

# RANDOM TESTING

Richard Hamlet

## 1. Introduction

In computer science, originally in its rarefied offshoots centered around artificial-intelligence laboratories at Stanford and MIT, the adjective "random" is slang with a number of derogatory connotations ranging from "assorted, various" to "not well organized" or "gratuitously wrong" (Steele, 1983). "Random testing" of a program in this sense describes testing badly or hastily done, the opposite of *systematic* testing such as functional testing or structural testing. This slang meaning, which might be rendered "haphazard testing" in normal parlance, is probably the common one, as in the sentence, "Random testing, of course, is the most used and least useful method." In contrast, the technical, mathematical meaning of "random testing" refers to an explicit lack of "system" in the choice of test data, so that there is no correlation among different tests. (There is dispute over the usefulness of random testing in this technical sense; however, it is certainly not the most used method.) If the technical meaning contrasts "random" with "systematic," it is in the sense that fluctuations in physical measurements are random (unpredictable or chaotic) vs. systematic (causal or lawful).

Why is it desirable to be "unsystematic" on purpose in selecting test data for a program?

- (1) Because there are efficient methods of selecting random points algorithmically, by computing pseudorandom numbers; thus a vast number of tests can be easily defined.
- (2) Because statistical independence among test points allows statistical prediction of significance in the observed results.

In the sequel it will be seen that (1) may be compromised because the required result of an easily generated test is not so easy to generate. (2) is the more important quality of random testing, both in practice and for the theory of software testing. To make an analogy with the case of physical

measurement, it is only random fluctuations that can be "averaged out" to yield an improved measurement over many trials; systematic fluctuations might in principle be eliminated, but if their cause (or even their existence) is unknown, they forever invalidate the measurement. The analogy is better than it seems: in program testing, with systematic methods we know what we are doing, but not what it means; only by giving up all systematization can the significance of testing be known.

Random testing at its best can be illustrated by a simple example. Suppose that a subroutine is written to compute the (floating-point) cube root of an integer parameter. The method to be used has been shown to be accurate to within  $2 \times 10^{-5}$  for input values in the interval  $X = [1, 10^7]$ . Assuming that all values in this range are equally likely (that is, the *operational profile* that describes how the subroutine will be used, has a uniform distribution), a random test of 3,000 points can be performed as follows:

Generate 3,000 uniformly distributed pseudorandom integers in the interval  $X$ . Execute the subroutine on each of these, obtaining output  $z_t$  for input  $t$ . For each  $t$ , compute  $z_t^3$ , and compare with  $t$ .

The process can be automated by writing a driver program incorporating the pseudorandom-number generator and the output comparison. If any of the 3,000 outputs fails to be within  $2 \times 10^{-5}$  of the desired result, the subroutine must be corrected, and the test repeated, but without reinitializing the input generator. That is, the pseudorandom sequence is continued, not restarted. Suppose that eventually the subroutine passes the test by computing 3,000 correct values. As will be shown in the sequel, the probability that the subroutine could fail one or more times in 1000 subsequent runs is less than about 0.05. Roughly, it is 95% certain that the subroutine is going to be correct 999 times out of 1000. The ability to quantify the significance of a test that does *not* fail is unique to random testing.

Along the way to a successful random test, there may be many cycles of finding a program failure, and sending the software back for repair. It is to be expected that random testing is less good at exposing software defects than systematic methods designed for just that purpose. However, as discussed in section 3, the deficiency is not so severe as might be imagined--a great deal of effort may be invested in systematic testing without realizing much advantage.

The example above is unrealistic in two important ways, ways that show the weakness of random testing in practice. First, it is seldom so easy to calculate random test inputs. The input values may be more complicated than integers confined to a given interval. Worse, the operational profile may be unknown, so that predictive random inputs cannot be found at all. The second (and more damning) unrealistic feature of the example is that it is seldom possible to check the program's results against what it is supposed to do. In the usual case the comparison with correct behavior is hard because the requirements are imprecisely stated; in the worst case the required behavior is so complex that checking it at all presents formidable problems.

## **2. The Random-testing Method**

The example in the Introduction indicates how random testing is done: the input domain is identified, test points are selected independently from this domain, the program under test is executed on these inputs (they constitute a random test set), and the results compared to the program specification. The test is a failure if any input leads to incorrect results; otherwise it is a success. To make this description precise requires a discussion of program and test types, of the input-selection criterion, and of the output comparison.

### **2.1 Program and Test Variations**

The cube-root subroutine of the Introduction might be used directly by a person to calculate and print roots, with a driver program that is very similar to the test harness already described. The predictions of its quality then refer to human usage in which it is no more likely that one cube root

will be requested than another. The random test is a system test in this case, and its intent would be twofold: to uncover as many failures as possible, and finally when no more are found, to predict the future behavior of the software.

On the other hand, the cube-root routine might be part of a library, and testing it in isolation is an example of unit testing. For unit testing, the predictive feature of random testing is lost, because a subroutine may have entirely different usage when incorporated in a variety of software systems. If in system P the cube roots requested are concentrated in the input interval  $[1, 100]$ , while for system Q they range over the whole interval  $[1, 10^7]$ , then the test of the Introduction better applies to Q than to P. Since the system usage of a general routine is arbitrary ("random" in the slang sense!), statistical prediction is not possible. Furthermore, in some system R, most system functions might not require cube roots at all, so that even if an accurate prediction were available for the root routine, it would have little to do with a corresponding prediction for R itself. Random system testing of P, Q, and R themselves would be needed to make predictions about their behaviors. However, random testing is still used at the unit level, because it is surprisingly competitive with systematic methods in failure-finding. The uniform input distribution is used in the absence of any information about the location of failure points in the input domain. (This topic is discussed in detail below, section 3.)

At the system-test level, the simplicity of random testing is also complicated by the several ways in which programs may be used, which influences the way they are tested.

"Batch" programs are purely functional in nature: a batch program treats one input at a time, and its output depends only on that input. The sequencing of input is irrelevant. Each test of a batch program starts from scratch, so random testing such a program is straightforward--the tests have nothing to do with each other.

"Interactive" programs are in operation more or less continuously, but repeat an interactive cycle of reading input, computing, and writing output. At the head of the cycle, there may be a long wait for input, as a human user thinks, goes to lunch, etc. However, each interaction is not like a batch run, because an interactive program typically has a "memory." Results from

one interaction may influence subsequent interactions. It is typical for an interactive program to have a number of "modes" into which it is thrown by certain inputs, and to behave entirely differently in different modes. Random testing of interactive programs must take account of input sequences.

Some programs, like operating systems, telephone switches, or other realtime control programs, operate continuously, receiving input and producing output in bits and pieces over time. When such a program is slowed down and given a simple input pattern, it may appear to act like an interactive program, but in normal operation the situation is much more chaotic. Random testing of a realtime program requires streams of input, where not only the input values and their sequence, but also their spacing and overlap in time, are significant.

Most of the theory of random testing applies easily only to batch programs.

## **2.2 The Operational Profile**

Every testing method (save exhaustive testing for batch programs) is less than perfect. It is altogether possible that success will be observed on some test data, yet in fact the software contains defects not brought to light. It has therefore been suggested (Musa, 1988) that the choice of test points take into account usage patterns the software will encounter in its intended environment. The argument is that by testing according to usage, the failures found by imperfect methods are more likely to be the important ones, i.e., the ones most users would first encounter. In statistical prediction, the argument for tests mimicing usage is even more compelling: unless random tests are drawn as the software will be actually used, the tests are not a representative sample and all statistical bets are off.

Strictly speaking, the operational environment for a program is one part of the software requirements. An *operational profile* can be obtained in practice by partitioning the input space, and assigning to each subdivision  $D$  of the partition a probability that in normal usage, the input will come from  $D$ . The name comes from representing the frequencies as a histogram whose bars

form a "profile" of typical usage frequency vs. subdivision. Any partitioning scheme may be used, but the most common is a functional partition that groups inputs for which the software is required to perform the same function. The input space may also be divided into equal-sized chunks without regard for function, or features of the program structure may be used to define input classes. However, the requirements are usually not helpful in assigning frequencies except for the functional partition. Fig. 1 shows an illustrative operational profile for a program with one real input, in which five functions  $F_1, \dots, F_5$  are required, with frequencies 0.2, 0.4, 0.3, 0.05, 0.05 respectively.

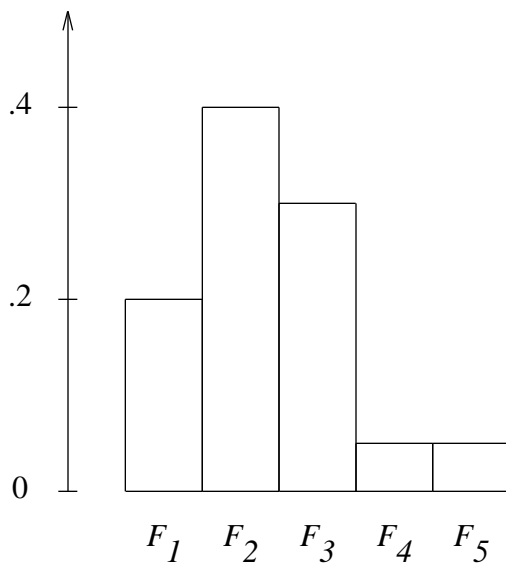


Figure 1. An operational profile.

In the limit as the partition is refined to smaller subdivisions, the operational profile becomes a probability density function giving the probability that each input point will arise in use. In practice, the software requirements provide no way to refine the partition except to assume that the probability of choosing a point is constant within each subdomain. By arbitrarily subdividing subdomains according to some scale, the probability can be apportioned accordingly. For example, if in Fig. 1 the input space is the integer interval  $[0, 100)$ , and the subdomains occupy respectively intervals of length 20, 10, 40, 10, and 20 in order, then the density function has the appearance of

Fig. 2. (The density is discrete, but the dots of Fig. 2 are not carefully positioned at the integers.)  
 The probability amplitudes are 1/100, 1/25, 3/400, 1/200, 1/400 respectively. (For example, 0.3 of the points are to be chosen in the interval [30,69) for  $F_3$ , where there are 40 possibilities, making the probability of choosing each point 0.3/40.)

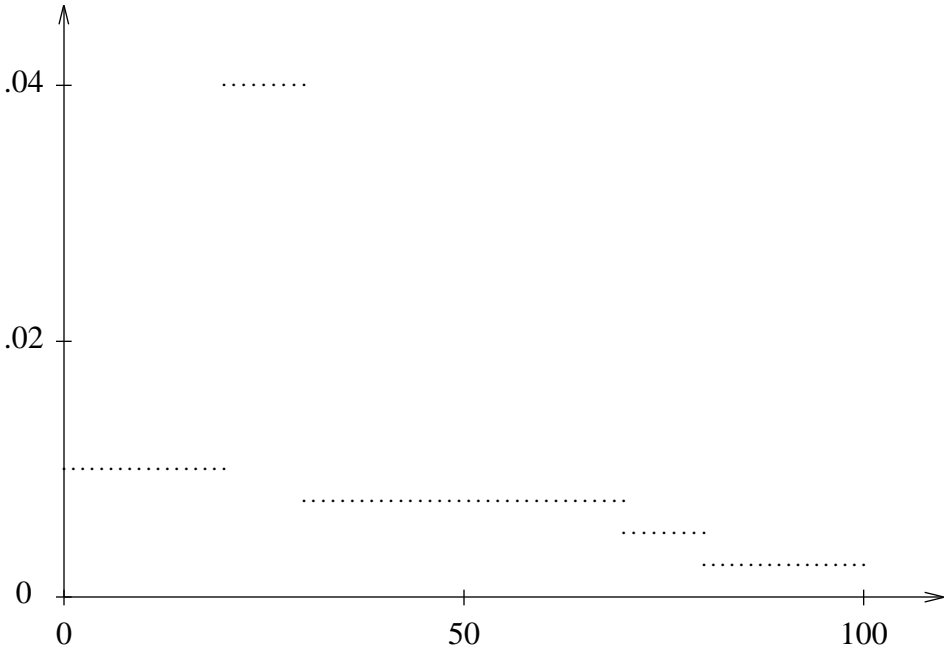


Figure 2. A probability density.

Rather than the density function  $f$ , it is usual to work with the distribution function

$$F(t) = \int_{-\infty}^t f(x)dx$$

(for the case of a continuous density  $f$ ; for a discrete density a sum replaces the integral). Fig. 3 shows the operational distribution corresponding to Fig. 2.

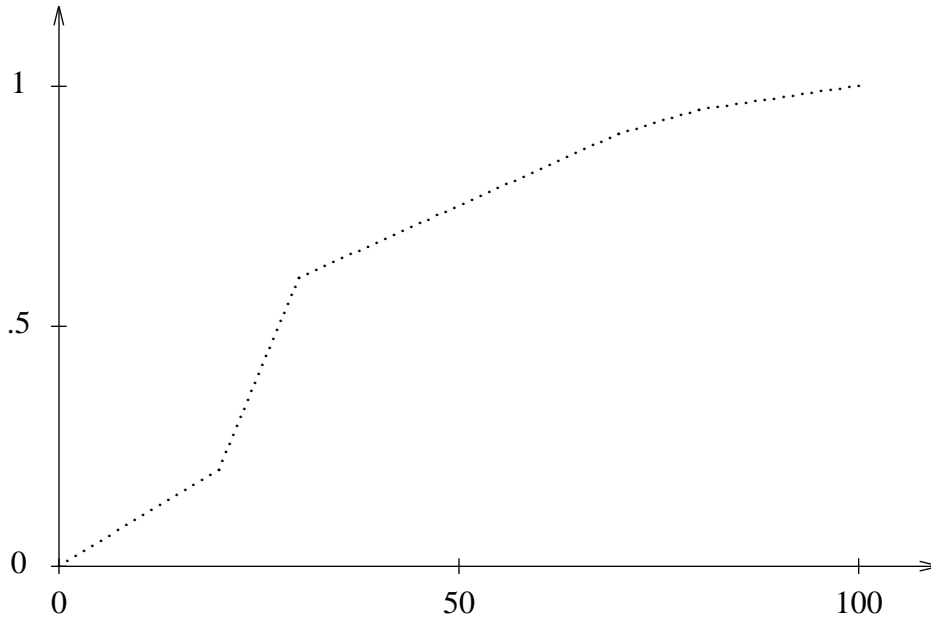


Figure 3. An operational distribution.

To select test inputs according to an operational profile for a batch program, choose uniformly-distributed pseudorandom numbers in the intervals corresponding to the requirements partition, each interval considered its proper fraction of the time. In the example above, to properly sample  $F_3$ , 0.3 of all test points should be uniformly distributed in  $[30,69)$ . Or, if an operational distribution  $F$  is given, start with uniformly distributed pseudorandom real numbers  $r \in [0,1]$ , for which the corresponding  $F^{-1}(r)$  are distributed according to  $F$ .

For interactive or realtime programs, the choices are more complicated, because input sequences must be chosen. Often, the requirements are silent about the probability of one input following another, so the only possibility is to order choices uniformly. Or, the input space can be expanded to include variable(s) representing stored information, whose values are taken to be uniformly distributed. Neither choice is very satisfactory if it cannot be supported by the requirements document.



## 2.3 The Specification Oracle

Any testing method must rely on an oracle to judge success or failure of each test point. To be useful in testing, the oracle must be *effective*--that is, there must be a mechanical means of judging the correctness of any program result. Testing theory usually assumes that an oracle is given, but in practice this false assumption has more influence than is usually admitted. It is responsible for the bias that the better testing method requires fewer test points. (In that case the oracle--which is likely to be hand calculation--has less to do.) The assumption is also responsible for testing done with trivial values, which may distort the ability of systematic methods to uncover failures. (With trivial values, hand calculation is easier.) Random testing cannot be attempted without an effective oracle, however. A vast number of test points are required, and they cannot be trivialized to make things easier for a human oracle.

## 2.4 Description of the Method

For batch programs whose operational profile is known, for whose domain a pseudorandom number generator is available, and for which there is an effective oracle, random testing is straightforward. A test set size  $N$  is selected.  $N$  may be based on a desired reliability (as indicated in the Introduction, and discussed in detail in the following section); however, in practice it is more likely to be chosen to reflect available resources for testing. Using the operational profile with  $K$  subdomains  $D_1, D_2, \dots, D_K$ , these points are apportioned according to the corresponding probabilities  $P_1, P_2, \dots, P_K$  that input will be in each subdomain. The pseudorandom number generator is used to generate  $N_i$  test points for subdomain  $D_i$ ,  $N_i = p_i N$ , for  $1 \leq i \leq K$ . These  $N$  points together form the test set. The program is executed, and the oracle used to detect any failures. If failures are found, the software is corrected and the test repeated with a new pseudorandom test set. Random testing ends when no failures are detected for the size- $N$  test set.

In unusual cases, the requirements document will describe expected usage for an interactive or realtime program so that a partition is defined for sequences of inputs. That is, the operational profile will describe classes of input sequences, and the above procedure can be used to select a random test set (of sequences). Or, the inputs may be available from a physical system like a radar, or from a simulation of such a system, so that random test sets are available by just recording them as they occur. (The question of selecting a time interval to record is often a difficult one, particularly when the resulting test set varies widely with time.) But the usual situation, particularly for realtime programs, is a requirements document that gives little information about input sequences and overlaps, because the usage information is unknown. (A few constraints may be available, for example that certain events will not occur too frequently.) Thus in the usual case, the operational profile is not available, since it is sequences of values that constitute actual input, not single values.

When the operational profile is not available, random testing can still be used with a uniform distribution. But the predictive power of a successful test is compromised, because if actual usage concentrates on some inputs, the predictions for such inputs will be overly optimistic, and observed behavior may be arbitrarily worse than expected. For batch programs, all inputs are taken to be equally likely; this is equivalent to taking the whole input space as a single subdomain that occurs with probability 1. For an interactive program, sequences of inputs can be constructed by selecting each member of a sequence according to an operational profile for single inputs. It may happen that imprecise requirements nevertheless reject certain sequences constructed without regard for the probability that one input will follow another. For example, in a transaction-processing system, it may be known that certain combinations of transactions seldom or never occur. For a realtime system, test sets are multiple concurrent sequences of inputs. The relative time position of inputs in each sequence can be chosen using a standard distribution (e.g., the Poisson distribution), but without some help from the requirements about distribution parameters, such sequences are very unlikely to have any connection to reality, even if individual-value probabilities and sequence probabilities are available. (In the realtime case, it may happen that overlap requirements are better understood than sequencing requirements.)

It should be evident that the application of random testing to interactive and realtime systems requires far more knowledge about these systems' usage than is usually available. It can be argued that the absence of usage data is a critical deficiency in the requirements phase of development: insofar as the necessary operational-profile information is not available to construct accurate random test sets, the application has not been sufficiently analyzed to attempt a computer solution. To proceed without this information would be considered unprofessional negligence in most branches of engineering. Because "software can do anything," we are led to attempt general solutions to difficult problems, resulting in software that we cannot test as it will be used, which can fail because of design flaws excited by the (unspecified) circumstances of its special use. Other branches of engineering are more aware of their limitations.

## 2.5 Reliability and Random Testing

It is the primary strength of random testing that the significance of a successful random test can be statistically predicted. Subject only to the availability of an accurate operational profile, and to the validity of certain independence assumptions, a successful random test of program  $P$  can justify statements like, "It is 99% certain that  $P$  will fail no more than once in 10,000 runs" (for a batch program); "It is 99% certain that  $P$  will have a mean time to failure of greater than 10,000 hours of continuous operation" (for a realtime program). The validity question will be considered in the sequel; here we derive the conventional theory behind such statements.

Consider first the batch case in which previous inputs cannot influence the outcome of a test of program  $P$ . It is usual to postulate that  $P$  has a constant *failure rate* (or *hazard rate*)  $\theta$ . Then on a single test, the probability that  $P$  will fail is  $\theta$ , or  $1-\theta$  that  $P$  will succeed. On  $N$  independent tests drawn according to the operational profile, the probability of universal success is  $(1-\theta)^N$ , or probability  $e = 1-(1-\theta)^N$  that at least one failure be observed.  $1-e$  is the confidence probability that a failure will occur no more often than once in  $1/\theta$  runs. Solving for  $1/\theta$ , which is also the mean time to failure (MTTF),

$$\frac{1}{\theta} \geq \frac{1}{1-(1-e)^{1/N}}.$$

The number of tests required to attain confidence  $1-e$  in this MTTF is

$$\frac{\log(1-e)}{\log(1-\theta)}.$$

The example in the Introduction had  $N=3000$ , MTTF 1000 runs, so  $1-e \approx .9503$ .

The cases of interactive or realtime programs are the same, if the more complex operational profiles these programs require are available. In the case of realtime programs, if inputs arrive at an average rate of  $\lambda$ , then both MTTF and test-run count  $N$  can be converted into time units because  $\lambda t$  inputs arrive in time  $t$ . For example, 90% confidence in a MTTF of  $10^6$  hours requires testing for about  $2.3 \times 10^6$  hours.

It should be emphasized again that although these formulae can be used in the case that an unknown operational profile (or one that is inadequate for the interactive or realtime cases) is replaced by a uniform distribution, the results can be arbitrarily inaccurate; and, the error is always in the optimistic direction of predicting a greater confidence in a better MTTF than actually justified.

As the unique testing method for which test success can be quantified in meaningful software-quality terms (i.e., confidence in MTTF), random testing has an important theoretical and experimental role. It serves as the standard of comparison by which other methods should be evaluated. For all the difficulties the method presents, this touchstone role remains of great importance.

### **3. Efficacy of Random Testing for Finding Failures**

Most systematic testing methods arose in practice, from the idea of "coverage." In systematic testing, some aspect of a program is identified as a potential source of failure, and the method attempts to make sure that this aspect is not neglected in testing. If a statement has never been executed during testing, for example, it could be radically wrong, yet its fault would not show. Or, a required software function could be implemented very badly--perhaps not implemented at all--yet the error will not be detected if testing fails to invoke that function. Random testing, on the other hand, makes no claims to cover anything, except insofar as chance dictates. It is therefore unexpected that random testing can compete with systematic testing in its ability to expose failures. But that is exactly what studies have shown: under assumptions not unfavorable to systematic methods, they are not much better at finding failures than is random testing (Duran, 1984), (Hamlet, 1990).

The comparison of systematic methods with random testing is a useful exercise in its own right, since it has the potential for evaluating the systematic method to the random-testing standard. Systematic methods are often intuitively appealing, but their significance is in doubt, because they cannot be analyzed in any precise terms that are meaningful for software quality. To take an all-too-common example, many test standards require that (say) 80% of program paths (making some special provision for loops) be executed during testing. Yet there is no information on the effectiveness of 80% coverage at finding failures, nor on the significance of an 80%-coverage test that happens to succeed. The test standard is made from whole cloth, without any scientific basis beyond the intuition that covering 80% of something is better than nothing. If, on the other hand, a comparison with random testing as the standard were available, it might help us to write better standards, or to improve the significance of systematic methods.

The two studies cited above contrast "partition testing" with random testing. A partition test is an abstraction of any systematic method that divides the input space into disjoint subdomains, and requires that tests be selected from each subdomain. Path testing is such a method, because the equivalence classes of inputs defined by the relation "execute the same path," do constitute a partition. Assuming that the software under test has a well defined failure rate (see section 2.5), the

probability of finding at least one failure by random testing depends only on the number of test points. The probability of finding at least one failure by partition testing depends also on the distribution of failures over the subdomains, which is in general unknown. But by making empirical choices for this distribution, the methods can be compared. The results are:

It is difficult to force partition testing to be radically better than random testing, no matter how the subdomains fall relative to random choices of test points. That is, in general there is little to choose between the failure-finding ability of the methods.

Partition testing is typically a bit better at uncovering failures; however, the difference is reversed by a modest increase in the number of points chosen for random testing. Roughly, by taking 20% more points in a random test, any advantage a partition test might have enjoyed is wiped out.

The only way to give partition testing a clear advantage over random testing is to choose a partition in which some of the subdomains have substantially higher failure rates than the others. That is, good partition testing concentrates failures within its subdomains.

If partition testing is about the same as random testing, then the systematic testing methods in common use are not in fact much use for improving our trust in tested software. Typical systematic unit tests have on the order of 100 points in a testset. If such a testset were random, it would guarantee a MTTF of only about 62 runs (80% confidence level)--not very high quality.

The advice to seek partitions with high-failure-rate subdomains sounds like no advice--if the tester knew where failures lurked, no testing would be needed. But it does imply that subdividing the input space "for subdivision's sake" is not wise. If a partition is refined just to make the subdivisions smaller, without reason to believe that failures are being isolated in the smaller subdivisions, then the systematic method is not improved. It also suggests that potential flaws in development methods are a good source of subdomains for systematic testing. For example, specification changes that affect a software unit are likely to make that unit suspect (particularly if the change comes late in the development cycle, and involves the interface). A partition can be

defined that isolates use of the unit to a subdomain in which the failure rate may be very high.

There is an intuitive explanation for the surprising success of random testing relative to its systematic competitors. Systematic methods are more subjective than they appear, because in most cases there are an infinite number of ways to satisfy the method, and the choice is usually made by a human being. There are good and bad choices, relative to finding failures. Random testing can be viewed as replacing human choice with chance selection, including the possibility of no selection at all. For example, randomly selected test points *may* invoke all functions, all statements, etc., but the possibility always exists that some such elements could be missed. These random choice are not necessarily inferior to the ones made by human beings, since in practice a person may not have the information or skill to make a wise choice. The systematic method may not even be helpful to the human tester, because the coverage it requires may not be very useful in exposing failures. It might, for example, be better to try some functions many times, even if this means missing others all together. Random testing can make such choices, while systematic testing cannot.

#### **4. Critique of Random Testing**

Random testing has only a specialized niche in practice, mostly because an effective oracle is seldom available, but also because of difficulties with the operational profile and with generation of pseudorandom input values. It is beginning to be used in theoretical studies, which requires critical attention to its assumptions.

##### **4.1 Practical Issues**

It can be argued that an operational profile is an essential but neglected part of system requirements analysis, and that if proper attention were paid to this aspect of program development, random testing would benefit along with the developer, who would have a much better idea of what the system must handle. For unit testing, this argument is invalid--it is an essential feature of many

software components, particularly those designed for reuse, that *any* usage is acceptable.

Conventional random testing is inappropriate for such units, because its predictions are in general incorrect. The best one can do is test with a uniform distribution, and in some unanticipated application, the unit may be subjected to a usage profile that concentrates heavily on part of the input space, for which the predictions can be arbitrarily over-optimistic. Random unit testing may still be used to uncover failures, as described in Section 3. However, existing testing theory indicates that unit testing to establish software quality is inferior to a combination of formal and informal inspection methods.

On the other hand, the oracle problem is not one that better requirements analysis should necessarily address. It is not always the business of a software end user to know the answers so that the software can be checked. In some cases, it behooves a user to conduct an extensive acceptance test of software, for example, by running an accounting program in parallel with a system it is to replace. But such tests are not random. Often, although computer solutions can be judged in some final way by users, the judgement is of no use in testing, particularly in random testing. For example, a plane that crashes because the program calculating stress in wing structures gave incorrectly low values, or a chemical plant that runs away because a process-control program missed an interrupt, are obvious failures. But the ability to see the obvious is no help in random testing of the programs, when nothing is flying or flowing. This argument applies to all testing, but to random testing with particular force, because the number of test points is large in random testing. Thus it cannot be argued that the user's ability to calculate expected results by hand (which is surely enough to be able to say that the application is well understood), constitutes an oracle. Similarly, a simple, slow algorithm is not an oracle for a faster production program, because there isn't time to use it. In short, in many applications the user wants a few results, with high confidence that they are correct; random testing can guarantee this only by using a large test set, for which the only fast enough computation comes from the application program itself.



In special cases, however, an oracle does exist:

The program to be tested may be a replacement for a trusted program. The latter can then be used as an oracle.

Checking the results of a calculation may be easier than making the calculation, both in the sense of computational complexity, and in the sense of intellectual control. For example, the program for solving an equation is typically slower and harder to understand than a program that substitutes solutions into an equation to check them. Yet the latter is an oracle for the former.

Some computations are paired with their inverses, as in a code/decode and protocol applications. An "end-to-end" test can then be conducted by applying both functions and insisting that the input be reproduced as result. This procedure can be incorrect, because compensating failures can occur in computation and inverse, but a combination of program inspection and random testing can make it unlikely that such failures exist.

In multi-version programming, several programs to solve a common problem are developed independently. It is not correct to run these programs against each other as oracles, because all might fail at the same test point. Nevertheless, many failures are not common, and the more programs there are, the less likely that all will agree accidentally.

Some formal specifications, notably for abstract data types, can be mechanically executed to provide an oracle. This is the most general solution to the oracle problem, but also the most problematic, since the executable specification is usually very inefficient, and it may produce results in a domain difficult to compare with that of a conventional program.

Sometimes a physical system can be constructed to model the results of a computation, and measurements of this system can serve as an oracle. An interesting special case is the use of human vision as oracle for pictorial output. A person can "tell at a glance" if a complex colored picture is substantially correct, and this ability can be used in random testing of graphics programs (Taylor, 1989).

When an oracle exists, random testing is a possibility; without an oracle, it is not.

The final difficulty in applying random testing concerns generating pseudorandom values. The input values that make up test sets are not always from a finite collection. Of course, every computer program input must be limited to a finite number of possibilities, since inputs use some medium of limited size for storage or expression. But in practice the limits are so large that subdomains might as well be infinite. For such large spaces, it is not clear what a pseudorandom value means.

An example will illustrate the difficulty. Suppose that one is testing a compiler, so that inputs are programs. What constitutes a "random" program? The problem is partly that of the operational profile in disguise: one must have information about what programs will in fact be compiled, and in particular about the proportion of correct programs, those with syntax errors, and the range of sizes to be expected. But no matter how detailed this usage information is, the subdomains of the operational-profile partition will be large and somewhat vaguely defined, for example, "Small programs of 100-200 statements, containing mostly simple assignment statements and conditionals, with less than 20 simple variables." There seems no possibility of generating programs at random but to define as many small, orthogonal intervals as possible, make a succession of uniform pseudorandom choices from each interval, and observe constraints among them. In the example, one might select a size from [100,200] (say 163), a number of assignments from [130, 150], (say 137), a number of conditionals from [2,26] (say 20), a sequence of 6 other statement types from {i/o, loop}, and a number of variables from [3,20] (say 7). These parameters have involved a number of relationships and arbitrary choices for the intervals. However, there is still a long way to go before a test program is obtained: one must choose particular identifiers for the variables, the form of assignments, etc., and the whole must be a syntactically correct program according to the language grammar. It seems unreasonable to insist that distributions be provided in the necessary detail; but if they are not, it is impossible to believe that the resulting tests will be random in any sense but the slang one that introduced this article. Thus random testing may be precluded by an inability to make reasonable pseudorandom choices.

The pseudorandom number generators provided with most languages and systems usually compute word-sized bit patterns that can be interpreted as approximations to reals  $r \in [0,1]$ . Other floating-point intervals can be obtained by scaling and offsetting  $r$ . A pseudorandom integer in  $[n,m]$  can be obtained as  $r \bmod (m-n+1) + n$ . A caution is in order, however. Many systems do not provide an adequate pseudorandom number generator, and deficiencies are not documented. For example, the generator may have short cycles or even fixedpoints in the sequence it generates; the low-order bits may not themselves form a random sequence (so the use of the mod function to obtain integer sequences is unwise), etc. (Knuth, 1981) describes both the generation and testing of pseudorandom sequences.

In summary, the best practical application of random testing, one that can supply a plausible reliability figure for software from a successful test, requires the following:

The test is at the system level.

There is both an operational profile and the ability to generate intuitively representative pseudorandom values in its subclasses. (Or, equivalently, an operational distribution on the input space is known.)

An effective oracle is available.

When these conditions do not exist in the testing situation, not only is random testing counterindicated, but testing of any kind cannot provide a plausible prediction of software quality. Tests may uncover failures so that many defects are fixed, but confidence that the software will behave properly in normal use is attainable only by using methods other than testing.

## **4.2 Theoretical Deficiencies**

Most criticism of random testing is really objection to misapplication of the method using inappropriate input distributions, as indicated in section 2.4. Using a uniform distribution because the operational profile is not available, or because no single profile is appropriate, may represent the

best a tester can do, but when its predictions fail, the fault does not lie with the random-testing method. Similarly, testing interactive or realtime software as if it were batch software, is simply engaging in wishful thinking. Unfortunately, the practical character of testing and its weak theoretical foundations encourage the idea that much effort is a substitute for well founded techniques. The software must be tested, so we do the best we can and hope for the best. The worst misuse of random testing occurs when the method is not used at all; a systematic method is used instead, and claims for reliability (or even correctness!) are based on its success.

However, fundamental theoretical issues cloud even the most careful use of random testing. These issues can all be traced to the program input space, and the use of an operational profile there.

It was argued in section 4.1 that for certain unit tests, notably those of reusable components, no operational profile is appropriate. It is tempting to argue that this lack excuses the use of a uniform test distribution, perhaps with more than the usual number of test points. But a large input domain and an application that emphasizes only a small part of it, make this position untenable. The uniform-distribution test predictions can be easily wrong by many orders of magnitude--the random test might just as well not have been done. (Put another way, orders of magnitude more uniform-distribution tests will be required to make accurate predictions about the application. But there are already too many points in the random testsets to be practical.) The same argument can unfortunately be applied at the system level. Each human user may have a quite different profile; and, each person's profile may change over time, quite discontinuously. When one profile has been used for random testing, the results may be arbitrarily incorrect for another profile. A simple example contrasts "novice" and "expert" users, who utilize the same software in quite different ways. It is a common experience that a novice can immediately break software that experts have been using for a long time without problems. The novice's first attempt is an input far outside the experts' operational profile. ("What did you do that for?! No one is dumb enough to try that!") Again it is tempting to use a uniform test distribution or some other compromise between possible profiles. But for general-purpose software, it may be a better description of reality to treat each

user, perhaps each input from each user, as a unique special case rather than a sample from a distribution. And again the practical difficulties will not permit orders-of-magnitude over-testing to compensate for the wrong distribution.

The second difficulty is more fundamental, and illuminates the problems with an operational distribution. The inputs selected according to an operational distribution may not be independent in the sense needed for random testing, thus invalidating any prediction of the reliability theory presented in section 2.5. The difficulty is that the process by which failures result from software defects is not really a random one, but systematic, so that failures cannot be treated statistically.

We might suspect that there is a mismatch between the statistical theory and actual software failures, because the theory fails to predict important observed facts. For example, a software "law" that is observed empirically is that larger programs have more defects in rough proportion to their size (measured in source statements). This observation is easy to account for intuitively, since the defects result from errors made by human designers and programmers, and such errors occur when people are overwhelmed by complexity. Thus program size should enter the reliability model as a parameter, but as section 2.5 shows, it does not. The theory predicts exactly the same reliability for a 20-line program computing roots of a quadratic equation as for a 200,000-line accounting system, if each is tested with the same number of points. The result seems intuitively wrong because we believe that there are more nooks and crannies in larger programs, which should take more test data to explore.

For the phenomenon of software failure to be treated as a random process relative to test inputs, there can be no failure correlation between inputs. To take the most extreme case, when a random phenomenon is being sampled, repeating a sample is meaningful because no two samples are really the same--the random variations that are being sought may act to make them different. But software is not random in this sense, and intuitively it does not give more confidence in a lower predicted failure rate to repeat the same test over and over. Unfortunately, apparently independent inputs can constitute "the same test" for the purposes on exposing a failure.

Consider the simple situation in which there is a fault in an arithmetic expression within a large program. The integer variable  $x$  is incorrectly used in the expression, so that for some values of  $x$  the expression takes the wrong value, and hence the program fails.  $x$  happens to be an input variable restricted to nonnegative values, but it also suffers a "state collapse" just before its use in the faulty expression. A state collapse (Voas, 1990) occurs when a program statement vastly reduces the potential values for  $x$ . For example

$x := x \bmod 5$

reduces the possible values for  $x$  to the set  $\{0,1,2,3,4\}$ . Of these, suppose that only 0 excites the fault in the following  $x$  expression. Then all of the  $x$  input values in (say)  $\{5x+2 \mid x \geq 0\}$  are the same relative to this fault (they do not excite it), yet the input distribution may select many "independent" values from this set. Intuitively, success on many such "independent" inputs gives false increments in confidence that the program will not fail.

Many similar examples can be constructed, in which relative to a particular program fault, many inputs are "the same test." Consider the faulty code as a "unit". Its use within the system has a peculiar operational distribution, very different from whatever distribution is applied as system input. Thus the predictions from the input distribution are incorrect, just as predictions from a unit test using one distribution are incorrect for an application in which the unit has a different usage distribution. Since the random testing method continues to send software back for repair until a test set succeeds, the correlated points, ones which are really the same relative to any failures that are not detected, are finally all success points, so the predictions are overly optimistic.

## 5. Current and Future Research

In this section we briefly mention research ideas that will likely influence the theory and practice of random testing in the future.

A theory of uniformly sampling the program's entire state space, that is, the possible values of all its variables throughout execution, has been proposed (Hamlet, 1988). This theory is still subject to criticism like that that ends section 4.2 (Morell & Voas, 1990), because state values may also be correlated. To sidestep many of the difficulties connected with the operational distribution for programs, it has been proposed that program states be randomly perturbed and the change in results observed (Miller, 1992). When results resist change through perturbations of a state, that state is called "insensitive." An insensitive state suggests that testing cannot detect failures there, so test points are likely to be correlated.

Successful application of probabilistic ideas to calculation and proof hold promise for testing in the future, testing that may use statistical methods, but will differ profoundly from what we now call random testing. The theoretical drawback in random testing, is that test success in the presence of potential failure is not necessarily an unlikely event, nor are multiple tests that miss a failure necessarily unlikely. In contrast, the probabilistic proof that a number is prime, by each probe that fails to show a number composite, reduces the probability that it is composite by a factor. Thus a few probes demonstrate a high probability that the number is prime. In testing, it is difficult to contrive a method where probabilities multiply in this desirable way. One suggestive research project is investigating "self-checking" algorithms (Blum, 1989), ways of invoking a program redundantly so its multiple consistent results make it very unlikely that any is wrong.

Finally, it has been suggested that the definition of "random" in testing be changed to the Kolmogorov notion of "difficult to recognize algorithmically." A test that is truly random in this sense, can be shown to be foolproof (Podgurski, 1991).

## **References and Bibliography**

Blum, M and P. Raghavan, Program correctness: can one test for it?, In G. X. Ritter, ed., *Proc. IFIP '89*, North-Holland, 1989, 127-134.

Butler, R. W. and G. B. Finelli, The infeasibility of experimental quantification of life-critical software reliability, *Proc. ACM SIGSOFT Conference on Software for Critical Systems*, New Orleans, December, 1991, 66-76.

Duran, J. and S. Ntafos, An evaluation of random testing, *IEEE Trans. Software Eng.* SE-10 (July, 1984), 438-444.

Hamlet, R. G. and R. Taylor, Partition testing does not inspire confidence, *IEEE Trans. Software Eng.* SE-16 (December, 1990), 1402-1411.

Hamlet, R. G., Probable correctness theory, *Info. Proc. Letters* 25 (April, 1987), 17-25.

Knuth, D. L., *The Art of Computer Programming, vol 2: Semi-numerical Algorithms*, 2nd Ed., Addison Wesley, Reading, Mass., 1981.

Miller, K., L. Morell, R. Noonan, S. Park, D. Nicol, B. Murrill, and J. Voas, Estimating the probability of failure when testing reveals no failures, *IEEE Trans. Software Eng.* SE-18 (January, 1992), 33-44.

Morell, L. J. and J. M. Voas, Inadequacies of datastate space sampling as a measure of trustworthiness, *Software Engineering Notes*, 1991, 73-74.

Musa, J. D., Faults, failures, and a metrics revolution, *IEEE Software*, March, 1989, 85,91.

Parnas, D. L., A. van Schouwen, and S. Kwan, Evaluation of safety-critical software, *CACM* 33 (June, 1990), 636-648.

Podgurski, A., Reliability, sampling, and algorithmic randomness, *Proc. Symposium on Software*



*Testing, Analysis, and Verification (TAV4)*, Victoria, BC, October, 1991, 11-20.

Shooman, M. L., *Software Engineering, Design, Reliability, and Management*, McGraw-Hill, 1983.

Steele, G. L. Jr., et al., *The Hacker's Dictionary*, Harper & Row, New York, 1983.

Taylor, R., An example of large scale random testing, *Proc. Seventh Annual Pacific Northwest Software Quality Conference*, Portland, OR, September, 1989, 339-348.

Voas, J. M., Preliminary observations on program testability, *Proc. Ninth Annual Pacific Northwest Software Quality Conference*, Portland, OR, October, 1991, 235-247.