Faults on Its Sleeve: Amplifying Software Reliability Testing

Dick Hamlet[†]
Center for Software Quality Research
Portland State University

Jeff Voas
Reliable Software Technologies Corporation

Abstract

Most of the effort that goes into improving the quality of software paradoxically does not lead to quantitative, measurable quality. Software developers and quality-assurance organizations spend a great deal of effort preventing, detecting, and removing "defects"—parts of software responsible for operational failure. But software quality can be measured only by statistical parameters like hazard rate and mean time to failure, measures whose connection with defects and with the development process is little understood.

At the same time, direct reliability assessment by random testing of software is impractical. The levels we would like to achieve, on the order of 10^6 - 10^8 executions without failure, cannot be established in a reasonable time. Some limitations of reliability testing can be overcome, but the "ultrareliable" region above 10^8 failure-free executions is likely to remain forever untestable.

We propose a new way of looking at the software reliability problem. Defect-based efforts should *amplify* the significance of reliability testing. That is, developers should demonstrate that the actual reliability is better than the measurement. We give an example of a simple reliability-amplification technique, and suggest applications to systematic testing and formal development methods.

Keywords: software reliability, testability, fault, failure

† Work supported by NSF grant CCR 9110111

1. Reliability Amplification

Software quality is the focus of renewed attention, in a world where software is becoming all-pervasive and its quality is too often left to chance. It is a common belief that the software industry is at risk for a disaster or some kind, a disaster in which the blame will be clearly laid to software, and all will be tarred with the brush used on the unlucky developer. To guard against such a disaster, many software organizations are beginning to devote considerable attention to attaining software quality. The most popular approach is one of "process." The Software Engineering Institute (and in Europe, the International Standards Organization) have attempted to describe informally the *way* in which quality software should be developed, and to indicate characteristics of an organization that can develop it.

However, attention to process and developer organization are only the most recent fads in seeking software quality. Software testing is a much older method with the same intent, as are inspections of software products. Formal methods in great variety are being applied throughout the development cycle. All of these ideas, from process definition and control to systematic testing, have one failing in common: there is no established relationship between the method and quantitative assessment of the quality that method is supposed to engender. This is not to say that effort aimed at quality is wasted, or that people trying to attain it cannot do so. It is to say that common sense and the best intentions are not an adequate basis for trusting software, nor an adequate hedge against disaster.

No one contests that attention to finding and enforcing good engineering practice for software development is well advised. Any process in which people are continually made aware of a goal, however imprecisely stated and imperfectly sought, is likely to increase the chance that the goal will be attained. As a concrete example, inspecting a software design for problems is likely to find some, and to aid in their early removal. But quality is not measured by effort, however well intentioned. Discovery of many defects can as well be caused by an oversupply at the outset as by finding most of them. All measures of the efficacy of defect prevention, detection, and elimination methods yield the same results when rotten software is made only a little less rotten as when better software is perfected. In their haste to sell the sound ideas of software-development process control and defect-based methods, people choose to forget that these methods are necessary but not sufficient. It may be impossible to develop quality software without them; it is certainly possible to develop unreliable software with them.

Software is an object whose behavior defines its quality. Quality behavior is not the organization that developed the software, nor the vigor with which defects were attacked. Quality behavior is directly measurable only in the field: quality software does not fail too often, and never fails in catastrophic ways. It may seem harsh to discount the good intentions of developers, the clever tricks they used at great expense to do the best job possible. But engineering has a harsh judge in the operating environment, and nothing excuses failure there. Insofar as we cannot predict quantitatively the effect of our methods on the measurable quality parameters of software, those methods (or at least our understandings of them) are wanting.

This paper suggests a new viewpoint for software-engineering quality research. Development ideas and practices should be subjected to a new kind of reliability analysis. A method that can be quantitatively shown to lead to enhanced reliability is to be preferred to one whose quantitative connection with reliability is unknown. We suggest a further refinement: a method is best if it *amplifies* reliability measurements. That is, reliability analysis of a development method should show that when the method has been used, and reliability measured, the actual reliability is necessarily better than the measurement. Furthermore, the analysis should be quantitative, so that the reliability improvement can be calculated.

Good engineering practice in software development is obviously a necessity on the most intuitive grounds: we cannot expect to get away with haphazard construction of the most complex objects in human history. But quantitative reliability amplification is also necessary because direct measurement of adequate software quality is impractical, and will probably always remain so. The testing effort required to establish a certain mean time to failure (MTTF) with 90% confidence is at least twice that MTTF. There are about 10^7 sec in a work-year. Even ignoring all problems of test generation, test oracles, and test-administration overhead, and making use of overspeed execution and parallel hardware for test, it is difficult to foresee more than about 10 tests/sec of complex software. Current practice is perhaps three orders of magnitude less. Thus testing for months could measure a MTTF of 10^3 runs in current practice, or 10^6 runs at best. But the requirement for mass-distribution software is a MTTF of about 10^7 (hundreds of runs over hundreds of thousands of copies). Thus testing must fall short by a factor of about 20 (or a factor of 20,000 in practice today!). In the "ultrareliable" region—MTTF of about 10^9—it will always be impractical to test programs. Thus a way must be found to quantitatively predict that reliability is better than what can be directly measured.

We thus imagine that each software system is subjected to an unsatisfactory reliability measurement—it will prove impossible to conduct enough testing to predict the required MTTF. It is the task of good engineering practice to amplify the measurement. For example, suppose that a MTTF of 10^5 runs is required, but that testing resources allow at best a prediction that the MTTF is greater than 10^3 runs. The developer’s task is to demonstrate that this prediction is pessimistic by at least two orders of magnitude, because the development methods used establish the needed additional quality.

The remainder of this paper is organized as follows: Section 2 describes reliability measurement, and Section 3 describes testability analysis. Section 4 presents a simple example of reliability amplification, using testability analysis. Section 5 is a critique of the models underlying conventional reliability and testability. Section 6 considers systematic methods usually used for unit testing, and the possibility of using them for reliability amplification. Section 7 suggests how existing formal development methods might be justified by reliability-amplification arguments. In Section 8 the related idea of self-testing programs is examined.

2. Conventional Reliability Measurement

Software reliability measurements through random testing are difficult to make in practice. The most apparent practical problem is the absence of an effective oracle for testing—a means of automatically judging software behavior as meeting or not meeting specifications. But from a theoretical viewpoint, the worst difficulty is the lack of a specified operational profile. Without an accurate profile, there can be no validity to the test—its points are not representative of subsequent usage. Coupled with an unspecified profile may be doubts about what it means to select inputs "at random," that is, without correlation. All of these difficulties are glossed over in the usual description of conventional reliability testing, as follows:

> To estimate software reliability, choose test points at random according to an operational profile. The MTTF can be directly measured, using these test points and the oracle to detect failures.

If no failures occur, substantial further assumptions are needed, namely that the software has a sensible probability of failure, related to the size of the test that has not failed. This situation can be described using hypotheses about (mis)interpreting the test; or, perhaps more simply, it can be postulated that there is a meaningful long-term failure rate Θ and that N random tests have established an upper confidence bound $1-\alpha$ that Θ is below some level θ. These quantities are related by [Thayer et al.]:

$$(\blacklozenge\blacklozenge) \qquad 1-(1-\theta)^N \le \alpha.$$

Thus it is always possible to trade confidence $1-\alpha$ for maximum failure-rate θ in equation ($\blacklozenge\blacklozenge$): A given test size N can always be viewed as providing low confidence that Θ is below a small θ, or higher confidence that Θ is below a larger (worse) θ. The MTTF, assuming Θ is constant over program runs, is $1/\Theta$.

An important difference between the case of a measured MTTF and that of a calculated θ when no failures occur, is that the former permits experimental confirmation of its predictions (by repeating the measurements, for example). When failures do not occur, and a large MTTF is predicted as a result, no such experiment is possible.

In principle, reliability testing can be applied to any software system; in practice, the real-time control systems for which MTTF is most needed have more difficulty with an oracle and an operational profile than do batch systems [HamletB].

3. Testability Analysis

In the context of reliability, notions of "testability" based on peculiar properties of testing methods must be rejected. For example, it makes no sense to call a program "testable" when branch coverage (say) is easy to obtain, because the relationship between branch coverage and program failure is unknown. Our definition insists on a correlation between test outcome and program quality:

> The *testability* of a program P is the probability that if P could fail, P will fail under test.

Making testability conditional on P's failure is important, to leave open the possibility that a program has high testability, yet never fails. The definition also implies the use of a particular testing method; if the method changes, the testability could change. And finally there is an implication that testability is concerned with the quality of being defect-free rather than with mere reliability, since the definition speaks of possible failures rather than likely failures.

Any assessment of testability must be based on a model of the fault/failure relationship. The following simple model of this relationship has been proposed [Voas]:

> Each textual location in a program is considered as a possible location of a fault. At a given location, a possible fault could result in a failure if and only if:
>
> An input results in execution reaching the location. (The *execution* condition.)
>
> The data state that results from execution is in error. (The *infection* condition.)
>
> Infection results in an observable incorrect output. (The *propagation* condition.)

This model is very simplistic, because it imagines that faults occur at single locations; however, it can be used to define a practical approximation to testability.

If we confine the possibility of a program's failure to this simple fault model, and imagine three independent probabilities of execution, infection, and propagation (which we call the *conversion* probabilities), then the probability that a fault will turn into a failure is the product of the three conversion probabilities. In terms of this model, the definition of testability becomes:

> The *testability* of a program P is the probability that if P contains fault(s), P will fail under test.

A lower bound on testability of a program is obtained as the least conversion-probability product over all locations in the program. Since it is not known which (if any) locations contain faults, the probability of seeing a failure is at least as large as the probability that one could result from the least-apparent fault. Intuitively, a program with high testability "wears its faults on its sleeve;" faults are likely to be revealed as failures. A program with low testability hides its faults in the sense that they are unlikely to come to light under test. Low testability can result from a low probability of execution, infection, or propagation, or from a combination.

It is possible for a program to have a testability near 1, yet in fact contain no faults. Such a situation arises when any *possible* faults would almost certainly appear as failures (that is, all three conversion probabilities are near 1 at all locations), but in fact no location contains a fault. Without a model of fault-to-failure conversion, we cannot describe the desirable situation of a correct program with high testability.

The conversion probabilities of the Voas model can be estimated, and the analysis can be automated [Miller et al.], driven by an input profile from which points are drawn. Conventional program instrumentation at each location can determine the frequency with which the location is executed by points drawn from an operational profile, thus estimating the execution probability. The infection probability can be estimated by making mutations of the source code at each location, and monitoring the state (as in weak mutation testing) to detect changes (if any). The base states are those that result from data drawn from the operational profile. Finally, the propagation probability can be estimated by introducing random perturbations into these base states, and observing whether or not outputs are affected. Estimating the conversion probabilities allows an estimate of the testability lower bound to be calculated as the least value of their product over all program locations.

Testability estimation uses many of the ideas of testing, but is fundamentally different. The program output is not compared to a specification, so no oracle is required. The analysis yields not an estimate of the failure probability (as a test would), but rather an estimate of how likely it is for potential faults to turn into failures. Thus in the situation that the program is very unlikely to fail, and it does not fail under test, testing can only estimate a failure probability of 0; on the other hand, testability can be high or low.

Glenford Myers is credited with the insight that the purpose of testing is to expose failures, not to find nothing wrong [Myers]. The idea of testability gives Myers' argument a new twist. Imagine two programs, one P_L with low testability, and the other P_H with high testability. Which program should a developer seek to create? A developer's first response might be that P_L is better. It will be difficult to cause P_L to fail, so it will appear to be of high quality, and (say) pass its acceptance tests. P_H is worse in this view because it is more likely to fail under test. Why, this developer might ask, should programs be designed to fail? Such a developer has missed Myers' point. At the completion of whatever testing can be afforded, neither program will have failed in its final, release version. For P_L this might be because whatever faults it contains are hidden; for P_H it is more likely that there are less faults to find. Thus we can add testability to Myers' advice as follows:

> Design for high testability, then test to uncover failures. When no more are found, there are less likely to be hidden ones that have been missed.

"Design for testability" is a recognized goal in hardware development, whose essential idea is that internal states of a chip should be made visible on the pins for testing. The analogous idea for software has been explored [Hoffman], but "design for testability" in the sense of this section is a rather different idea. The software designer who wants testability near 1.0 must seek to avoid small conversion probabilities. That is, programs should not contain code that is seldom executed in normal use. They should not use expressions that collapse state (for example with **`mod`** functions, or reducing a complex set of values to a Boolean condition), particularly when producing output. A simple example of both mistakes is a seldom-invoked Boolean function with several real inputs. Such a function has a low execution probability because of its infrequent usage; it has low infection/propagation probabilities because the many state possibilities its inputs represent are collapsed into {true, false} in the result.

Developers cannot always avoid low testability; for example, a problem may require Boolean decision functions to handle low-frequency cases. Then the design must encapsulate the low-testability code into routines that can be verified by means other than testing. This advice makes additional development work, but it also saves testing of components (and with poor design, whole systems) that are intrinsically untestable.

4. Reliability Amplification using Testability

Reliability testing is flawed precisely because the significance of a successful test can only be predicted using a dubious model. When tests drawn from the operational profile do not fail, the only estimate available for the failure rate is 0. As described in Section 2, by assuming a conventional failure model, it is possible to predict confidence in an upper bound on the failure rate, e.g., "99% confidence that the failure rate is below 10^{-4}/run" (which would require 46,000 test runs from equation (♦♦)). The number of test runs needed precludes establishing high confidence in low enough bounds [Butler & Finelli]. Figure 1 shows the situation after a successful random test of N points has established confidence C that failure rate θ exceeds the actual failure rate Θ. A particular value

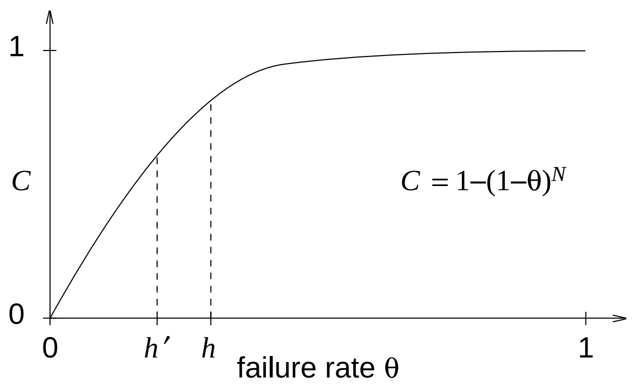

Figure 1. Reliability testing

of θ such as h in Figure 1 establishes confidence $1-(1-h)^N$ that the actual failure rate Θ is below h. For the same number of test points, a better bound such as h' can be selected, but with lower confidence.

The developers may believe that in fact the software is more reliable than the smallest upper bound for which testing with adequate confidence is practical. Testability provides a way to quantify this belief, and to amplify the reliability measurement. Suppose that a lower bound on the testability of a program has been estimated to be h. That is, if the program has faults, the probability that it will fail for a certain operational profile is greater than h. Figure 2 shows the probability that failure rate θ exceeds actual rate Θ.

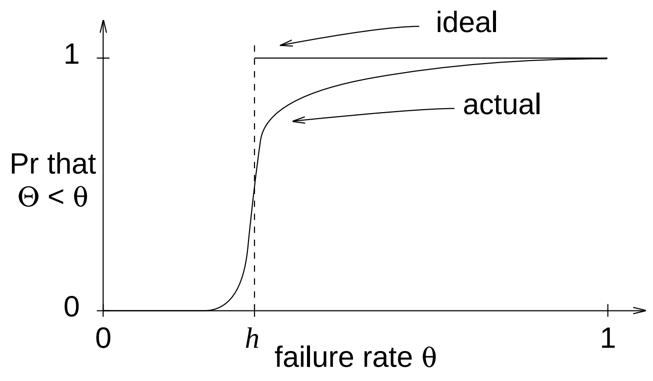

Figure 2. Testability analysis

In reality, there will be some uncertainty about the testability, resulting in the "actual" curve; ideally, we can take the transition to be a unit step function at h.

If reliability testing with the same profile establishes $\theta = h$ as an upper bound on the actual failure rate Θ, we have a "squeeze play" [Voas & Miller]: from testing there is some confidence that Θ is below h. But from testability we believe that if there are faults, a failure rate above h will be observed. Hence we have some confidence that there are no faults, that is, that the software is correct. Figure 3 shows the testing curve $1-C = (1-\theta)^N$ from Figure 1, and the testability curve as an idealized step function.

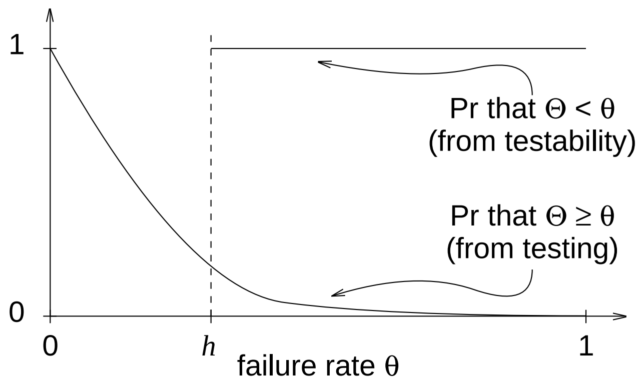

Figure 3. Squeeze play

From testing, the probability that Θ lies above h is $(1-h)^N$. That is, the probability that Θ lies in the interval $[h,1]$ is $(1-h)^N$. From testability, assuming a unit step at h, the probability that Θ lies in the interval $(0,h)$ is 0. Thus the probability that Θ lies in (0,1) is $(1-h)^N$. This is the probability that the software can fail, hence the probability of correctness is $1-(1-h)^N$. (For a more realistic testability curve like the "actual" one in Figure 2, this estimate would be reduced by a non-zero probability from the interval $(0,h)$.)

In practice, the testability estimate h is a property of the software, while the size of the test N can be increased with more testing effort. Table 1 displays some numerical values of these parameters. The squeeze play always works, and works very well for a practical range of testability estimates and test sizes. When it works less well it is because a high-confidence upper bound on Θ lies far above the testability estimate h.

5. Critique of the Amplification Example

The reliability amplification presented in Section 4 is a practical scheme for software validation through testing and testability measurements. The two complementary techniques are used to achieve what neither could alone: a confidence bound for the correctness of a program.

testability estimate h	number of test points N	probability that software is *not* correct $(1-h)^N$
.0000001	1000	1.0
.0000001	10M	.30
.00001	1000	.99
.00001	100K	.37
.00001	1M	.000045
.001	1000	.37
.001	10K	.000045
.001	20K	.0000000020
.1	100	.000027
.1	200	.00000000071

Table 1. Practicality of the squeeze play

However, we are the first to admit that many details of the models in Sections 2 and 3 are a poor abstraction of reality. Conventional reliability testing gives an overly optimistic bound on the failure rate, because it fails to take into account the effect of an incorrect operational profile, and possible correlation between test points that "do the same thing" relative to failures [HamletA]. Testability analysis is obviously in its infancy, and its ways of estimating the conversion probabilities are crude approximations, justified only because they make automatic measurements possible. Testability also relies on an operational profile, although the distortion that results from an incorrect profile is less clearly optimistic than for reliability testing. Simple mutations do not model faults very well, casting doubt on the infection part of the analysis. The state perturbations used have no necessary relation to perturbations that can occur in actual executions. The three conversion probabilities are not independent as assumed. The most glaring defect of the model is the assumption that each fault is tied to a single program location. Real faults that result in low-frequency failures are more likely to be distributed, involving interaction between program locations that are unobjectionable in isolation.

These flaws in the fault-to-failure conversion model introduce an uncertainty into the testability estimates that is impossible to quantify. There should be a confidence bound attached to the testability estimate, but the statistical effect of the number of inputs used for analysis will certainly be swamped by outright mistakes in the model. On the other hand, testability analysis obtains probability estimates, rather than confidence in unmeasurable bounds as in reliability testing, so some confidence in the measurements can be obtained in practice by repeating them and observing the stability. Sometimes the fault/failure model does surprisingly well. For example, its propagation component can handle omitted assignment statements. At the location of a missing assignment to variable (say) **X**, perturbing **X** in the state simulates the missing statement. If the infection fails to propagate, the testability of the location is low. An experiment to measure the ability of propagation to predict missing assignments gave a correlation of .96 between predicted and measured failure rates [Voas].

Whatever the flaws in the models of Sections 2 and 3, we emphasize that the idea of reliability amplification does not depend on those details. A more plausible reliability theory for software is needed, and a better understanding of the fault-failure relationship. Better foundational theory will make reliability amplification more accurate. But even the simple-minded example of Section 4 represents an improvement over testing-only verification schemes. An underlying fault/failure model is central to a better understanding of reliability amplification. We believe that the heart of the theoretical problem is finding a proper home for the failure rate, and we believe that it should be assigned to the program computation and its data-state values, instead of to points of the input domain.

6. Reliability Amplification Based on Systematic Testing

The bulk of software testing is "systematic," that is, directed by a testing method of some kind, and not "random." The descriptive adjectives are biased against "random," which has the peculiar hacker's meaning of "not well organized." But for reliability predictions, "systematic" is *not* what tests should be—the statistical predictive power is lost when test inputs follow a pattern. Thus most of practical testing is not testing for reliability, and when tests succeed (as they eventually must, and the software is released) there is little statistical implication that this success will be replicated in use [Hamlet & Taylor].

There is, however, a widespread belief that systematic testing is essential to software quality, and that true reliability cannot be attained without it. In particular, there is spirited resistance to reducing structural and functional unit testing—the systematic methods in widest use—in favor of random reliability testing at the system level. Practical developers do not trust the statistical methods. The basis of their belief is that systematic methods find defects causing system failure with a frequency too low to detect with random tests, but too high to risk release. This is precisely the description of reliability amplification: systematic testing is thought to probe the failure-rate region that random tests cannot reach, and thus to enhance whatever reliability can be established. However, this argument has never been made precise, so systematic testing continues to receive only anecdotal support.

Systematic methods are related to the testability analysis described in Section 3. Insofar as testing is required to satisfy a systematic criterion, the tester must pay more attention to low-testability locations, because there the criterion will be hard to satisfy. For example, at a location where the data state is insensitive to infection, weak mutation coverage will be difficult to attain. However, systematic methods are not used to find insensitive locations. Instead, the tester works hard to satisfy the test method everywhere. Can we then quantify the improvement in reliability, if any?

Systematic methods can be assessed statistically. For partition methods, measures such as the probability that at least one failure will be exposed, can be derived by a combination of analytic and simulation methods, in terms of the failure rates in the subdomains for the partition [Duran & Ntafos, Hamlet & Taylor, Weyuker & Jeng]. When an actual partition test is conducted, and no failures are observed, it establishes confidence bounds on these failure rates, just as an overall random test does for the whole domain. However, the failure rate and its confidence bound observed through the overall operational profile are not simply related to failure rates in the subdomains, because the relationship must be expressed in terms of the probabilities that points from the operational profile fall into each subdomain.

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 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 rate Θ under D should be $\Theta = \sum_{i=1}^{k} p_i \theta_i$. 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 rate 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 rate for all distributions. (This analysis is a much-simplified approximation to an accurate calculation of the upper confidence bound for the partition case [Tsoukalas et al.].)

In one sense, then, partition testing merely multiplies the reliability-testing problem. 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 θ_i, which requires kN tests. However, the payback is a profile-independent result. That is, partition testing can be the basis of a reliability estimate that applies to all profiles.

The testability analysis of Section 3 can be applied to partition testing, but now in a profile-independent manner. The estimate of execution frequency comes from uniformly selecting points in a partition subdomain; other parts of the analysis are unchanged. (In some cases, the partition definition forces an execution frequency of 1.0, for example, in a functional partition when a code location is always used for a unique function. However, it is still necessary to actually sample the partition, to obtain a set of states needed for infection and propagation analysis.)

When testability analysis has been carried out separately in each subdomain, a certain confidence has been established in the software's correctness. The partition may also be used to focus attention on subdomains with low-testability locations. Either the code may be redesigned to improve its testability, or additional reliability testing in the subdomain may be conducted to achieve a given confidence in correctness. If the partition is functional, the amplified reliability can be used as a "quality profile" of the software: the confidence in correctness for each subdomain indicates how much each function can be trusted, based on testing and testability measurements.

We believe that the discussion above captures an intuition that partition testing amplifies reliability measurements. Partition testing shifts emphasis from the operational profile to partition subdomains. If a subdomain has a high failure rate, but is neglected by the operational profile, then potential failures there may be invisible to an overall reliability measurement. (That is, testability measured through the overall operational profile will be low for locations used by the subdomain; the low testability masks the high failure rate.) Sampling the subdomain directly brings the failure rate into the measurable region. Hence when no failures are observed in a partition test, it provides evidence about a failure region unobservable through the operational profile.

Partition testing for reliability amplification is a new idea placing new constraints on the partition subdomains, however. An arbitrary division of the input domain is inappropriate, and it is instructive to examine the reasons. The primary assumption of the theory is that θ_i, the subdomain failure rates, are constant. A related assumption is that the operational distribution D actually scatters points uniformly within each subdomain. These assumptions justify taking the overall failure rate Θ as the weighted sum $\sum_{i=1}^{k} p_i \theta_i$. In practice, the assumptions are false. It can easily happen that D concentrates on some part of a subdomain S_i; if θ_i is not constant as assumed, but higher where D concentrates, the contribution of S_i to Θ will be greater than $p_i \theta_i$. In an extreme case, suppose that a program fails at test point F. Partitions that place F in a shrinking subdomain, and an operational profile that peaks on that subdomain, will lead to $\Theta \rightarrow 1.0$ in the limit. Any other partition or profile will underestimate Θ and yield a false confidence in the program's correctness. (This extreme case contains the essence of the argument that *no* statistical theory is appropriate for software design flaws.)

Thus to use partition testing for reliability amplification, it is essential that each subdomain have a constant failure rate. This is a generalization of the well known criterion that subdomains should be "homogeneous" in the sense that all points should fail or all should succeed. Constant θ_i means that the probability of failure for each point in S_i should be the same. The usual assumption of a constant overall failure rate Θ for an entire software input domain is not at all plausible; perhaps the theoretical significance of partition testing is to be able to arbitrarily subdivide the domain, seeking S_i for which constant θ_i *can* be justified. However, just how this subdivision should be accomplished is a difficult question that requires further investigation. It does not seem very likely that (say) path-equivalence subdomains have constant θ_i.

7. Reliability Amplification Through Formal Methods and Software Process

The intuitive feeling that systematic testing can increase reliability, analyzed in Section 6, also underlies the current preoccupation with "formal methods" and "software process." Following successful practice in other engineering disciplines, we would like to think that formalizing the procedures of software development will lead to improved quality. The evidence for quality improvement by process is so far anecdotal, but includes a number of case studies whose participants are enthusiastic about the methods used.

It is probably impossible to make a rigorous, quantitative analysis of even the most formal development methods, since human beings have a wide latitude of choice in each application. However, the framework of reliability amplification does suggest the kind of supporting arguments that should be advanced by proponents of development methods: a method plausibly improves software quality if it can be shown that its use, in conjunction with reliability testing, quantitatively argues for a lower failure rate than that which is actually measured.

The danger in methods directed at "zero defects" is that instead of amplifying reliability, they might only reduce testability. That is, a development method might not have much effect on the fault insertion rate, but might make faults more difficult to uncover by testing. For uncritical applications, such a development method would be advantageous, since the low-testability faults will not come to light. But for applications requiring ultrareliabilty, the method would be counterindicated—it creates a false confidence based only on inadequate reliability testing. Since methods are necessarily evaluated well below the ultrareliability region, their proponents must address the testability question.

Intuitively, a method that knowingly gains reliability at the expense of testability is irresponsible engineering. An analogy in aeronautical engineering would be designing an airframe to perform well in simulation and wind-tunnel tests, when it is known to be of doubtful quality in practice because those results do not scale well. An engineer should worry a good deal about methods that can mislead in this way, and the only tools available for protection are theoretical: plausible arguments must be given to show that reliability is amplified, not "improved" because its measurement is misleadingly.

To give a revealing example of a reliability-amplification argument, analysis of cleanroom development [Cobb & Mills] is the obvious choice. Briefly, cleanroom design and implementation uses top-down design, but with continual attention to the functional state transformations being computed by the developing code. Particular reliance on the formal analysis is forced by a prohibition on testing during design and implementation. Conventional system reliability testing is used as the final step in development.

In case studies the only failures found are those that would be detected by almost any input. This suggests that in fact the reliability is better than that measured—perhaps the software is defect-free after obvious bugs are removed. However, the argument advanced by cleanroom proponents rests on the intuitive appeal of their method, and on the weak empirical evidence of case studies. Reliability amplification suggests a framework for a stronger supporting argument, and also suggests modifications to improve the cleanroom method.

The interesting cases for analysis lie in the failure-rate region below that measured by the system reliability test. Failures in this region will not be detected; what assurance can we give that cleanroom development makes them less likely? In cleanroom terms, a failure arises from implementation of an incorrect function, which is the composition of functional implementations of the top-down development process. Because the formal development process considers each function in isolation, coincident failures—those in which an erroneous state value output by one function happens to be erroneously handled by the next to which it is input—are unlikely. That is, the method addresses both infection probability and propagation probability by separating the analysis into compositional steps.

This analysis suggests two improvements in the cleanroom method:

(1) Propagation probability would be more directly addressed if attention were paid not only to the functions computed, but to values outside their domains. That is, the informal examination would not only seek to show that each function is correct on its domain, but that if it receives erroneous input states these will be detected.

(2) Cleanroom does not address execution probability at all. It could do so by requiring structural partition tests as indicated in Section 6, in addition to operational-profile-based system tests, as exit criteria for development. (It is argued [Cobb & Mills] that structural tests are inferior in finding failures that users will first encounter. But if the software is reliable enough that there are no early-encounter failures, structural tests are better at finding the problems that can only appear under extended use; those may well be the first that users encounter with high-quality software.)

Our analysis is not deep or precise—we are not experts in the cleanroom methodology. We mean only to suggest the kind of analysis to which formal methods should be subjected.

8. Reliability Amplification and Self-testing

A number of researchers are investigating the idea of "self-testing" programs, in which consistency checks are automatically applied as the software executes. In one approach [Blum] an input is first treated directly, then decomposed, given to the same software, and the results recombined to check the original result. For example, matrix multiplication can be decomposed by partitioning the matricies. If a program continually gives the same result from randomly chosen decomposed versions of an original input, it is correct for that input with high probability. In a more conventional approach [Antoy], an abstract-data-type implementation is given an axiomatic specification executable by rewriting, and the specification is executed in parallel with the ADT, checking each value of the type produced, even when the ADT is embedded in an application. Finally, a suggestive empirical study [Leveson] comes to the preliminary conclusion that consistency checking of internal data structures is a better means of detecting software failure than is multiversion programming.

Self-testing programs are not obviously related to reliability amplification. However, we can imagine a self-testing program being reliability tested. Self-testing serves as an oracle for random testing, which may make the reliability measurement practical, but can we argue for amplification? In each approach, each test point actually tests more than a single outcome. Blum tries input decompositions, Antoy verifies internal results from ADTs, and Leveson suggests internal-state consistency checks. These extra pieces of information can be used in an argument, similar to that given in Section 7 for the cleanroom method, that failure not observed on a single test point has a wider significance. However, it seems unlikely that any of these techniques can yield orders of magnitude amplification factors, since they multiply each test's significance by at most a small factor, and do so only at extra testing expense.

9. Conclusions

We have suggested a framework of "reliability amplification" for validation of software, to establish statistical confidence that software is correct. Testability measurements can show that the software "wears its faults on its sleeve," that any fault will likely appear as a failure. Reliability testing shows that failures are not likely. The conclusion is that there are no faults, with a confidence derived from the reliability measurement. This idea was quantitatively applied to systematic testing methods, and qualitatively applied to a software process methodology.

Much work remains to be done in improving the models for testability and reliability. Design for testability is a new idea that can be applied at many levels, from the statements of programming languages through modules and subsystems. Partition-based reliability amplification is promising, both as a practical way to divide and conquer a hard problem, and as a theoretical approach to plausible assumptions about failure rates for software.

References

[Antoy]

S. Antoy and R. G. Hamlet, Self-checking against formal specifications, *Proc. Int. Conf. on Computing and Information,* Toronto, May, 1992, 355-360.

[Blum]

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

[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.

[Duran & Ntafos]

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

[HamletA]

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

[HamletB]

D. Hamlet, Random Testing, Portland State University Department of Computer Science Technical Report TR 93-5, March, 1993.

[Hamlet & Taylor]

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

[Hoffman]

D. Hoffman, Hardware testing and software ICs, *Proc. Pacific Northwest Software Quality Conference,* Portland, OR, 1989, 234-244.

[Leveson]

N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall, The use of self checks and voting in software detection: an empirical study, *IEEE Trans. Software Eng.* SE-16 (September, 1990), 432-443.

[Miller et al.]

K. Miller, 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.

[Myers]

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

[Thayer et al.]

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

[Tsoukalas et al.]

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]

J. M. Voas, A dynamic failure-based technique, *IEEE Trans. Software Eng.* SE-18 (August, 1992), 717-727.

[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 & Jeng]

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