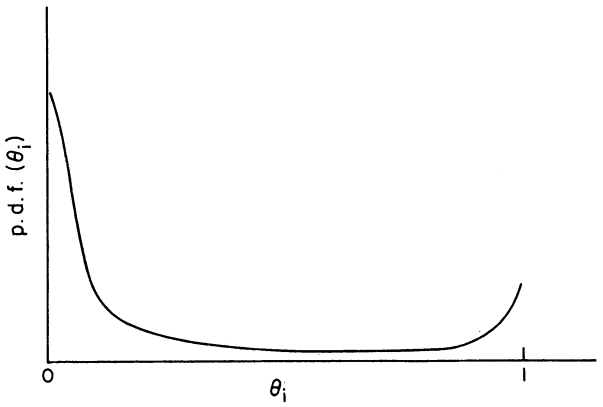


Fig. 1. Hypothetical probability distribution for θ_i values.

In the following section, various program conditions are simulated and the values of P_r, P_p, E_p, and E_r are calculated and compared.

III. Partition Testing Simulations

The point of doing the work of partition testing is to find errors. A partition testing instance such as path testing will be good to the extent that a test case chosen from a subset will have a high probability of finding an error (i.e., exhibiting an execution failure) if one is present affecting that subset. Ideally, then, θ_i should be either 0 or 1 (without k having to grow too large). Many feel that path testing approaches at least a little way toward this ideal. This is equivalent to believing that the distribution of values for program paths looks something like that shown in Fig. 1. On the other hand, Howden's experiments [10] discovered many examples of program paths that compute correct values for some, but not all, of their input data. This may mean that the distribution of θ_i values for path testing is more nearly uniform than suggested by Fig. 1. Therefore, we carried out simulations using distributions approximating that of Fig. 1 and also using uniform distributions.

In order to investigate the relative values of P_r and P_p, a 25 subset partition testing scheme was simulated, with one randomly chosen test case per subset, i.e., $k = n = 25$ and $n_i = 1$. For each trial, a new set of p_i and θ_i values was picked. The θ_i's were chosen from a distribution such that 2 percent of the time $\theta_i \geq 0.98$, and 98 percent of the time, $\theta_i < 0.049$. The p_i values were chosen from a uniform distribution. Then, the running of n random test cases and $k = n$ partition test cases was simulated. We found $P_r \geq P_p$ in 14 trials out of a total of 50. The mean value of (P_r/P_p) was 0.932. A histogram of the (P_r/P_p) values found is shown as Fig. 2. Fig. 3 gives a histogram of a similar experiment, run using the same distributions, but with $k = n = 50$. The expected values E_r and E_p were also calculated under the above assumptions. Fig. 4 gives a histogram of the E_r/E_p values for a 25 subset partition, while Fig. 5 is for $k = n = 50$. The results suggest that for those programs for which carrying out some contemplated partition testing scheme is much more expensive than performing an equivalent number of random tests, random testing will find more errors per unit cost.

To investigate the effect of the probability distribution of the θ_i's on the ratio P_r/P_p we also tried a number of uniform

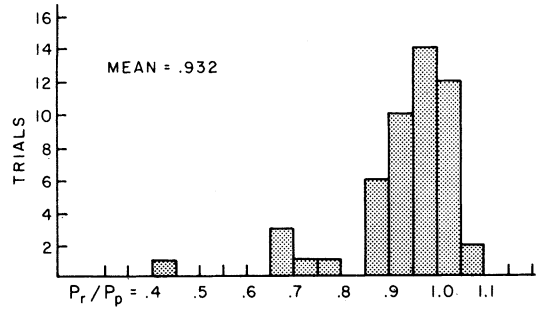


Fig. 2. Histogram of P_r/P_p with 25 partitions and 50 trials.

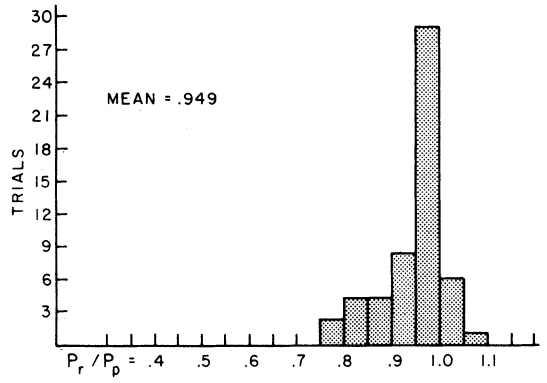


Fig. 3. Histogram of P_r/P_p with 50 partitions and 50 trials.

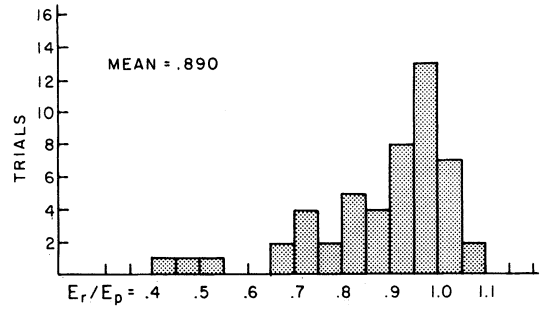


Fig. 4. Histogram of E_r/E_p for 25 partitions and 50 trials.

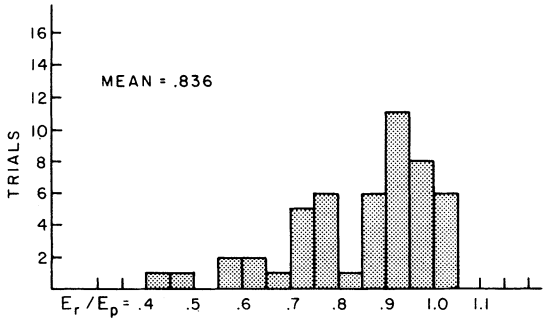


Fig. 5. Histogram of E_r/E_p for 50 partitions and 50 trials.

distributions where the θ_i's are allowed to vary uniformly from 0 to a value $\theta_{max} \leq 1$. θ_{max} was varied from 0.01 to 1. The results are summarized in Table I which shows the number of trials for which $P_r > P_p$, $P_r = P_p$ (i.e., $|P_r - P_p| < 0.000001$) and $P_r < P_p$ for different values of θ_{max} and two different partition sizes (50 and 25). Again one test case was selected from each subdomain in the partition. As θ_{max} increases, both P_r and P_p tend to 1 resulting in $P_r = P_p = 1$. If the failure rates are such that P_r, $P_p < 1$, we note that random

TABLE I
SIMULATION COMPARISONS OF P_r VERSUS P_p WITH THE FAILURE RATES DISTRIBUTED UNIFORMLY IN THE RANGE $0 - \theta_{max}$

		# of trials with									
k = 50	$P_r > P_p$	22	18	14	9	1	-	-	-	-	-
	$P_r = P_p$	1	-	-	4	31	50	50	50	50	50
	$P_r < P_p$	27	32	36	37	18	-	-	-	-	-
k = 25	$P_r > P_p$	17	15	11	11	8	7	2	-	-	-
	$P_r = P_p$	-	-	-	-	-	-	6	14	23	25
	$P_r < P_p$	8	10	14	14	17	18	17	11	2	-
	θ_{max} =	.01	.1	.2	.3	.4	.5	.6	.7	.8	1.0

TABLE II
COMPARISONS OF E_r VERSUS E_p WITH θ_i VARYING UNIFORMLY FROM 0 TO θ_{max}

k	# of cases for partition testing	# of cases for random testing	θ max	# of trials with $E_r < E_p$	# of trials with $E_r > E_p$	# of trials with $E_r = E_p = 0$
25	25	25	0.1	38	12	0
25	25	25	0.4	50	0	0
25	25	25	1.0	50	0	0
50	50	50	0.1	39	11	0
50	50	50	0.4	50	0	0
50	50	50	1.0	50	0	0
50	50	100	0.1	0	50	0
50	50	100	1.0	21	29	0

testing performed better for the lower failure rates and also when the size of the partition is 25 instead of 50.

Similar studies were carried out for the E_r and E_p measures. The results are summarized in Table II. Notice that in those instances where 100 simulated random test cases for a program were compared to 50 simulated partition test cases, random testing was superior. Since one would often expect to be able to perform 100 random tests less expensively than the 50 partition tests, these results reinforce the conjecture that random testing will often be more cost effective.

Table III summarizes a simulation with different assumptions about the θ_i values. It is plausible to suppose that for well designed programs, θ_i will be zero for the majority of subdomains of a partition. This idea is modeled by giving each θ_i a 90 percent chance of being zero. The results are similar to those of Table II.

IV. RANDOM TESTING EXPERIMENTS

Although the simulation results show that random testing can be cost effective under quite plausible conditions, the final and most convincing measures of effectiveness of any test method must be in terms of the results of actually testing

TABLE III
COMPARISONS OF E_r VERSUS E_p, WHERE PROB $(\theta_i = 0) = 0.9$

k	# of cases for partition testing	# of cases for random testing	θ max	# of trials with $E_r < E_p$	# of trials with $E_r > E_p$	# of trials with $E_r = E_p = 0$
25	25	25	0.1	28	18	4
25	25	25	0.4	32	14	4
25	25	25	1.0	43	3	4
50	50	50	0.1	29	21	0
50	50	50	0.4	35	15	0
50	50	50	1.0	47	3	0
50	50	100	0.1	4	46	0
50	50	100	1.0	23	27	0

programs. The results of several experiments with the random testing of programs are presented below.

The first three programs in the "Common Blunders" chapter of Kernighan and Plauger [11] which contain other than initialization errors are a sine program, a sorting program and a binary search program. (The initialization errors were not considered because they are "too easy" to detect.)

SIN Program: There are three noninitialization errors which can cause the program to fail to execute according to specification. Fifty random test cases were run. The test cases detected one of the errors 11 out of 50 times, another was detected 24 out of 50 times, and a third, 45 out of 50 times.

SORT Program: Twenty-four test cases of randomly generated lists of size ranging from 1 to 50 were executed. The "off-by-one" error of the program was detected by 21 of the 24 test cases.

BINARY SEARCH: Fifty text cases were generated, with table size ranging from 0 to 20. Here, an "operational profile" of the program was considered. In 20 percent of the test cases the input element searched for was not in the table, but was in the table for the remaining 80 percent of the cases. The program's error was detected by 18 of the 50 test cases.

This last case brings up an important question. Should random testing be done by a uniform sampling of the input domain, or from an operational profile, or both? For reliability estimates, the operational profile should be used. We speculate that this may not generate data as effective, overall, for error detection as would uniform sampling, but it might be more effective for the more critical errors.

Next, we tested the line editor program discussed by Goodenough and Gerhart [6], using random character strings weighted for more frequent selection of blanks and new line characters. The proportions of test cases which detected the five errors given in [6, Table III] are shown in Table IV. Only one error at a time was present in the program during the experiment.

BUGGYFIND [1], [2] contains a subtle error, actually made during a conversion of Hoare's FIND program. Boyer *et al.* [1] state "This bug is quite difficult to detect by conventionally testing the program with random array inputs. . .." Our randonly generated test data detected the error in 3 out of 50 cases. This may at first seem like poor performance but actually it is quite good. The test case generation and execution was easy to perform and did demonstrate a high error detection probability for the test sequence as a whole. (Consider that if the error has an associated failure rate of 0.06, the probability of detection in 50 trials is approximately 0.95). It is interesting to notice that branch testing is not adequate for detecting this error [1]. However, our 3 random test cases which detected the error did each cover all but one of the program's branches. Three other cases each covered all but one branch, without detecting the error. The total set of test data did ensure the execution of all branches.

TABLE IV
ERRORS DETECTED IN LINE EDITOR PROGRAM

Error No.	Detection Proportion
1	5/10
2	8/10
3	9/10
4	10/10
5	9/10

DeMillo *et al.* [2] also experimented with random testing of FIND. Our test cases were chosen from vectors of size ranging from 1 to 10, with element values chosen uniformly from the range -100 to 100. Using a similar test generation method, DeMillo *et al.* created a 1000 member set of random test cases which was equivalent in power, according to their mutant catching measure, to their carefully constructed seven member test case set referred to as D_1 in [2]. From the description in [2], creating D_1 appears to have required considerable effort. Additionally, they created a set of 100 random test cases. By their measure, it was only slightly inferior to the larger random set and to D_1, leaving 11 live mutants rather than 10. Our results indicate that such a 100 vector test set would be nearly certain to catch the (presumably subtle) actual error in BUGGYFIND. Considering that the random test cases were inexpensive to generate and execute, mutation analysis can be viewed as confirming the cost effectiveness of the error finding ability of random testing in this particular experiment.

The standard deviation program studied by Hetzel [8] and Howden [9] also contains an actual error. All 25 randomly generated test cases detected the error, reminding us that many "real-life" errors are in fact easy to detect if one is careful to determine whether a test run failed to execute properly. No test data are good enough if the tester is unable or unwilling to determine whether the output is correct. This suggests that automatic or machine assisted output checking is highly desirable. It is essential if large numbers of random tests are to be performed.

Two programs (algorithms 424 [4] and 408 [7]) from the Collected Algorithms of the ACM were tested in order to obtain further experimental evidence about "real-life" errors. Considering the source of these programs, their errors might be considered to be quite subtle. The programs were picked without bias from those for which errata had been published, except that large programs were not considered due to clerical considerations. Further, only "true" errors were considered, in that errata related to efficiency and portability matters were not considered.

Algorithm 424 (Clenshaw-Curtis Quadrature): No actual experiments were needed. Virtually any test case will discover the error, so long as the tester carefully checks the output. The problem is that the error causes an incorrect secondary output value, which was apparently not checked during original certification. The definite integral value computed as the primary output is apparently correct.

Algorithm 408 (Sparse Matrix Package): Subroutine TRSPMX of the algorithm package is reported to have two errors. The first causes a failure whenever the input is a matrix with only one column. The second causes failure when all elements in the first row of the input matrix are zero. Since sparse matrix packages are presumably useful only when matrix dimensions are not very small, we set up the random test case generation to create no tests with row or column dimensions smaller than 10. (The maximum was set at 50.) When the density of test matrices was allowed to vary uniformly between 0 and 0.4, the second error was detected by 14 cases out of a total of 135. With the density varying between 0 and 0.2, the error was detected by 6 cases out of 38. However, the test data will obviously not detect the first error. It could, of course, be argued that the first error should have little effect on the operational reliability. If no lower limits are placed on the dimensions of the matrix, there is a small but nonzero probability that a randomly selected test case will detect the first error.

V. Coverage Measures for Random Testing

In addition to performing simulations and error finding experiments, random testing can be evaluated using various test coverage measures. That is, one can determine the extent to which random testing achieves the goals implicit in other testing strategies. In particular we look at segment testing, branch testing, a variation of structured path testing, mutation analysis [2], required pair testing [12], and the linear code sequence measures suggested by Woodward *et al.* in [15].

A program segment is a sequence of consecutive statements that are always executed together. Segment testing requires that we develop test cases which will cause each program segment to be executed at least once. A more extensive strategy is branch testing where we ask that each possible transfer of control in the program be exercised by at least one test case. Clearly branch testing subsumes segment testing. Since many errors can be detected only if the segments and branches of a program are executed in a certain order, testing each execution path in the program might increase the probability of error detection. This strategy is known as path testing and is generally considered impractical because programs with loops can have an infinite number of paths. Structured path testing [9] refers to a class of path testing strategies where the number of paths is kept managable by considering only up to k iterations of each loop where k is a small constant. For our purposes we shall let $k = 2$. Besides structured path testing we will consider two other strategies that attempt to bridge the gap between branch and path testing. In [15], Woodward *et al.* proposed a set of Test Effectiveness Measures based on the notion of Linear Code Sequence and Jump (LCSAJ) where an LCSAJ is a sequence of consecutive statements in the program text starting at the entry point or after a jump and terminating at another jump or the end of the program. The general measure is defined as follows:

$$\mathrm{TER}_{n+2} = \frac{\begin{array}{l}\text{number of distinct subpaths of length } n \text{ LCSAJ's} \\ \text{+ number of complete paths of length } < n \\ \text{LCSAJ's exercised at least once}\end{array}}{\begin{array}{l}\text{total number of distinct subpaths of length } n \\ \text{LCSAJ's + total number of complete paths of} \\ \text{length } < n \text{ LCSAJ's.}\end{array}}$$

The measures TER_1, TER_2 are defined so that $\mathrm{TER}_1 = 1$ if segment testing is achieved and $\mathrm{TER}_2 = 1$ if branch testing is completed. We will use TER_3 and TER_4 to measure the degree of coverage achieved by random testing. Another strategy that includes branch testing but stays short of path testing is the required pairs strategy [12] where certain combinations of segments (required pairs) are tested. The required pairs are obtained using data flow analysis and they are pairs of segments $[i, j]$ such that a definition to a program variable in segment i reaches a reference to that variable in segment j (a definition in i reaches a reference in j if there is a path from i to j along which the variable is not undefined or redefined). Finally, we consider mutation analysis, which has been proposed as a measure of test set adequacy. In mutation analysis [2] a number of simple changes (mutations) are introduced into a program in a systematic manner. The adequacy of a test set is measured by its ability to distinguish the mutant programs from the original program.

Segment, branch, structured path, and required pair testing can be used to evaluate the effectiveness of random testing by determining the extent to which random testing satisfies each strategy, i.e., what proportion of segments, branches, structured paths, and required pairs are executed by a randomly generated set of test cases. The other measures can be used directly. One problem in carrying out such an evaluation is to decide how many random test cases should be used and how should those cases be generated. Our approach to the first issue was to use some moderate number of test cases (20-120), on the premise that random cases are inexpensive to generate, and to repeat each experiment with a different number of cases. The second issue is more involved because the range from which the random inputs are generated can affect their performance and this effect is generally program-dependent. The simple solution of using a uniform distribution on the ranges specified in the input specification is useful for many programs, but for others is not very practical, as it may produce test cases that are very expensive to run. For example, selecting large numbers as the dimensions of some input array will usually cause both test case generation time and program execution time to

TABLE V
COVERAGE MEASURES FOR RANDOM TESTING

program	# of random test cases	proportion of segments covered	proportion of branches covered	proportion of paths covered	proportion of req. pairs covered	TER_3	TER_4	ranges used for input variables
SIN	20	5/5	5/6	1/4	12/19	4/6	8/14	$0.0 \leq \|X\| \leq 10.0$
'	80	5/5	5/6	1/4	12/19	4/6	8/14	$0.0 \leq E \leq 0.00005$
'	80	5/5	5/6	3/4	12/19	5/6	11/14	$0.0 \leq \|X\| \leq 20.0$ $0.0 \leq E \leq 0.3$
BINOM	20	10/10	13/13	5/10	25/27	9/11	19/24	$10 \leq M \leq 15$
'	40	10/10	13/13	6/10	25/27	9/11	19/24	$0 \leq N \leq 8$
'	40	10/10	13/13	10/10	25/27	11/11	24/24	$5 \leq M \leq 15$ $0 \leq N \leq 10$
TRITYP	25	22/30	26/39	4/9	39/86	15/25	32/58	$10 \leq I,J,K \leq 25$
'	100	26/30	28/39	5/9	44/86	18/25	38/58	
'	25	30/30	39/39	8/9	64/86	24/25	56/58	$1 \leq I,J,K \leq 5$
SEARCH	30	11/11	13/13	11/38	23/50	10/11	21/29	$1 \leq N \leq 10$
'	100	11/11	13/13	19/38	29/50	10/11	26/29	$1 \leq F,A(I) \leq 50$
'	120	11/11	13/13	29/38	31/50	10/11	25/29	$1 \leq N \leq 16$ $1 \leq F,A(I) \leq 48$
SORT	40	12/12	15/15	29/75	33/35	13/16	33/50	$1 \leq N \leq 6$
'	80	12/12	15/15	58/75	33/35	13/16	33/50	$1 \leq A(I) \leq 100$

increase. Thus, we used ranges that seemed reasonable and tried different ranges for each experiment to determine the sensitivity of the effectiveness of random testing on the range from which random test inputs are chosen. The following set of five programs was used in the experiment.

1) *SIN [11]:* Maclaurin series sine function (correct version). It computes the sine of X to accuracy E.

2) *BINOM (Algorithm 515–Collected Algorithms of ACM):* A routine to compute the binomial coefficient $\binom{M}{N}$.

3) *TRITYP:* Classifies triangles with sides of length I, J, K.

4) *SORT [11] (Correct Version):* A routine that sorts an array A of size N.

5) *SEARCH [11]:* A binary search routine (correct version). It searches for F in an array A with N elements.

Table V contains representative results of the evaluations. To summarize, the moderate number of random test cases used in the experiments on the average achieved 97 percent of segment testing, 93 percent of branch testing, 57 percent of structured path testing, 72 percent of required pairs testing, 81 percent of TER_3, and 74 percent of TER_4. In most programs the results were not very sensitive to the ranges chosen for the input variables but this effect can be seen in the TRITYP program.

Also, a limited evaluation of random testing using mutation analysis is reported in [12]. Seven programs were tested, using the mutation system at Georgia Tech, with from 8 to 20 random test cases. 79 percent of the mutants were eliminated by the test cases as compared with 84 percent for branch testing and 90 percent for required pairs testing (the number of remaining mutants includes those that are equivalent to the original program). Random testing was least effective in two triangle classification programs where equal values for two or three of the sides of the triangle are important but difficult to generate randomly. In the remaining five programs, random testing performed better than branch testing in four and better than required pairs testing in one program.

VI. CONCLUSIONS

The results compiled so far indicate that random testing can be cost effective for many programs. Also, random testing allows one to obtain sound reliability estimates. Our experiments have shown that random testing can discover some relatively subtle errors without a great deal of effort. We can also report that for the programs so far considered, the sets of random test cases which have been generated provide very high segment and branch coverage. The unexecuted branches tend to be those which provide handling of exceptional cases. This suggests that random testing should be augmented with extremal/special values testing [9].

Random testing definitely has a place in the software engineer's repertoire of testing techniques. Although the most cost effective mix of testing strategies is difficult to determine and will vary from case to case, a powerful mix would be to begin with random testing, followed by extremal/special values testing, while keeping track of the branch coverage achieved. The testing mix would conclude with tests to cover any remaining untested branches.

ACKNOWLEDGMENT

We would like to thank S. Chen for helping with the experiments described in Section IV, J. Wiorkowski for discussions and suggestions on the statistical models, and the referees for their suggestions to improve the clarity of this paper.

REFERENCES

[1] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT–A formal system for testing and debugging programs by symbolic execution," in *Proc. 1975 Int. Conf. Reliable Software*, pp. 234–245.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[3] J. W. Duran and J. J. Wiorkowski, "Quantifying software validity by sampling," *IEEE Trans. Reliability*, vol. R-29, pp. 141–144, 1980.

[4] K. O. Geddes, "Remark on algorithm 424," *ACM Trans. Math. Software*, vol. 5, p. 240, 1979.

[5] E. Girard and J.-C. Rault, "A programming technique for software reliability," in *Conf. Rec., 1973 IEEE Symp. Comput. Software Reliability*, pp. 44–50.

[6] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 156–173, 1975.

[7] F. Gustavson, "Remark on algorithm 408," *ACM Trans. Math. Software*, vol. 4, p. 295, 1978.

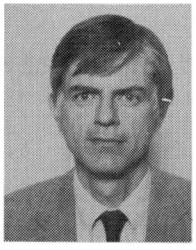
[8] W. C. Hetzel, "An experimental analysis of program verification methods," Ph.D. dissertation, Univ. North Carolina, 1976.

[9] W. E. Howden, "Symbolic testing–Design techniques, costs, and effectiveness," Nat. Bureau of Standards Tech. Rep. PB-268517, May 1977.

[10] ——, "Functional program testing," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 162–169, 1980.

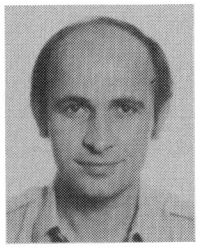
[11] B. S. Kernighan and P. J. Plauger, *The Elements of Programming Style*, 2nd ed. New York: McGraw-Hill, 1978.

[12] S. C. Ntafos, "On testing with required elements," in *Proc. COMPSAC 81*, Nov. 1981, pp. 132–139.
[13] G. J. Myers, *The Art of Software Testing*. New York: Wiley, 1979, p. 36.
[14] R. A. Thayer, M. Lipow, and E. C. Nelson, *Software Reliability*. Amsterdam, The Netherlands: North-Holland, 1978.
[15] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with path analysis and testing of programs," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 278–286, May 1980.



Joe W. Duran (S'74–M'76) received the B.S. degree in physics from Southern Methodist University and the Ph.D. degree in computer science from the University of Texas, Austin.

He has worked for Texas Instruments, Inc., Tracor, Inc., and Southwest Research Institute, and was on the faculty of the University of Texas, Dallas. He is currently an Associate Professor in Computer Science at the University of Texas, San Antonio. His research interests include software reliability and testing, programming, methodology, and man–machine interaction.



Simeon C. Ntafos (S'76–M'78) was born in Trikala, Greece, in 1952. He received the B.S. degree in electrical engineering from Wilkes College, Wilkes-Barre, PA, in 1974, and the M.S. degree in electrical engineering and the Ph.D. degree in computer science from Northwestern University, Evanston, IL, in 1976 and 1979, respectively.

He was a Visiting Assistant Professor in the Department of Electrical Engineering and Computer Science at Northwestern University from September 1978 to August 1979. Since September 1979, he has been an Assistant Professor in the Computer Science Program at the University of Texas at Dallas, Richardson. His current research interests are in the areas of software reliability and computational complexity.

On the Execution of Large Batch Programs in Unreliable Computing Systems

CLEMENT H. C. LEUNG AND QUI HOON CHOO

Abstract—**The execution of long-running batch programs imposes severe reliability constraints on a computing system since the occurrence of a failure during its execution is more likely and that once occurred, a failure would destroy all the processing performed thus far. This paper studies the execution delay and machine resources consumed in supporting the running of large batch programs in a computing environment interrupted by failures. The effect of checkpoints and their optimal insertion are also considered. The results are applicable to arbitrary law of failure.**

Index Terms—**Batch program, checkpoint, failure law.**

I. Introduction

THE execution of long-running batch programs often places severe reliability constraints on a computing system since any system failure during its execution will prevent successful completion and the processing carried out thus far will be wasted, and rerunning of the program perhaps many times is necessary. Of course, a reliable computing system is essential not only to long-running programs but also to shorter ones as well. However, while short- and medium-running programs can often be guaranteed a successful completion in no more than a few reruns, long-running ones may require an excessive number of reruns to achieve a successful completion due to the increased likelihood of a failure during its execution, and one occurring late in the run is especially costly in terms of wasted machine resources. Such repetitive reprocessing and the consequential delay may not only be unacceptable to the user as frequently there are deadlines for the results but also, since abortive runs represent wasted work, substantial erosion to the computer processing capacity will also ensue.

Long-running batch programs can arise in a wide variety of contexts and their demand on system resources is often considerable. They may, for example, be related to industrial optimization, numerical computation, or simulation of complex systems. For instance, one of the programs used by the Computational Physics Group of Reading University consists of simulating the behavior of 3000 particles to derive their thermodynamic properties. Usually 10 000 time steps are required to be simulated and each time step takes approximately 5 CPU seconds on the CDC 7600 computer so that the entire experiment often requires a total of 14 hours of CPU time.

Manuscript received October 1, 1982; revised May 27, 1983.

C.H.C. Leung is with the Department of Computer Science, University College, London University, London WC1E 6BT, England.

Q. H. Choo is with the Division of Mathematics and Computing, London Research Station, British Gas Corporation, London SW6 2AD, England.

0098-5589/84/0700-0444$01.00