Technical Report CS/E 85-003 February, 1985

Release Testing for Probable Correctness

Dick Hamlet
Oregon Graduate Center
Beaverton, OR 97006
(503) 645-1121

Abstract

A plausible, probabilistic theory for *release testing* of software is presented. In a release test a given program is assessed and testing ends. The theory gives the number of test points that must be used to guarantee correctness with a given probability. Its strengths are: nothing need be assumed about the distribution or likelihood of failures or faults *a priori*; and, it is a true correctness theory, not one predicting future behavior by sampling a supposed "operational distribution. Application of the theory to partition testing and structural testing suggests ways to decide when and how those methods should be used.

1. The Many Kinds of Testing

Testing of computer software occurs in a wide variety of situations, each with different goals. Initial unit tests are often part of debugging, for local diagnosis and fault correction. When software is first integrated, testing may provide input to a model of fault repair that seeks to predict the effort needed to reach an acceptable level of quality. And finally, when software is released, test performance is pure assessment: at best no failures are found, but in any case the purpose of the test is to gain confidence in the decision to stop testing. Theories appropriate to these distinct goals also perhaps are distinct. Debugging theory may require a model of the programming process that includes the important psychological contribution of the human programmer [1]. The fault-correction process is so complex that an empirical model with many parameters for fitting complex behavior may be appropriate [2].

Theories of testing have often lacked a clear focus [3]. An "absolute" theory views the program and its specification as a puzzle, and tests as probes intended to solve it. The object of testing is to find such clever tests that the puzzle can be unravelled, the program can be shown by the tests to be correct. Although in special cases of restricted languages [4] or special knowledge of the form the correct program must take [5, 6] the absolute view has been successful, in general it must fail because the problem of fixing the infinite behavior of a program with a finite number of test cases cannot be solved [7]. A "debugging" theory seeks to detect errors that commonly occur in programs. Tests are so chosen that often-made blunders will be caught. Examples of such schemes are structural testing (for a recent example see [8]), in which the program's parts are exercised; and domain testing based on the specification, which seeks to catch out the programmer who has omitted cases or confused boundary conditions [9]. These debug methods are very useful--indeed, faced with the actual unit testing of real software, no other systematic method is available--but they cannot claim theoretical validity. To see this, imagine the programmer as an antagonist. If the debug methods to be used on a program were known, the program might be adjusted to pass its tests, yet still be incorrect.

The release test represents a clean situation. In the simplest case the test exposes no errors. The goal of a release-test theory is to assess the confidence inspired by this success, as the debug theories cannot. One may hope to realize this goal without considering the process by which the software was developed. Any useful theory of release testing must be probabilistic in nature. Testing often stops only because resources, patience, ingenuity are exhausted. The quality of the resulting test can be expected to vary, so success has varying implications for the confidence to be placed in the program. An absolute theory can only call most such tests failures, and cannot

distinguish the better from the worse.

2. Tests as Samples of Program Behavior

"Probabilistic" testing theories have not been popular (but see [10, 11]). The idea that tests are samples allowing statistical statements to be made about a program seems unexceptionable, but there is controversy about the significance of these statements. Existing models and analogies are flawed.

2.1 Input data Space

The most straightforward statistical view of testing sees the test inputs as drawn from the entire input space as samples. By observing the program's behavior on these samples, predictions can be made about its behavior on an arbitrary sample not yet drawn. Samples must be independently selected, and the likelihood that a given input be selected must conform to actual usage. There is an "operational distribution" for any program, the probability F(x) that input x will actually be used. For a sample drawn according to this distribution, standard methods can be used to predict the probability of failure, and the confidence to be attributed to this prediction, based on the failure behavior of the sample. For 1 - e confidence that the probability of no faults is at least p,

$$\log e / \log p$$

points must be drawn from the distribution without a failure [11]. For example, 1-.1 confidence that the probability of no faults is .95 requires 45 test points. The double probability in the theory is a flaw, since it is not clear whether one should seek high confidence in a modest probability, or modest confidence in a high probability; a single number would be better. The result should depend on the size of the input domain, and on the size of the program, but does not. Finally, the numerical value seems much too low.

Of course, if the operational distribution for a problem is not known, it cannot be used to determine test samples. Another distribution (say the uniform one) cannot lead to accurate predictions, because it may draw points from an unlikely part of the domain, or fail to concentrate them in a likely part. Indeed, for typical very large input spaces,

the uniform distribution is so unlike the actual operational distribution that it is unlikely to select *any* point that could actually occur. For example, in testing a program whose input is a radar signal, the difference between actual signals and uniformly distributed random tests is dramatic, because the actual signals have a substantial structure the overwhelming majority of "possible" inputs lack. Similarly, regularities in each record field, and in the relationships between fields, make a typical payroll master file very unlike randomly generated data. In some cases enough information may be available about the input space to approximate the operational distribution. For the radar one can use live signals, and for the payroll, historical files.

Sampling according to an operational distribution has a conceptual flaw more serious than the uncertainty of the distribution. Programs often solve problems in which *no* distribution describes the usage. When the program is large and complex, its inputs may all be special cases in the sense that behavior on one input is unrelated to that on others. Then sampling is a poor idea, precisely because tests are samples not so much of the input space, but of parts (or combinations of parts) of the program. Since code is produced under widely varying circumstances, knowledge about the quality of one part predicts nothing about another part. The statistical theory assumes a uniformity in the process being sampled, such that good behavior often repeated in one area has implications for behavior in a seldom-used area. This assumption obviously fails for a program comprising many special cases.

Thus one never knows if the right distribution has been used, or if any distribution is appropriate. It is a common experience for software to pass stringent tests, as closely duplicating the actual operating environment as possible, yet fail badly as soon as it is used for live data.

2.2 Time Sampling, Mean Time to Failure

Some programs are continuously in operation, so that the idea of sampling from an input space seems to apply to them less well than sampling over the time domain. Such programs are often called "real time" because they respond to demand input from the environment. Operating systems and remote inquiry/control systems are important examples, and most "embedded" systems and process-control systems are of this kind. In the most common statistical testing scheme such a program is put into operation and the time distribution of its failures observed [12]. Conclusions are then drawn about the likely time until the first failure, should the program be restarted afresh. If the program were a natural process, operating according to unknowable laws, this would be the best

one could do.

However, the idea of inputs and an operational distribution is really behind time sampling, too. All time intervals of program execution are not equally significant, and unless the program is responding to typical driving inputs (that is, drawn from an operational distribution), the statistical behavior observed across time is irrelevant to actual operation. In some cases this is obvious. For example, an operating system that is only idling--no tasks are changing status or resource usage, and things have fallen into a regular pattern--can be run indefinitely without collecting any real information on its likelihood of failure. Similarly, when a process-control program is handling only priority exceptions that override normal complex operation, its behavior has little predictive power.

Thus time-sampled theories have the flaws of those requiring an input operational distribution, only hidden behind observations in the time domain.

2.3 The Analogy to Quality Control Sampling

The analogy between production of software and production of so-called durable goods is an attractive one. Because the latter is better understood, there is a continual call for making software as appliances are made, with consequent improvement in quality and productivity. The language reflects our desires: "software factories," "software blueprints," and ultimately, "software quality assurance," are meant to imply that the desirable characteristics of the analogous manufacturing process have been captured. Most of these phrases are empty. An analogy is useful when it shows what to do (and how to evaluate the result) in an unknown field. A clarion call for (say) "software assembly lines" is merely silly without some idea of what these might be in practice. It is desirable to replicate Henry Ford's success, but Ford did not succeed by (say) adapting water power to the task, however well this had worked to industrialize textile mills.

Analogies can be counterproductive, too. When the correspondence is weak or spurious, thinking about an unknown field in familiar terms only hides the actual principles of the new subject. There is reason to believe that the "factory" or manufacturing analogy to software production is a misleading one, if only because the complexity of software is so much larger than anything else in the man-made world, and the process so at the mercy of human vagaries. Successful manufacturing relies on simplicity and strong physical laws central to the production process, neither present for

software.

Quality assurance is a form of sampling intended to apply to mass production of functional objects. The objects manufactured are supposed to do something, and it is the task of quality assurance to see if indeed they do. This is necessary for mass production as it is not for hand-crafting because the steps of production do not directly relate to the intended function. The assembly line station where part of a machine is installed has rules of operation, but the worker there can only tell if the rules are being followed, not if the complete machine is going to work.

Quality assurance selects complete units (or subunits in complicated cases) created by the mass production process, and tests them (perhaps destructively) for the desired characteristics. When the samples are selected at random (that is, without correlation to the production process), the expected failure rate and the confidence in that rate can be calculated. The analogy to software development and testing seems clear: software parts are created to designs that those writing code do not know fully, so the integrated software must be tested for the behavior desired. The only difficulty is in the sampling: there is only one unit produced. Since software is intended to have very complex behavior, it is also difficult to see what is analogous to testing of a unit pulled from the assembly line. Running an operating system through its paces is not very similar to putting a washing machine through the cycle. However, these two difficulties appear to cancel: a typical software test is a sample, and this reintroduces the statistical element. The complete analogy is thus that tests are conducted at random on completed software, and the expected failure rate is predicted from the test results.

Again, the operational distribution enters, because it is evidently necessary that tests represent typical cases. The analogy is thereby called into question, because this idea has no counterpart for assembly lines. The correspondence would be better if a single manufactured unit were subjected to many partial tests.

The cause of assembly-line failures is assumed to be in components or their assembly, each such difficulty present with a small probability. This probability is over the space of all components or assembly operations, and this space is appropriately sampled so long as there is no correlation between units selected for test and component circumstances. There would be such a distorting correlation, for example, if every tenth complete unit is selected for quality assurance testing, and some component arrives at an assembly station in batches of ten, in a fashion that damages the last to arrive. Then the unit assembled from this last, damaged component is likely to malfunction, but the quality-assurance sample will be misleading. If every damaged unit is taken, it will show

that all units fail; if the cycles of ten are not aligned, it will show that no units fail.

The cause of program failure is not so clearly an identifiable component of the software. For example, which part of a system has failed if a typing error excites a case that the system is not required to handle, and for which code has not been provided? However, in most cases even design flaws of omission can be attached to some portion of the program. A subroutine that fails to provide for some potential input is something like a physical component with improper ratings, and can be said to fail. However, the appropriate way to sample and detect failures of this kind is not clear. Obviously one cannot sample the "component space" of all possible routines, since that space is very large and only one member is available.

Test data points for a system as a whole provide only a distorted probe into the system's parts, because by the time values reach a deeply buried part, they have been conditioned in a way that makes them heavily correlated. For example, uniformly distributed system inputs may very well result in only a single test value being sent to some internal routine; it does not augment the routine's sampling to enlarge the system sample. The difficulty is that we cannot sample the space of interest, but must use a probe into that space (the input) that does not necessarily treat the members fairly. As an analogy to the more usual probability situation, suppose one were trying to determine the color of balls in an urn by sampling, but the balls could only be selected by a peculiar mechanism. If it were deduced that all were silver, it would seriously compromise the result to learn that the balls were polished brass and steel, and the "random" selection used a magnet. It does not help to use an operational distribution in selecting inputs, because this does not remove the bias in the component space.

Thus the component model of failure and the idea of sampling over a system input or time space are fundamentally at odds, and the analogy to durable-goods quality assurance is not useful.

3. Probable Correctness

A program is correct if no input exists for which it fails. By selecting independent inputs at random and observing their success or failure this property is sampled directly. In the simplest case all tests succeed, and we wish to know the significance of this result: how likely is it then that the program is correct? The samples are drawn from a pool of potential executions, each either a success or failure; the continued absence of a drawn

failure makes it more and more likely that there are none.

The distinction between "probably correct" and "unlikely to fail in use" is an important one. If the sample pool is one of success and failure, and inputs are only the unbiased mechanism for drawing from this pool, then no operational distribution is needed. What we seek is the probability that no failure points exist to be drawn, not the probability that some pattern of use will not excite a fault.

In a different context, Valiant [13] has found a clever way to analyze exactly this situation. Consider sampling a space seeking some property of its objects. Define N(h, S) as the minimum number of Bernoulli trials needed, each with a probability at least 1/h of success, to force the probability of having less than S successes to below 1/h. If a space contains K kinds of objects, and probability at least p is required that each of them has been detected by a drawing, Valiant shows that N(1/(1-p), K) independent samples suffice. The application to testing is appealing because nothing need be assumed about the frequency of occurrence for the objects being sampled, nor about the drawing. That is, we need not know if the program can fail, and need not assume an operational distribution. Valiant further shows that

$$N(h, S) \leq 2h(S + \ln h)$$

for all $S \geq 1$ and all $h > 1$.

For a release test the objects are program executions, each identified with an input, each either a success or failure. Thus $K = 2$, so for example if .99 probability of correctness is required, it will suffice to take

$$N(1/(1-.99), 2) \leq 200(2 + \ln 100) \simeq 1320$$

independent samples without a failure. For .9999 probability of correctness,

$$N(10000, 2) \leq 112{,}100$$

samples are required.

Intuitively, the number of tests required to obtain a certain degree of confidence in a program ought to grow with the program size, and with the size of the input domain; the theory presented above does not properly depend on these parameters. Furthermore, without recourse to the idea of predicting future behavior (and hence again falling back on an operational distribution), how can an appropriate level of correctness be

determined? These questions can be addressed by changing the sample space.

3.1 Domain Testing for Release

Consider partitioning the program domain into M parts D_1, D_2, ... , D_M. Then if the sample space is taken to be success/failure within a partition, there are between M and 2M different kinds of objects (partitions might be so chosen that all members either succeed or fail). If N(1/(1-p), 2M) independent samples show only success, the probability of correctness is at least p. More tests (since N(h, S) increases with S) guarantee more: parts of the domain that do not appear in the sample are likely never to appear. An untested partition may be known to be important, however. Then direct testing can be used: samples drawn independently within each partition. The N(1/(1-p), 2) estimate then applies to failures *within* each partition, but a total of MN(1/(1-p), 2) samples are required to gain probability p of correctness over the whole domain. Had these samples been taken overall, it would have improved the correctness probability. For example, the total number of points taken in 85 partitions to guarantee .99 probability of correctness, would give .9999 probability of correctness if chosen disregarding partitions.

Duran and Ntafos [11] computed the effectiveness of domain-restricted tests vs. overall random testing. It is difficult to evaluate the many assumptions they required for the calculation, but their result was that there is little difference. Here we can distinguish two cases:

> If overall sampling reaches all partitions more or less uniformly, partitioning is not useful. But there is no reason to believe in uniform coverage *a priori,* and no way to verify it after the fact. In a release test where only success is observed, the distribution of success observations shows nothing about the failures, and hence nothing about the partition coverage.
>
> On the other hand, suppose that some partitions may be neglected to an arbitrary degree by sampling that ignores partition boundaries. The assumption that failures are uniformly distributed across partitions is reasonable if partitions arise from specifications, so that none represents a harder programming problem than the others. Then no global sample, however large, can give the degree of confidence that samples within partitions can.

There is a common-sense explanation of a lower confidence within partitions requiring more sample points. If the space *is* one of success and failure within partitions (as it is under the assumption of failures uniformly distributed among them), then the global confidence result is simply wrong. In this theory the only parameter is the space over which the test samples are drawn; hence a conflict in results must be resolved by selecting the right space. If there is reason to believe that failures will be distributed uniformly across partitions, and that some partitions are under-represented in a random drawing from the whole domain, then it makes sense to test those partitions separately. On the other hand, if little is known about failures by partition, greater confidence can be obtained from uniform random testing. In either case, the level of correctness can be selected using an estimate of the likely failure rate. When experience shows that faults are likely, a low level of probable correctness suffices. For example, if the code for some partition has proved difficult to get right in the past, then that partition need only be tested at a low probability. On the other hand, where a complex domain is difficult to divide, and there is reason to believe that the code is of good quality, a higher probability overall is appropriate.

The often-suggested partitioning idea that inputs within each partition should require or receive "the same treatment" (all tests should either succeed or fail in a domain) seems to be irrelevant in this theory.

3.2 Structural Testing for Release

Intuitively, the textual parts of a program are its "components," each perhaps faulty. The granularity of intuitively correct components is a tricky subject. For example, components must be allowed faults of omission. A statement missing is not a fault in statements near where it should be; but, it could be thought of as a fault in a procedure that should contain it. However difficult it is to define plausible units, a good bound on their number probably lies between the number of procedures and the program line count.

An input may invoke several fault units, however. Should the sample space be defined by units, by the power set of units, by all sequences of units? This question can be answered by considering the negative side of a successful release test. The objects *not* discovered in the drawn samples are failures and successes *of a certain kind,* depending on the choice of sample space. A good space is one in which the statement "this kind of failure will probably not occur" is worth what it costs. When more kinds are considered,

more tests will be required to reject them at any given probability level.

First consider fault units alone. For one particular unit S, the space is success/failure/S-invoked/S-not-invoked. A successful random sample of size N(1/(1-p), 4) would then guarantee probability p that failure with S invoked cannot occur. This same sample, however, would predict the same for any other fault unit. If failures in the units were independent, this test would yield probability p^L that no unit fails, among L units. That is, the units must be tested to correctness probability $p^{1/L}$ to guarantee probability p for the independent collection. Taking p = 1-e, and keeping only the first two terms of the binomial series expansion, $p^{1/L} \simeq 1 - e/L$, gives

$$N(L/e, 4) \leq 2L/e\ (4 + \ln L/e).$$

For a program with 500 fault units (if 20 statements constitute a fault unit, then there would be 10,000 statements), the number of tests to guarantee .99 probability of correctness would be

$$N(1/(1-.99^{1/500}), 4) \leq 1{,}480{,}000$$

selected independently without a failure.

Any coverage technique can be used to define a sample space, but these spaces are large (or infinite) and their significance dubious. For the space of fault units themselves, the result is "the program probably will not fail when any unit is invoked." For sets (*resp.* sequences) of fault units, it is "the program probably will not fail when invoking any set (*resp.* sequence) of units." Viewed in this way, the combination spaces (the sequence one corresponds to path testing) do not seem worth their cost. The sequence space is infinite; even the fault-unit-set space is hopelessly large--2^{L+1} objects for L fault units. (For example, $2^{501} > 10^{50}$.) The ideal structural space would be described by "the program probably will not fail when any fault unit is invoked with any program data state." This space is usually infinite.

The large structural coverage spaces suggest the futility of conventional test coverage methods for release testing. In these methods *one* test is required for each structural unit. Thus in a space of size K, about K samples are taken. They are likely not independent, but even if they were, the probability of correctness is nearly 0. (For K only 500, the value is about probability .002 of correctness.)

Because fault units are only selected indirectly by inputs, there is no direct analog to sampling within domains; the samples cannot be made independent. Hence the tester concerned with unsampled fault units has no recourse but to increase the number of tests

overall; and, as for partitions, success-only coverage cannot be trusted. However, if fault units can be directly executed in isolation, as they might be at the procedure level, then (literally) unit testing can be done. It seems reasonable to assume that faults are equally likely in each unit, and that a random input selection may badly neglect some units. Under those assumptions, unit testing is wise, and the previously calculated estimates can be applied to each procedure. If there are 50 procedures, each with about 10 fault units, the number of tests for .99 probability of correctness in them all is

$$50\ N(1/(1\text{-}.99)^{1/10},\ 4) \leq 1{,}091{,}000.$$

Here the comparison between testing units and testing overall is not unfavorable, in contract to the situation for domain partitioning.

It has been suggested [14] that partition and structural testing be combined by intersecting specification-derived partitions with structural coverage classes. This multiplies the sample space size, in order to establish assertions like, "the program probably will not fail when any fault unit is invoked within any partition." There does not seem to be an intuitive gain in such combinations.

3.3 Practicality of Random Release Testing

The utility of random testing depends on the difficulty of generating test data and judging success/failure in results. Both input and output sides of this question are difficult.

In large or unbounded input domains, arbitrary truncation is required. The distortion that truncation introduces can be intuitively measured by imagining the *program* restricting the domain with explicit boundary tests. The input-generation problem is less severe for smaller units--partitions, or procedures with distinct, simple domains.

Program output is even more difficult to handle: who will examine a million results, and decide their success? Systematic procedures for generating input-output pairs cannot be used, because they conflict with the need to choose independent input points. The only general solution to this problem is an effective specification, one that can be used to mechanically judge results. Existing specifications seldom have this property. An attractive alternative is "dual coding" [15] in which two independently created programs

run against each other under test.

If random testing were more respectable, it is likely that many common-sense, clever tricks would develop around its use. For example, in the absence of an effective specification, heuristic methods could automatically categorize most test results, simplifying the task of human examination. The subject is certainly a candidate for an aritificial-intelligence expert system.

4. Summary and Suggestions for Future Work

A probabilistc theory of release testing has been presented that does not depend on an operational distribution, and cannot be faulted for hidden input-space sample correlation. Because the theory depends only on selection of an appropriate space of objects that tests sample, it can be used to analyze practical methods quantitatively.

Some obvious ways to validate this theory suggest themselves. First, some software systems have been tracked for failures following release, and the early versions of these could be tested to show that the confidence calculated is no better than subsequently found. An easier trial of the theory could take a large, ready-to-release software product, and test it sufficiently to expose faults at the usual level. For example, a program likely to contain one fault per 20 fault units (roughly the rate of carefully software-engineered systems) should not pass a release test predicting .95 probability of correctness.

References

1. K. Ehrlich and E. Soloway, An empirical investigation of the tacit plan knowledge in programming, Research Report 236, Yale University, April, 1982.

2. B. Littlewood, Stochastic reliability growth: a model for fault removal in computer programs and hardware designs, *IEEE Trans. Reliability* R-30 (October, 1981), 313-320.

3. R. Hamlet, Debug testing and confidence testing, TR CS/E 84-004, Oregon Graduate Center, August, 1984.

4. D. Tsichritzis, The equivalence problem of simple programs, *JACM* 17 (October, 1970), 729-738.

5. S. J. Zeil, Perturbation testing for computation errors, *Proc. 7th ICSE*, Orlando, FL, 1984, 257-265.

6. R. Hamlet, Reliability theory of program testing, *Acta Informatica* 16 (1981), 31-43.

7. R. Hamlet, Testing programs with finite sets of data, *The Computer Journal* 20 (Aug., 1977), 232-237.

8. S. C. Ntafos, An evaluation of required element testing strategies, *Proc. 7th ICSE*, Orlando, FL, 1984, 250-256.

9. E. Weyuker and T. Ostrand, Theories of program testing and the application of revealing subdomains, *IEEE Trans. Software Eng.* SE-6(1980), 236-246.

10. J. Duran and J. Wiorkowski, Toward models for probabilistic program correctness, *Proc. ACM Software Quality & Assurance Workshop*, San Diego, 1978.

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

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

13. L. G. Valiant, A theory of the learnable, *CACM* 27 (November, 1984), 1134-1142.

14. D. Richardson and L. Clarke, A partition analysis method to increase program reliability, *Proc. 5th ICSE*, San Diego, CA, 1981, 244-253.

15. D. Panzl, Automatic software test drivers, *Computer* 11 (April, 1978), 44-50.