Connecting Test Coverage to Software Dependability

Dick Hamlet

Portland State University
Center for Software Quality Research

Abstract

It is widely felt that software quality, in the form of reliability or "trustworthiness," can be demonstrated by the successful completion of testing that "covers" the software. However, this intuition has little experimental or theoretical support. This paper considers why the intuition is so powerful and yet misleading. Formal definitions of software "dependability" are suggested, along with new approaches for measuring this analog of trustworthiness.

1. The idea of test coverage

From the beginning of computer programming, clever people have looked for clever ways to catch the mistakes that too easily enter even the best programs. 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 popular. They have been repeatedly reinvented and captured in testing tools. Control coverage is intuitively appealing because software defects appear as textually located *faults* along the path of execution, giving rise to failures. Functional coverage is even more basic—function-based trials put the software through its paces.

Although these coverage ideas have appeared in the technical literature for more than 20 years, their penetration in practice is surprisingly shallow.

The idea of *mutation coverage* [Hamlet77, DeMillo78, Howden82], in which the computational structure of a program is exercised, is not so obvious as control coverage. Mutation remains a research technique rather than one exploited in practice, partly because it is computationally expensive, but probably more because it is harder for people to use than is control coverage. Mutation is a double-edged sword: it appears that mutation-adequate tests are difficult to "fool" in the sense of detecting more faults than other kinds of structural coverage; but at the same time, the test data needed to attain mutation coverage are less obvious, and harder to systematically amass,† than data for control coverage. Despite its limited practical acceptance, mutation may be the key idea that connects coverage testing with measures of software quality. This connection arises because mutation can be viewed as massive fault-seeding, a technique not for finding failures, but for clocking how many may have been missed.

1.1. State-of-the-art coverage testing

Current "best practice" for coverage testing as described by Brian Marick, the author of a high-quality public-domain tool (GCT), can be paraphrased as follows [Marick91]:

> Functional testing from the specification drives all systematic testing. An initial testset should be derived by listing as many cases and situations as can be identified in the specification, and then creating test points that cover these. The initial testset should be augmented by other functional cases that can be identified only by examining the program design; the additional "functions" are those involving important data structures, and often cover exception or boundary situations. Call the combined set of test data generated by this analysis the "functional" testset. The quality of the functional testset is now assessed by measuring its structural coverage. Deficiencies in coverage should be addressed by returning to the specification and design, adding to the list of cases and situations to include functions that were missed. Corresponding additions to the functional testset should account for all uncovered structure.

† In control-flow testing, one must deal with *infeasible paths* that make it impossible to attain coverage, but these are easy to grasp in principle, and in practice not hard to identify. The corresponding problem for mutation testing, that of *equivalent mutants*, is both conceptually more difficult, and in practice a frequent and difficult stumbling block.

Marick believes that the best test points are those that cover multiple functional cases, because such "complex" tests are most likely to excite failure modes involving function interaction, and reducing the number of tests saves time. A contrary belief has some experimental support: Thévenode-Fosse successfully used uniformly distributed random tests within each functional class [Thévenode-Fosse93].

It is instructive to consider using Marick's method with mutation instead of control-flow coverage. When the quality of the "functional" testset is assessed by mutation coverage, it will be found to be quite inadequate. It is universally observed by mutation users that the mutation criterion is difficult to satisfy. In practice, the tester proceeds to do precisely what Marick counsels against: each live mutant is examined and a special test is devised to kill it (or the tester becomes convinced that the mutant is equivalent). Finding tests to kill mutants is a difficult process by any means; it would be even more difficult to do as Marick advises, to eliminate a mutant by discovering an underexercised function that left it alive. Thévenode-Fosse's method is also impractical, because randomly-generated tests do not often kill mutants [Frankl&Weiss94§].

Carrying the argument one step farther, suppose that instead of mutation, fault seeding were used as the "coverage" criterion to assess the quality of Marick's functional testset. When the functional testset misses some of the seeded errors, if the seeding represents "real" faults well, the hit ratio measures the quality of testing. But to then purposefully seek to "improve" the coverage so that the remaining seeded faults are found is nonsensical. 20% hit ratio (say) of the functional testset is a valid estimate that 20% of all faults have been found; 100% hit ratio *attained using knowledge of the seeded faults*, is meaningless.

Thus coverage testing practiced using control-structure coverage as a check on functional coverage, is a plausible systematic way to search for software failures. But when we think of shifting "coverage" toward a measure with significance for the real quality of the tested software—seeded faults—forcing coverage does not seem a plausible way to gain confidence in quality.

§ For a small string-matching routine with only 356 non-equivalent mutants and 49 executable DU pairs, they found that only .019% of randomly selected testsets of size 8 achieved mutation coverage, while 14% achieved all-uses dataflow coverage—mutation coverage was almost 1000 times harder to achieve.

1.2. Evaluation criteria for coverage testing

What objective criteria could be used to decide questions about the value of coverage testing in general, or to compare the merits of different coverage techniques or testing procedures? Historically, techniques were evaluated either by unsupported theoretical discussion, or by "experiments" based on circular reasoning. The inventor of a new coverage technique (say "*my-T*") was expected to argue that *my-T* was subjectively cleverer than its predecessors, and to compare *my-T* to other techniques *in terms defined only by themselves*. For example, it was common to find a testset for *my-T* coverage of some program, then see what fraction of branch coverage that testset attained; or, to find a testset for branch coverage and see what its *my-T* coverage was. *My-T* was considered to be validated if its testset got high branch coverage but not the other way around. Such studies are really investigating special cases (for a few programs and a few testsets) of the "subsumes" relation, to be discussed next.

Control-flow coverage methods can be compared based on which method is more "demanding." Intuitively, a method is "at least as demanding" as another if its testsets necessarily satisfy the other's coverage. The usual name for this relationship is *subsumes*. 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" was 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, it was 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+91] showed that the subsumes idea 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. An example was given in which statement testing was more likely to detect a failure than was branch testing; however, even the contrived example was unable to evidence much superiority for the "less demanding" method, indicating that "subsumes" is not so misleading after all. In a recent promising paper [Frankl&Weyuker93], the subsumes relationship is refined (to "properly covers") and it is shown that the new relationship *cannot* be misleading in the probabilistic sense.

The subsumes relationship began as a generalization of how well test methods do in each other's terms, that is, without any necessary reference to objectively desirable properties of the software or the testing. A "more demanding" method Z that strictly subsumes X is actually better only if we assume that what X and Z demand is really useful beyond their mere definitions. The introduction of "misleading" was an attempt to incorporate an outside objective measure (failure-detection ability), and to show that for natural coverage measures, "subsumes" does not necessarily relate to the objective measure.* Failure-detection probability is thus currently enshrined as the accurate measure of test quality†, replacing a circular use of "coverage" to assess the quality of "coverage."

Failure-detection probability has also been used in another comparison between testing methods, a comparison with a link to software reliability. A series of papers [Duran&Ntafos84, Hamlet&Taylor90, Jeng&Weyuker91, Tsoukalas+91] have compared so-called "partition testing" to random testing, with the result that failure-detection measures for these methods are within an order of magnitude of each other, except in pathological cases+. These studies must make a number of doubtful assumptions, not the least of which is that "partition testing" is the same as "coverage testing." The classes of a partition must be disjoint, but the common structural methods such as statement-coverage testing do not have disjoint classes§. Despite flaws in the theory, it is difficult to escape the impression that coverage testing and random testing should have roughly comparable failure-detection probabilities.

For random testing conducted using an operational profile, the failure-detection probability has another meaning: it is the failure intensity, the reciprocal of the mean time to failure (MTTF). Again, there are many assumptions needed to justify the application to software, but the conventional theory is certainly an approximation to the situation in which tests are a sample drawn independently from real usage, and thus somehow statistically representative of that usage. Let us also accept this theory, and focus on the connection between coverage testing and random sampling for reliability.

Unfortunately, even granting that the partition/random comparison applies to coverage testing, and that failure-detection probability for random testing determines MTTF, the only conclusion that can be reached is the negative one that coverage testing is at best no more significant than random testing, or at worst of no significance. A random test can establish upper confidence bound α that the failure intensity is not above θ on the basis of N tests with F failures. The formula connecting these quantities in the TRW software reliability theory [Thayer+78] is:

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

If a coverage test is an equally good statistical sample, it might realize a similar or better bound on failure intensity. But the very intuitive factors that make coverage testing desirable for finding failures, make its failure-detection probability different from a failure-intensity. Coverage testing achieves superior failure detection precisely by sampling not the operational profile, but according to classes that emphasize failure [Hamlet&Taylor90, Jeng&Weyuker91]. These classes bear no necessary relation to the operational profile, and hence the failure intensity may be large even though the failure-detection probability for coverage is small, if coverage testing found failures in low-profile-usage areas, and neglected high-profile-usage areas. (Section 2.2 shows that there is no limit to the disparity.)

Thus the comparable or better failure-detection probabilities of coverage testing vis a vis random testing are not failure-intensity predictions at all, and there is no support in the comparison for the position that coverage implies reliability.

---

* It is easy to devise "unnatural" coverage measures in which "subsumes" is more misleading. For example, the coverage measure (Z) "more than 70% of statements executed" strictly subsumes (X) "more than 50% of statements executed," but if the statements executed using Z happen to be all correct ones in some program, while those executed using X happen to include its buggy code, then X is actually better than Z for this program.

† A number of people, including Weyuker, have questioned whether failure-detection probability is the appropriate measure of testing quality, but it remains the only one on which work has been done.

+ A typical pathological case occurs when the partition includes an input equivalence class containing only a few points, on which the program fails. If these points mark the only failures in a large input domain, they are "needles in a haystack" to random testing, and the partition does much better. On the other hand, although badly chosen partitions may favor random testing, its superiority can never be too great as the number of test points increases, since the worst case occurs when the partition test "wastes" many of its points in classes containing no failure inputs, and random testing must waste points there, too.

§ It is suggestive that the best result in improving "subsumes" [Frankl&Weyuker93] turns on the way in which the classes of a coverage method overlap.

Finally, data from direct observation of the relationship between coverage testing and reliability in the field might be experimentally obtained. Such experiments, like all experiments with real software, will be notoriously difficult to do, but we can expect to see some data as testing practices are controlled and monitored, and field failures carefully recorded. A recent paper presents preliminary data for one system [Dalal+93, Fig. 2], in the form of a scatter plot of unit-test statement coverage *vs.* system-test faults later attributed to those units. The plot shows many examples of high-coverage routines that later revealed no faults, and a few examples of low-coverage routines that later showed several faults; however, it also shows many routines with medium to low coverage, that showed few or no faults. The data is suggestive, but its significance depends on the quality of the system test: was it a reliability measurement using high-volume random testing from an operational profile? This paper also cites another study [Piwowarski+93] in support of coverage "effectiveness," but the cited paper only postulates a model of the coverage/quality relationship, and its authors admit to "varying the value" of a "constant" in the model in order to fit data for one system. The difficulty of performing these experiments is in marked contrast to the modest results.

1.3. Summary: coverage finds faults, but...

The strength of coverage testing has always been its obvious plausibility: if functions or structure remain uncovered by tests, the developer has *no* information about the quality of those aspects of the program. However, this theoretical backing supports only the *necessary* part of the argument: failures may lurk in untested places, but coverage may not find them, either.

Theoretical work using measures like "properly covers," and careful experimental comparisons of failure-detection rates [Frankl&Weiss91], are promising ways to establish that coverage testing does uncover failures, probably better than does random testing, and to learn which coverage methods are best at failure detection. However, there is as yet very little practical evidence of a connection between coverage and reliability, and no theoretical basis for such a connection.

The persistent idea that coverage supports reliability speaks strongly for an intuition that has not been captured by existing theories. Why do practical testers believe that eliminating the faults found by coverage testing is enough to make products "work"? Why are engineers in safety-critical applications reluctant to reduce coverage testing even in favor of more direct statistical testing for reliability? In the remainder of this paper we suggest directions for theoretical research to address these questions.

2. Software dependability

Intuitively, software quality must be measured in terms of results, not in terms of effort expended, however well meant. Thus naive measures of testing like "hours spent testing," or even the more sophisticated failure-detection probability discussed in Section 1, capture only how hard people (or techniques) tried, not what they demonstrated about software quality. While MTTF calculations and their confidence bounds—conventional reliability measures—are better, they do not get at the heart of what it means to believe software is "trustworthy." The operational profile is to blame here, because a potential catastrophic failure in a very low-usage function has little effect on reliability, but makes it impossible to trust the software. It will not help to weight profiles by cost, because disaster may cross all lines of functionality. For example, the probability of file-system destruction by an untrustworthy program might not depend on what a user requested the program to do†. Thus any measure of trustworthiness must be independent of the operational profile.

It seems inescapable that in principle, *no* failures can be tolerated in a trustworthy program. In practice, when failures occur, some may be of minor importance and thus impact only reliability, not trustworthiness. But in the situation that no failures have been detected, theoretical consideration of trustworthiness must treat any failure as potentially catastrophic. It is obviously important to be able to handle the case of zero detected failures, since for critical software it will probably always prove impractical to observe failure in a final program before release [Butler&Finelli91, Littlewood&Strigini93]. However, no one believes that perfection can be obtained in a human endeavor as complex as software development, so a statistical measure is needed for trustworthiness.

"Zero-defect" software is the goal of many development methods, today particularly those that seek to apply formal methods to specification and

† Catastrophic failures might be given their own heavily weighted classes in the operational profile; indeed, this is what safety fault-tree analysis seeks to do. But in general, catastrophic failures occur because no one has any conception of the situations that lead to them, so "failure classes" cannot be isolated.

design. Their proponents make an argument that is remarkably similar to the one for coverage testing: 'Use my method diligently,' they say, 'and your hard work will pay off in defects caught before they enter the software.' But while it is plausible that defects will be caught, it is much more difficult to establish that none will enter in spite of the formal method. Proposed methods do not attempt a quantification of quality achieved, and none seems a good candidate for theoretical analysis; coverage testing is a more promising context for studying trustworthiness.

To distinguish a technical notion from the intuitive software property of "trustworthiness," let us call the statistical property we hope to define, "dependability" of software. Three definitions **U,W,Z** come immediately to mind:

The *dependability* of a program P could be:

U: P's reliability for a **u**niform input distribution

W: P's reliability in the **w**orst (least reliable) class of its operational profile

Z: the probability that P has **z**ero defects

In choosing among these definitions there are two considerations: (1) How well does the definition capture the intuition that a (technically) dependable program is (intuitively) trustworthy? and (2) What is the connection between coverage testing and dependability?

2.1. Intuitive considerations of dependability

Definitions U and W are both meaningful because they are based on reliability. Definition U is certainly profile independent, but at the cost of selecting the blandest possible profile. Most real profiles would emphasize some portion E of the input domain. If inputs in E could lead to failure, Definition U would yield a dependability that is better than the actual reliability. This overly optimistic property is sufficient reason for rejecting Definition U.

Definition W is framed to avoid Definition U's false optimism. By ignoring all but the least reliable part of the input domain, it will assign a dependability that is always more pessimistic than the reliability for any real profile. However, Definition W is heavily dependent on the classes that make up the operational profile. As Section 2.2 shows, these classes leave a good deal to be desired as a foundational idea. Furthermore, Definition W does not seem to speak to the objection that failures can cut across class boundaries.

Definition Z is the most appealing—it directly captures the idea of a profile-independent, maximally pessimistic trustworthiness. However, Definition Z cannot fall back on reliability for its technical meaning. Reliability essentially involves two probabilities, the failure intensity and statistical confidence in the sampling that measures it, while Definition Z speaks of a single probability. New ideas are required to make Definition Z work, as described in Sections 2.3 and 2.4. The promise of these ideas is that they appear to circumvent the argument that reliability cannot be practically measured by testing [Butler&Finelli91].

2.2. Reliability within input subdomains

It is illuminating to consider subdividing the input domain, and applying reliability theory to its parts, as Definition W requires.

Suppose a partition* of the input domain 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 $\Theta = \sum_{i=1}^{k} p_i \Theta_i$. 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.

Suppose failure-detection probability Θ^M is measured for a particular profile D_M, and taken to be the failure intensity, yet profile D_A is the actual profile, with failure intensity Θ^A (identical to the failure-detection probability using D_A). It is evident that Θ^M bears no necessary relationship to Θ^A, because the probabilities that profiles' points fall into each subdomain may be arbitrarily different. D_M may emphasize failure-free subdomains so that Θ^M is near 0, while D_A's intensity is near $\Theta_{\max}$; or, it may be the other way around. The uniform distribution of Definition U need not have a failure detection probability close to the correct intensity Θ^A, nor need it be close to $\Theta_{\max}$.

By coverage testing without failure, using the Thévenode-Fosse technique of a uniform distribution

* A partition is defined as having disjoint subdomains that together exhaust the input domain. The subdomains of most structural testing methods overlap, and so do not form a partition. The difficulties raised in this section also occur in the more complicated overlap case.

within each S_i, an upper bound can be established on Θ_{max}, and hence on the overall failure intensity for all distributions. The dependability according to Definition W is thus $1-\Theta_{max}$. To measure the dependability according to Definition W will require k times as many random tests as measuring the reliability for one profile, because we don't know which subdomain is worst, and so must bound all k of the Θ_i. In view of the impracticality of testing in the ultra-reliable region, Definition W is also impractical. However, there is a worse conceptual difficulty with subdomain reliabilities. The partition is apparently arbitrary, yet two extreme cases are not acceptable. If $k=1$, with $\Theta = \Theta_1 = \Theta_{max}$, the only subdomain is the whole input domain. Then Definition W becomes Definition U, which is intuitively unacceptable. At the other extreme, as k becomes very large, some subdomains might be so small that individual failures would make $\Theta_{max} \approx 1$, and hence the dependability nearly 0. Although this is certainly the correct pessimistic result for software that can possibly fail, it is not useful.

These extreme cases suggest that the difficulty with Definition W lies in the assumption that the k failure intensities Θ_i are constants. This assumption is the reflection in reliability theory of the intuitive notion that points in a subdomain should be "treated the same:" within each subdomain, the program should be equally likely to fail on each point†. Subdomains have not been studied for this property.

Even if its conceptual difficulties were resolved by a deeper understanding of subdomains, Definition W, like any definition based on reliability, would not allow the practical measurement of dependability.

2.3. Testability and probable correctness

The idea of "probable correctness" expressed by Definition Z is difficult to make meaningful—a program either has defects or it doesn't. The intuition behind the definition is that the probability arises in an uncertain assessment through testing. An early attempt to use reliability alone as the basis for Definition Z [Hamlet87] was not very successful. Voas has proposed [Voas&Miller92] that reliability be combined with *testability* analysis to do better. Reliability measurement is an upper bound on failure intensity; testability is a *lower* bound on failure intensity.

For a test T, a program has testability h with confidence C iff:

> If the program can ever fail, we have confidence C that it will exhibit failure intensity greater than h during T§.

A high testability thus describes a program that "wears its faults on its sleeve:" if it can fail, it will be seen to fail.

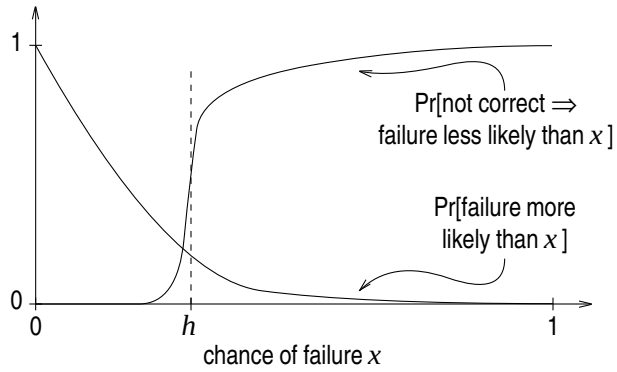

Figure 2.3-1. 'Squeeze play' between testability and reliability

The "squeeze play" between the two bounds is shown in Fig. 2.3-1 [Hamlet&Voas93]. The falling curve is the confidence from reliability testing; the step function comes from measuring a testability h. The probability that the actual failure intensity lies to the right of h can be read from the reliability curve, and can be made small by moving h to the right; the probability that the failure intensity lies below h is nearly zero *if the software can fail*. As h increases, it becomes less and less likely that the failure intensity lies anywhere in [0,1]. The only other possibility is that the software *cannot fail*, and the combination of the two confidence bounds is its probability, that is, precisely the dependability of Definition Z.

Testability may be a practical way to measure dependability. For example, using Voas's simple model, a testability of 0.001 would require only a 10,000-point reliability test for a dependability of 99.995%.

To use the "squeeze play" requires that reliability measurement be made using an operational profile, and it is convenient to make the tests of this measurement also serve for testability. But the Voas theory is

† If all of the program's failure inputs are grouped in subdomains containing no successful inputs, the assumption is justified. But no one expects this of a coverage partition—it would mean knowing exactly which tests will expose every program defect.

§ Voas's very rough theory ignores the size of T, and C; nevertheless, Fig. 2.3-1 shows a less than ideal testability curve.

too crude to be able to discuss the relationship between profiles and testability. Investigation of this topic is promising because intuitively, testability measurements might be drawn from coverage-testing subdomains. The practicality of the squeeze play also raises the opportunity to use the idea of Definition W. Instead of using the squeeze play with an operational profile over the whole input domain, it could be used with a uniform distribution in each class of the profile, and the worst class taken as a pessimistic value for the dependability using Definition Z. This would seem to eliminate any profile influence, at a modest cost.

2.4. Self-checking programs

A quite different view of "probable correctness" has been suggested by Blum [Blum&Kannan89]. He solves the problem of profile independence by avoiding the input domain altogether. Instead of considering program quality on *all* inputs (what might be called a "uniform" theory), he suggests a "pointwise" theory: program quality is to be expressed as a probability that the program result is correct *for one particular input*. Definition Z can be modified (definition **ZP**) to correspond to this view:

ZP The *(pointwise) dependability* of a program P at input x is the probability that P computes the correct result on input x

Blum argues that Definition ZP is what people really mean by trustworthiness—users care only about each particular computation they perform, not about all possible computations.

Blum's insight was that the probability defined by Definition ZP can be calculated at run time, for each input a program receives. It can be calculated efficiently for programs able to probabilistically check their results; roughly, the program repeats "random" variations on the particular input it is asked to handle, and compares the results. Repeated agreement makes it very unlikely that the result is wrong. It is crucial that something be known about the program's "random" behavior, which links Blum's theory to reliability.

Testing and confidence bounds enter the theory in a novel way. Testing is needed to establish statistical confidence in the necessary "random" properties of a program. But when a particular input is given, a few trials added to the run yield a high dependability. For example, program agreement for 18 random variations on input x_0 gives a pointwise dependability at x_0 of $1-2^{-18} \approx 99.9996\%$. However, should there be disagreement among the variations, the program can report only something like "not very dependable at x_0." Intuitively, the portion of the input domain on which it cops out is related to the quality of the program, as measured by the confidence in prior random testing.

3. Conclusions and future work

The connection between coverage testing and reliability has no theoretical, and little experimental support. A formal notion of trustworthiness, which we call dependability, seems more promising for establishing this connection than reliability itself. Three promising research directions are:

(1) The relationship between failure intensity and input subdomains needs investigation. It bears on the question of whether there really is a sensible failure intensity, and what are appropriate subdomains, both for profile-independent reliability, and for coverage testing.

(2) Testability research is in its infancy. Voas's initial work does not consider the size or profile of testsets, nor the confidence to be placed in testability estimates. However, testability's complementary role to reliability testing, particularly in reducing the number of test points required for significant interpretation of results, is very promising.

(3) Blum's own investigation of pointwise reliability is concerned more with algorithms, their modification and suitability for self-checking, than with dependability. But Definition ZP opens many new and promising research questions. It should be possible to sample (and thereby quantify with confidence bounds) the quality of the program on "random" inputs. Considering the distribution of such samples may help to extend the self-checking theory to more programs, and to connect it to the subdomains of coverage testing.

Acknowledgements

An anonymous referee who rejected my paper for the recent Foundations of Software Engineering Conference, pointed out the similarity of mutation to error (fault) seeding.

References

[Blum&Kannan89]
M. Blum and S. Kannan, Designing programs that check their work, *Proc. 21st ACM Symposium on Theory of Computing*, 1989, 86-96.

[Butler&Finelli91]

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.

[Dalal+93]
S. R. Dalal, J. R. Horgan, and J. R. Kettenring, Reliable software and communication: software quality, reliability, and safety, *Proc. 15th ICSE*, Baltimore, MD, May, 1993, 425-435.

[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&Ntafos84]
J. Duran and S. Ntafos, An evaluation of random testing, *IEEE Trans. Software Eng.* SE-10 (July, 1984), 438-444.

[Frankl&Weiss91]
P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria," *Proc. Symposium of Software Testing, Analysis, and Verification (TAV4)*, Victoria, October, 1991, 154-164.

[Frankl&Weiss94]
Personal communication.

[Frankl&Weyuker93]
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.

[Hamlet&Taylor90]
D. Hamlet and R. Taylor, Partition testing does not inspire confidence, *IEEE Trans. Software Eng.* SE-16 (December, 1990), 1402-1411.

[Hamlet&Voas93]
D. Hamlet and J. Voas, Faults on its sleeve: amplifying software reliability testing, to appear in ISSTA '93, Boston, June, 1993.

[Howden82]
W. E. Howden, Weak mutation testing and completeness of test sets, IEEE Trans. Software Eng. SE-8 (July, 1982), 371-379.

[Jeng&Weyuker91]
B. Jeng and E. Weyuker, Analyzing partition testing strategies, *IEEE Trans. Software Eng.* SE-17 (July, 1991), 703-711.

[Littlewood&Strigini93]
B. Littlewood and L. Strigini, Validation of ultrahigh dependability for software-based systems, *CACM* 36 (Nov., 1993), 69-80.

[Marick91]
B. Marick, Experience with the cost of different coverage goals for testing, *Proc. Ninth Annual Pacific Northwest Software Quality Conference*, Portland, OR, October, 1991, 147-164.

[Piwowarski+93]
P. Piwowarski, M. Ohba, and J. Caruso, Coverage measurement experience during function test, *Proc. 15th ICSE*, Baltimore, MD, May, 1993, 287-301.

[Thayer+78]
R. Thayer, M. Lipow, and E. Nelson, *Software Reliability*, North-Holland, 1978.

[Thévenode-Fosse93]
P. Thévenode-Fosse and H. Waeselynck, Statemate applied to statistical software testing, *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, Cambridge, MA, June, 1993, 99-109.

[Tsoukalas+91]
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&Miller92]
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+91]
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.