Foundations of Software Testing: Dependability Theory

Dick Hamlet
Portland State University
Center for Software Quality Research

Abstract

Testing is potentially the best grounded part of software engineering, since it deals with the well defined situation of a fixed program and a test (a finite collection of input values). However, the fundamental theory of program testing is in disarray. Part of the reason is a confusion of the goals of testing — what makes a test (or testing method) "good." I argue that testing's primary goal should be to measure the dependability of tested software. In support of this goal, a plausible theory of dependability is needed to suggest and prove results about what test methods should be used, and under what circumstances. Although the outlines of dependability theory are not yet clear, it is possible to identify some of the fundamental questions and problems that must be attacked, and to suggest promising approaches and research methods. Perhaps the hardest step in this research is admitting that we do not already have the answers.

1. Testing Theory — Testing Art

Glenford Myers' textbook is entitled *The Art of Software Testing* [Myers], and it retains its popularity 15 years after publication. Indeed, testing is commonly viewed as an art, with the purpose Myers assigns to it: executing software to expose failures. Program testing occupies a unique niche in the effort to make software development an engineering discipline. Testing in engineering is the corrective feedback that allows improvement in design and construction methods to meet practical goals. Nowhere is the amorphous nature of software a more evident difficulty: testing is hard to do, and fails to meet engineering needs, because software failures software defy easy understanding and categorization. When a software system fails, often we learn little except that there were too many possible failure modes, so one escaped analysis.

A number of new results and ideas about testing, experimental and theoretical, are emerging today, along with a renewed interest in software quality. There is reason to hope that an adequate scientific basis can be supplied for the testing art, and that it can become the part of software engineering with the strongest foundation. It is right to expect exact results, quantitative and precise, in software testing. Each of the other development phases involves essentially creative, subjective work. Better methods in those other phases are aids and checks to human effort. No one expects, for example, that a specification could be proved to be "99.9% correct," since "correct" has an essentially subjective meaning. But by the time we come to testing, there is a completely formal product (the program), and a precise measure of quality (it should not fail). Testing theory can be held to a high engineering standard.

1.1. Assessing Testing Methods

It is sobering to imagine advising a developer of safety-critical software about how to test it. While there are textbooks and research papers describing many testing methods, there is a dearth of experiments on what works, and there no convincing theory to explain *why* a method should work, or under what circumstances it is indicated. Imagine giving a presentation on testing before the Federal Aviation Authority, and that the FAA would write into regulations its content. How safe would the speaker feel flying on planes whose flight-control programs were tested according to those regulations? I know what I would tell the FAA today: get the smartest test people available, give them whatever resources (particularly plenty of time) they need, encourage them to use any test methods they find useful; *and then*, everyone cross their fingers, and fervently hope the software doesn't fail. There is nothing wrong with our "best practices" of testing, and I would encourage the FAA to require their study and even their use, but to suggest *relying* on today's testing methods is unconscionable. This imaginary advice to the FAA recognizes that software testing today is not engineering. Its principles are not known, so they cannot be routinely learned and applied. Systematic effort and careful record-keeping are important parts of engineering, but they alone are only a facade. The substance is supplied by the underlying science that proves the methods work. In software testing it is the science that is lacking.

A good deal of the confusion in evaluating testing methods arises from implicit disagreements about what software testing is supposed to do. In the literature, it is possible to identify four epochs:

(1) *Seeking failures by hard work.* ([Myers], 1979.) Myers recognized that to uncover failures requires a negative mindset. The testing methods used to expose failure are mostly based on test coverage. An emphasis on systematic effort encourages the naive belief (not claimed by Myers) that when one has worked as hard as possible, the result *must* be all right. By concentrating on the software quality "process," the current fad diverts attention from its technical basis, and confuses following procedures with real success.

(2) *Failure-detection probability.* (Codification of (1), mid 1980s [Duran & Ntafos] to the present [Frankl & Weyuker].) Testing methods were first compared only anecdotally, or in circular terms. Today precise statements about failure-finding probabilities are expected in a comparison. An RADC-commissioned study [Lauterbach & Randall] illustrates the transition: it has a histogram showing that "branch coverage" got the highest coverage [branch!]; but it also measured methods' ability to expose failures.

(3) *Software reliability engineering.* (Beginning with the reliability theory developed at TRW [Thayer+], and currently most popular as models of debugging [Musa+].) Nelson's work at TRW attempted to apply standard engineering reliability to software. Reliability — the probability of correct behavior under given operating conditions for a given time period — is certainly an important property of any artifact. However, the application to software is suspect. Today "reliability" is usually associated with so-called "reliability-growth models" of the debugging process, which have been criticized as little more than curve fitting.

(4) *Dependability theory.* (Current research, e.g., [Hamlet & Voas].) Dave Parnas noted the difference between reliability and what he called "trustworthy" software: "trustworthiness" leaves out "given operating conditions" and "given time." Thus it is most like a confidence estimate that software is correct. I called this idea "probable correctness" [Hamlet87]; here it is called "dependability." Paradoxically, although it might seem harder to test software for dependability than for reliability, it may in fact be easier in practice.

These four ideas of the essence of software testing are quite different, and lead to quite distinct conclusions about what kind of testing is "good." Two people, unconsciously holding the viewpoints say (2) and (3) respectively, can easily get into a fruitless argument about testing.

Proponents of all four ideas lay claim to the goal explicit in (4): everyone wants to believe that software can be trusted for use because it has been tested. Casting logic to the winds, some proponents of the other viewpoints claim to realize this goal. To give a current example, those who today use defect-detection methods claim a connection between those methods and confidence in the tested software, but the argument seems to be the following:

> I've searched hard for defects in this program, found a lot of them, and repaired them. I can't find any more, so I'm confident there aren't any.

Consider the fishing analogy:

> I've caught a lot of fish in this lake, but I fished all day today without a bite, so there aren't any more.

Quantitatively, the fishing argument is much the better one: a day's fishing probes far more of a large lake than a year's testing does of the input domain of even a trivial program.

The first step in testing theory is to be clear about which viewpoint is being investigated. Secondly, an underlying theory of dependability (4) is fundamental. If we had a plausible dependability theory, it might be possible to establish (or refute) claims that test coverage (1) and (2) actually establish dependability.

In this paper I try to identify the problems central to a theory of testing for software dependability, and suggest promising approaches to their solution. Unfortunately, there are more problems than promise. But the first step in scientific inquiry is to pinpoint important questions and to establish that we do not know the answers.

1.2. Questions Arising from Practice

Myers' book describes many good (in the sense (1) above!) testing ideas that arose from practice. The essential ideas are ones of systematic *coverage*, judging the quality of a test by how well it explores the nooks and crannies of program or specification. The ideas of *functional coverage*, based on the specification, and *control coverage*, based on the program's control structure, are the most intuitively appealing. These ideas have appeared in the technical literature for more than 20 years, but their penetration in practice is surprisingly shallow. The idea of *mutation coverage* [Hamlet77, DeMillo78] is not so well regarded. But mutation, a technique of practical origin with limited practical acceptance, has had an important influence on testing theory.

The foundational question about coverage testing is: What is its significance for the quality of the software tested?

Traditional practical testing in the form of trial executions is today under strong attack by advocates of two "up front" software technologies. Those who espouse formal development methods believe that defect-free software can be created, obviating the need for testing. The position is controversial, but can only support the need for basic testing theory. Formal-development methods themselves must be validated, and a sound theory of dependability testing would provide the means. A profound belief in the inevitability of human error comes with the software territory, so there will always be a need for better development methods, and a need to verify that those methods have been used properly in each case. The second attack on testing comes from the competing technology of software inspection. There is evidence to show that inspection is a cost-effective alternative to unit testing; the IBM FSC group responsible for the space shuttle code has found that if inspection technology is used to the fullest, unit testing seldom uncovers any failures [Kolkhorst].

As a consequence of the success of inspections, the promise of formal methods, and a renewed interest in reliability (the three are combined in the "cleanroom" method [Cobb & Mills]), it has been suggested that unit testing be eliminated. Practical developers are profoundly wary of giving up unit test.

Those who advocate coverage methods, attention to formal development, system testing in place of unit test, etc., argue for their positions on essentially non-scientific grounds: "do this because it is obviously a good idea." A scientific argument would require that "good" be defined and convincing arguments be presented as to *why* goodness results. Even if there is agreement that "good" means contributing to dependability, the arguments cannot be given in the absence of an accepted dependability theory. There *is* agreement on the intuitive meaning of dependable software: it does not fail in unexpected or catastrophic ways. But no one suggests that it will soon be possible to conduct convincing real-world controlled experiments. Even case studies are too expensive, require years worth of hard-to-get field data, and could be commercially damaging to the company that releases real failure data. It appears that theory is the only available approach; but we have no accepted theory.

1.3. Foundational Theory to be Expected

The missing theory of testing is a "success" theory. What does it mean when a test succeeds (no failures occur)? It is hard to escape the intuition that such a theory must be probabilistic and fault-based. A success theory must treat the test as a sample, and must be of limited significance. The fault-based view arose from mutation [Morell & Hamlet] and from hardware testing [Foster]. Joe Duran has long been an advocate of the probabilistic viewpoint [Duran & Wiorkowski, Duran & Ntafos, Tsoukalas+].

In brief, a foundational theory of testing for software dependability will make precise the idea of sampling and not finding program failures. New ways of testing may be needed to realize dependability. But a major benefit to be expected is a precise analysis of existing testing methods, and believable answers to the questions about their significance. A good theory would also provide direction for experiment. It is probably impractical to measure dependability directly [Butler & Finelli], but a theory can be supported or disproved by deducing from it less than ultimate results, which can be checked.

In Section 2 to follow, some promising work is examined to show the pattern of an emerging body of analysis methods and results. Section 3 examines the validity of the statistical approach on which these results depend, and concludes that a better foundation is needed. Section 4 explores preliminary definitions of dependability.

2. Examples of Emerging Testing Theory

The driving force behind the emerging testing theory is probabilistic analysis. By accepting that the qualities of tests cannot be absolute, the way is opened for quantifying properties of testing.

2.1. Improving the ‘Subsumes’ Relation

As an example of changing research directions, consider analysis of the data-flow coverage methods. The initial work [Rapps & Weyuker] defined a hierarchy of methods (now usually called the *subsumes* hierarchy) such that if method Z subsumes method X, then it is impossible to devise a method-Z test that is not also a method-X test. The widespread interpretation of "Z subsumes X" is that method Z is superior to method X. (The most-used example is that branch testing is superior to statement testing, because branch coverage strictly subsumes statement coverage.) However, I suggested [Hamlet89] that subsumption could be misleading in the real sense that natural (say) branch tests fail to detect a failure that (different) natural statement tests find. A continued exploration [Weyuker+] showed that the algebraic relationship could be refined so that it was less likely to be misleading, and that it could be precisely studied by introducing a probability that each method would detect a failure. In a recent paper [Frankl & Weyuker], the subsumes relationship is refined to a relationship called "properly covers," and a probabilistic argument shows that "properly covers" cannot be misleading. The all-uses dataflow criterion, for example, properly covers branch testing. This analysis is the first comparison of methods on theoretically defensible grounds. It must be noted that the results apply to failure-detection, not to reliability or dependability.

2.2. ‘Partition’ Testing vs. Random Testing

Practical coverage testing could be called "partition testing," because its methods divide the input domain into subdomains, which constitute a partition in the two important cases of specification-based blackbox testing, and path-coverage structural testing. In the early 1980s, Duran and Ntafos conducted a seminal study contrasting partition testing with random testing [Duran & Ntafos]. Their study was presented as a brief for random testing, but its effect has been to illuminate basic questions of testing theory.

Duran and Ntafos considered two statistical measures of test quality, the probability that some failure(s) will be detected, and the expected value of the number of failures discovered. (The two measures gave similar results; only the former has been investigated subsequently.) Today the statistical treatment, and a goal clearly related to quality, seem merely appropriate; at the time the paper was presented, these were both novelties. The calculations required a detailed model for partition testing, and some simulation to realize the comparison. Despite the authors' care to give partition testing the advantage when assumptions were needed to make the comparison mathematically tractable, the results showed little difference between partition and random testing.

Subsequent work extended the comparison to study *why* partition testing failed to show a superiority that is commonly perceived. The model of partition testing was varied in one study [Hamlet & Taylor]; another [Jeng & Weyuker] used only analytical methods in a simplified setting. The two studies agreed that the failure-detection performance of partition testing depends on the *variability* of failure probability across the subdomains of the partition. If some subdomains are not known to be more failure-prone, then partitioning the input space will be of little advantage.

The combined analytical and simulation techniques pioneered by Duran and Ntafos are easier to use than either pure analysis, or pure experimentation. But they have not yet been exploited to examine questions of reliability or dependability.

3. Software Failure Intensity and Reliability

The results described in Section 2 rely on the conventional theory of reliability. Random testing [Hamlet94] supplies the connection between software quality and testing, because tests sample the behavior that will be observed in practice. However, the viewpoint that tests are statistical samples is controversial. The theory of software reliability, developed at TRW in the 1970s [Thayer+] remains a subject of dispute. This section investigates the question of whether this theory, or any statistical theory, can plausibly describe software failures.

3.1. Analogy to Physical Systems

Distrust of statistical ideas for software hinges on the non-random nature of software design flaws. The software failure process is utterly unlike random physical phenomena (such as wear, fabrication fluctuations, etc.) that make statistical treatment of physical systems plausible. All software failures are the result of discrete, explicit (if unintentional) design flaws. If a program is executed on inputs where it is incorrect, failure invariably occurs; on inputs where it is correct, failure never occurs. This situation is poorly described as probabilistic. Suppose that the program fails on a fraction Θ of its possible inputs. It is true that Θ is a measure of the program's quality, but not necessarily a statistical one that can be estimated or predicted. The conventional statistical parameter corresponding to Θ is the instantaneous *hazard rate* or *failure intensity* z, measured in failures/sec. For physical systems that fail over time, z itself is a function of time. For example, it is common to take $z(t)$ as the "bathtub curve" shown in Figure 3.1-1.

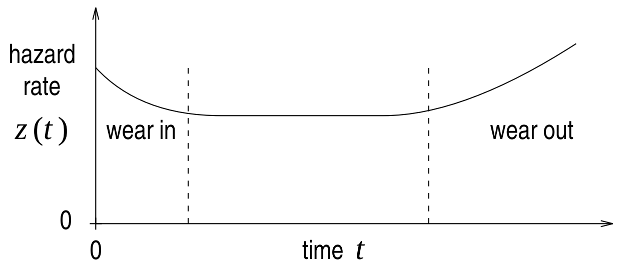

Figure 3.1-1. 'Bathtub' hazard rate

When a physical system is new, it is more likely to fail because of fabrication flaws. Then it "wears in" and the failure intensity drops and remains nearly constant. Finally, near the end of its useful life, wear and tear makes the system increasingly likely to fail.

3.2. Abstraction to the Simplest Situation

What is the corresponding situation for software? Is there a sensible idea of a software failure intensity? There are several complications that interfere with understanding. Because this paper is concerned with fundamentals, it attempts to simplify the situation as much as possible, to abstract away from extraneous issues, without losing the essential character of the problem.

The first issue is time dependence of the failure intensity. A physical-system hazard rate is a function of time because the physical system changes. Software changes only if it is changed. Hence a time-dependent failure intensity is appropriate for describing the development process, or maintenance activities. (The question of changing usage is considered in Section 3.3 below.) Only the simplest case, of an unchanging, "released" program is considered here. Thus we are *not* concerned with "reliability growth" during the debugging period [Musa+].

Some programs are in continuous operation, and their failure data is naturally presented as an event sequence. From recorded failure times t_1, t_2, ..., t_n starting at 0, it is possible to calculate the mean time to failure (MTTF): $(t_1 + \sum_{i=1}^{n-1}(t_{i+1}-t_i))/n$, which is the primary statistical quality parameter for such programs. But MTTF is of questionable statistical meaning for the same reasons that failure intensity is. It is a (usually unexamined) assumption of statistical theories for continuously operating programs that the inputs which drive the program's execution are "representative" of its use. The inputs supplied and their representativeness are fundamental to the theory; the behavior in time is peculiar to continuously operating programs. Exactly the same underlying questions arise for a "batch" program in which a single input instigates a "run" that either succeeds or fails, entirely independent of all other runs.

In summary, we take for analysis an unchanging, memoryless batch program, each of whose runs is instigated by a single input. The quality parameter corresponding to MTTF might now be called "mean runs to failure" (MRTF), and the instantaneous nature of the failure intensity is "per run." A great deal of complication has been eliminated, but the statistically questionable parameters remain.

3.3. Is Software Failure Intensity Meaningful?

If a statistical view of software failures is appropriate, failure intensity (or MRTF) can be measured for a program in an idealized experiment. Inputs are supplied, and the failure intensity is the long-term average of the ratio of failed runs to total runs. The experiment immediately raises the fundamental question of input distribution. If an exhaustive test can be performed, then it is possible to measure the failure intensity exactly. But whether or not failure intensity can be estimated with less than exhaustive testing depends on how the test inputs are selected. It is certainly possible to imagine selecting them to inadvertently emphasize incorrect executions, and thus to estimate failure intensity that is falsely high. The more dangerous possibility is that failures will be unfairly avoided, and the estimate will be too optimistic. When a release test exposes no failures, a failure-intensity estimate of zero is the only one possible. If subsequent field failures show the estimate to be wrong, it demonstrates precisely the anti-statistical point of view. A more subtle criticism questions whether MRTF is stable—is it possible to perform repeated experiments in which the measured values of MRTF obey the law of large numbers?

A partial response to problems in sampling inputs to estimate MRTF is to postulate an *operational profile*, a probability density function on the input space describing the likelihood that each input will be invoked when the software is actually used. If tests are drawn according to the operational profile, a MRTF can be estimated, can be evaluated for stability, and should apply to actual use. In practice there are a myriad of difficulties with the operational profile. Usage information may be expensive to obtain, or simply not available; different organizations (and different individuals within one organization) may have quite different profiles; and, testing with the wrong profile always gives overly optimistic results (because when no failures are seen, it cannot be because failures have been overemphasized!).

The concept of an operational profile does successfully explain changes observed over time in a program's (supposedly constant) failure intensity. It is common to experience a bathtub curve like Figure 3.1-1. When a program is new to its users, they subject it to unorthodox inputs, following what might be called the "novice" operational profile, and experience a certain failure intensity. But as they learn to use the program, and what inputs to avoid, they gradually shift to the "normal" user profile, where the failure intensity is lower, because this profile is closer to what the program's developer expected and tested. This transition corresponds to the "wear in" period in Figure 3.1-1. Then, as the users become "expert," they again subject the program to unusual inputs, trying to stretch its capabilities to solve unexpected problems. Again the failure intensity rises, corresponding to the "wear out" part of the Figure.

Postulating an operational profile also allows us to derive Nelson's software-reliability theory [Thayer+], which is quantitative, but less successful than the qualitative explanation of the bathtub curve. Suppose that there is a meaningful constant failure intensity Θ (in failures/run) for a program, and hence a MRTF of $1/\Theta$ runs. We wish to draw N random tests according to the operational profile, to establish an upper confidence bound α that Θ is below some level θ. These quantities are related by

$$1 - \sum_{j=0}^{F} \binom{N}{j} \theta^j (1-\theta)^{N-j} \geq \alpha \qquad (3.3.1)$$

if the N tests uncover F failures. For the important special case $F=0$, $1-\alpha$ is plotted in Figure 4.2-1 below.

Equation 3.3.1 completely solves the fundamental testing problem, because it predicts software behavior based on testing, even in the practical release-testing case that no failures are observed. The only question is whether or not the theory's assumptions are valid for software. What is most striking about equation 3.3.1 is that it does not depend on any characteristics of the program being tested. Intuitively, we would expect the confidence bound in a given failure intensity to be lower for more complex software.

To the best of my knowledge, no experiments have ever been published to support or disprove the conventional theory. It is hard to believe that convincing direct experiments will ever be conducted. A careful case study would take years to collect the field data needed to establish the actual failure intensity of a real program, and would be subject to the criticism that the program is somehow not representative. However, the following could be tried:

> Suppose that $F_0 \neq 0$ failures of a program are observed in N_0 runs. Then F_0/N_0 is an estimate of its failure intensity. The experiment may be repeated with additional runs of the same program, to see if a stable estimate $\hat{\Theta}$ of the failure intensity is obtained from many trials. Then α in equation 3.3.1 should estimate the fraction of experiments in which the observed failure intensity exceeds $\hat{\Theta}$, for that is the meaning of the upper confidence bound.

I do not believe the conventional theory would survive such experiments.

3.4. Thought Experiments with Partitions

The flaw in conventional reliability theory lies with the assumption that there is a sensible failure intensity defined through the input space. It is illuminating to consider subdividing the input domain, and applying the same conventional theory to its parts.

Suppose a partition of the input space creates k subdomains $S_1, S_2, \cdots, S_k$, and the probability of failure in subdomain S_i (the subdomain failure intensity) is constant at Θ_i. Imagine an operational profile D such that points selected according to D fall into subdomain S_i with probability p_i. Then the failure intensity Θ under D is

$$\Theta = \sum_{i=1}^{k} p_i \Theta_i . \qquad (3.4.1)$$

However, for a different profile D', different p_i' may well lead to a different $\Theta' = \sum_{i=1}^{k} p_i' \Theta_i$. For all profiles, the failure intensity cannot exceed $\Theta_{\max} = \max_{1 \le i \le k} \{\Theta_i\}$, because at worst a profile can emphasize the worst subdomain to the exclusion of all others. By partition testing without failure, a bound can be established on $\Theta_{\max}$, and hence on the overall failure intensity for all distributions. (This analysis is a much-simplified approximation to an accurate calculation of the upper confidence bound for the partition case [Tsoukalas+].) Thus in one sense partition testing multiplies the reliability-testing problem by the number of subdomains. Instead of having to bound Θ using N tests from an operational profile, we must bound $\Theta_{\max}$ using N tests from a uniform distribution over the worst subdomain; but, since we don't know which subdomain is worst, we must bound all k of the Θ_i, which requires kN tests. However, the payback is a profile-independent result. That is, a reliability estimate based on partition testing applies to all profiles.

The obvious flaw in the above argument is that the chosen partion is unconstrained. All that is required is that its subdomains each have a constant failure intensity. (This requirement is a generalization of the idea of "homogeneous" subdomains, ones in which all inputs either fail; or, all the inputs there succeed.) But are there partitions with such subdomains? It seems intuitively clear that functional testing and path testing do not have subdomains with constant failure rates. (Again, experiments are lacking, but here they should be relatively easy to conduct.) Of course, the failure intensity of a singleton subdomain is either 0 or 1 depending on whether its point fails or succeeds, but these ultimate subdomains correspond to exhaustive testing, and are no help in a statistical theory.

3.5. Where Does Failure Intensity Belong?

Results in Section 2, and those in Section 4 below, depend on the existence of a statistically meaningful failure-intensity parameter. So a first problem to be attacked in a dependability theory is to find a better space for this parameter than the program input domain. I have argued [Hamlet92] that the appropriate sample space is the computation space. A failure occurs when a design flaw comes in contact with an unexpected system state, and such "error" states are reasonable to imagine as uniformly distributed over all possible computational states. (The much misused "error" is in fact IEEE standard terminology for an incorrect internal state.) Rough corroboration for this view comes from measurement of software "defects/line-of-code," which is routinely taken as a quality measure, and which does not vary widely over a wide range of programs.

4. Dependability Theory

"Dependability" must be defined as a probability, in a way similar to the definition of reliability. If a state-space-based failure parameter such as suggested in Section 3.5 can be defined, it would do. However, other possibilities exist.

4.1. Reliability-based Dependability

Attempts to use the Nelson TRW (input-domain) reliability to define dependability must find a way to handle the different reliability values that result from assuming different operational profiles, since dependability intuitively should not change with different users. It is the essence of dependability that the operating conditions cannot be controlled. Two ideas must be rejected:

(**U**) Define dependability as the Nelson reliability, but using a uniform distribution for the profile. This suggestion founders because some users with profiles that emphasize the failure regions of a program will experience *lower* reliability than the defined dependability. This is intuitively unacceptable.

(**W**) Define dependability as the Nelson reliability, but in the worst (functional) subdomain of each user's profile. This suggestion solves the difficulty with definition **U**, but reintroduces a dependency on a particular profile. In light of the dubious existence of constant failure intensities in subdomains (Section 3.4), the idea may not be well defined.

Other suggestions use reliability only incidentally, introducing essentially new ideas.

4.2. Testing for Probable Correctness

Dijkstra's famous aphorism that testing can establish only the *incorrectness* of software has never been very palatable to practical software developers, who believe in their hearts that extensive tests prove *something* about software quality. "Probable correctness" is a name for that illusive "something." However, the TRW reliability theory (Section 3.3) provides only half of what is needed. Statistical testing supports statements like "in 95% of usage scenarios the software should fail less than 1% of the time." These statements clearly involve software quality, but it is not very plausible to equate the upper confidence bound and the chance of success, and turn "99.9% confidence in failure intensity less than .1%" into "probable correctness of 99.9%" [Hamlet87].

Jeff Voas has proposed [Voas & Miller] that reliability be combined with *testability* analysis to do better. Testability is a *lower* bound probability of failure if software contains faults, based on a model of the process by which faults become failures. In Voas's model, testability is estimated by executing a program and measuring the frequency with which each possible fault location is executed, the likelihood that a fault would corrupt the internal state there, and the likelihood that a corrupt state would not be later corrected. The combination of these factors identifies locations of low testability, places in the program where a fault could easily hide from testing. The program's testability is taken to be the minimum value over all its locations. A testability near 1 thus indicates a program that "wears its faults on its sleeve:" if it can fail, it is very likely to fail under test.

If testability estimates are made using an operational profile as the source of executions, and conventional random testing uses the same profile, a "squeeze play" is possible. Successful random testing demonstrates that failure is unlikely, but testability analysis shows that *if there are any faults* failures would be seen. The only conclusion is that faults are unlikely to exist. The squeeze play can be made quantitative [Hamlet & Voas] as shown in Figure 4.2-1. In Figure 4.2-1, the falling curve is the confidence

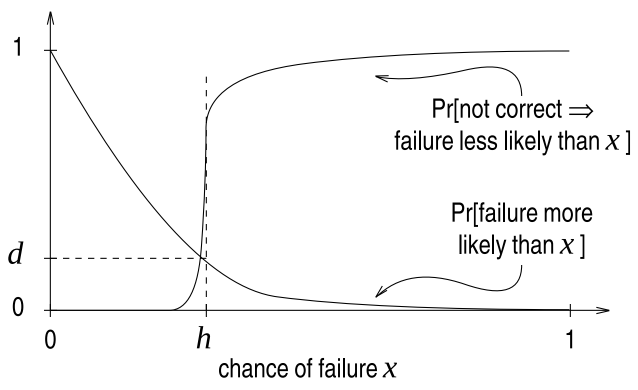

Figure 4.2-1. 'Squeeze play'

from reliability testing from Equation 3.3-1; the step function comes from observing a testability h. Together the curves make it unlikely that the chance of failure is large (testing), or that it is small (testability). The only other possibility is that the software is correct, for which $1-d$ is approximately the upper confidence bound, where d is the value of the falling curve at h in the figure. Confidence that the software is correct can be made close to 1 by forcing h to the right in Figure 4.2-1.

Manuel Blum has proposed [Blum & Kannan] a quite different idea as an adjunct to reliability. He argues that many users of software are interested in a particular execution of a particular program only — they want assurance that a single result can be trusted. Blum has found a way to sometimes exploit the low failure intensity of a "quality" program to gain this assurance. (Conventional reliability would presumably be used to estimate the program quality, but Blum has merely postulated that failure is unlikely.) Roughly, his idea is that a program can check its output by performing redundant computations. Even if these make use of the same algorithm, if the program is "close to correct," it is very unlikely that a sequence of checks could agree yet all be wrong.

4.3. Defining Dependability

Either Voas's or Blum's idea could serve as a definition for dependability, since both capture a probabilistic confidence in the correctness of a program, a confidence based on sampling.

Dependability might be defined as the confidence in correctness given by Voas's squeeze play. Even if conventional reliability is used for the testing part of the squeeze play, the dependability so defined depends in an intuitively correct way on program size and complexity, because even Voas's simple model of testability introduces these factors. His model also introduces an implicit dependence on the size of both input and internal state spaces, but this part of the model has not yet been explored.

Dependability might also be defined for a Blum self-checking program as the complement of the probability that checks agree, but their common value is wrong. This dependability may be different for different inputs, and must be taken to be zero when the checks do not agree. Thus a definition based on Blum's idea must allow software to announce its own untrustworthiness (for some inputs).

The promise of both Voas's and Blum's ideas is that they extend reliability to dependability and at the same time substantially *reduce* the testing cost. Instead of requiring "ultra-reliability" (roughly below 10^{-8} failures/run) that cannot be estimated in practice [Butler & Finelli], their ideas add a modest cost to reliability estimates of about 10^{-4} failures/run, estimates that can be made today. Blum's idea accrues the extra cost at runtime, for each result computed; Voas's idea is more like conventional testing in that it samples the whole input space, before release.

5. Conclusions

It has been argued that a foundational "dependability" theory of software testing must be statistical in nature. The heart of such a theory is program reliability derived from a constant failure intensity, but defining failure intensity over the input space is inappropriate. A more plausible reliability theory is needed, and the nearly constant defects/line-of-code data suggests that the failure intensity should be defined on the program state space. Dependability can then be defined using new ideas such a Voas's squeeze play, or Blum's pointwise correctness probability.

Although we do not yet have a plausible dependability theory, it is possible to imagine what the theory can establish. My vision is something like the following:

> Dependable software will be developed using more front-end loaded methods than are now common. Testing to find failures will play a minor role in the development process. The resulting software will be tested for dependability by a combination of conventional random testing, and new methods that probe the state space. Random testing will establish reliability better than about 10^{-4} failures/run. The new methods will provide (quantitative!) high confidence in correctness, so that in use the software will fail less often than once in 10^7 to 10^9 runs.

The cost of testing in my vision may not be less than at present, but neigher will such testing be intractable. The significance of passing the tests, however, will be far greater than at present. Instead of using release testing to find failures, as we do now, dependability tests that succeed will quantitatively predict a low probability that software will fail in use. Software that is badly developed will not pass these tests, and responsible developers will not release it.

References

[Blum & Kannan]

M. Blum and S. Kannan, Designing programs that check their work, *Proc. 21st ACM Symposium on Theory of Computing*, 1989, 86-96.

[Butler & Finelli]

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

[Cobb & Mills]

R. H. Cobb and H. D. Mills, Engineering software under statistical quality control, *IEEE Software*, November, 1990, 44-54.

[DeMillo78]

R. DeMillo, R. Lipton, and F. Sayward, Hints on test data selection: help for the practicing programmer, *Computer* 11 (April, 1978), 34-43.

[Duran & Wiorkowski]

J. W. Duran and J. J. Wiorkowski, Quantifying software validity by sampling, *IEEE Trans. Reliability* R-29 (1980), 141-144.

[Duran & Ntafos]

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

[Foster]

K. A. Foster, Error sensitive test cases analysis (ESTCA), *IEEE Trans. Software Eng.* SE-6 (May, 1980), 258-264.

[Frankl & Weyuker]

P. G. Frankl and E. J. Weyuker, A formal analysis of the fault-detecting ability of testing methods, *IEEE Trans. Software Eng.* SE-19 (March, 1993), 202-213.

[Hamlet77]

R. Hamlet, Testing programs with the aid of a compiler, *IEEE Trans. on Software Eng.* SE-3 (July, 1977), 279-290.

[Hamlet87]

R. Hamlet, Probable correctness theory, *Inf. Proc. Let.* 25 (April, 1987), 17-25.

[Hamlet89]

R. Hamlet, Theoretical comparison of testing methods, *Proc. Symposium of Software Testing, Analysis, and Verification (TAV3),* Key West, December, 1989, 28-37.

[Hamlet92]

D. Hamlet, Are we testing for true reliability?, *IEEE Software,* July, 1992, 21-27.

[Hamlet94]

D. Hamlet, Random testing, in *Encyclopedia of Software Engineering,* J. Marciniak, ed., Wiley, 1994, 970-978.

[Hamlet & Taylor]

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

[Hamlet & Voas]

D. Hamlet and J. Voas, Faults on its sleeve: amplifying software reliability testing, ISSTA ‘93, Boston, June, 1993, 89-98.

[Jeng & Weyuker]

B. Jeng and E. Weyuker, Analyzing partition testing strategies, *IEEE Trans. Software Eng.* SE-17 (July, 1991), 703-711.

[Kolkhorst]

B. G. Kolkhorst, personal communication, January, 1993.

[Lauterbach & Randall]

L. Lauterbach and W. Randall, Experimental evaluation of six test techniques, *Proc. COMPASS ‘89,* June, 1989, Gaithersburg, MD, 36-41.

[Morell & Hamlet]

L. J. Morell and R. G. Hamlet, Error propagation and elimination in computer programs, University of Maryland Computer Science Technical Report TR-1065, July, 1981.

[Musa+]

J. D. Musa and A. Iannino and K. Okumoto, *Software Reliability: Measurement, Prediction, Application,* McGraw-Hill, 1987.

[Myers]

G. Myers, *The Art of Software Testing,* Wiley, New York, 1979.

[Rapps & Weyuker]

S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Software Eng.* SE-11 (April 1985), 367-375.

[Thayer+]

R. Thayer, M. Lipow, and E. Nelson, *Software Reliability,* North-Holland, 1978.

[Tsoukalas+]

M. Z. Tsoukalas, J. W. Duran, and S. C. Ntafos, On some reliability estimation problems in random and partition testing, *Proc.*

Second International Symposium on Software Reliability Engineering, Austin, TX, May, 1991.

[Voas & Miller]

J. M. Voas and K. W. Miller, Improving the software development process using testability research, *Proc. Third International Symposium on Software Reliability Engineering*, Research Triangle Park, NC, October, 1992, 114-121.

[Weyuker+]

E. J. Weyuker, S. N. Weiss, and D. Hamlet, Comparison of program testing strategies, *Proc. Symposium of Software Testing, Analysis, and Verification (TAV4)*, Victoria, October, 1991, 1-10.