Theoretical Comparison of Testing Methods†

Richard Hamlet

Computer Science Department
Portland State University
Portland, OR 97207 USA
(503)464-3216
hamlet@cs.pdx.edu

Abstract

Comparison of software testing methods is meaningful only if sound theory relates the properties compared to actual software quality. Existing comparisons typically use anecdotal foundations with no necessary relationship to quality, comparing methods on the basis of technical terms the methods themselves define. In the most seriously flawed work, one method whose efficacy is unknown is used as a standard for judging other methods! Random testing, as a method that *can* be related to quality (in both the conventional sense of statistical reliability, and the more stringent sense of software *assurance)*, offers the opportunity for valid comparison.

1. INTRODUCTION

Testing methods are notoriously difficult to compare, because most testing lacks a theoretical foundation. For example, it certainly seems reasonable to test until every statement of a program has been executed. It would be stupid not to do so, since a never-executed statement could harbor an arbitrarily bad bug. But having achieved statement coverage, what does the tester know? It is easy to show that statement coverage can be achieved by many different test sets, some of which expose bugs while others do not. How then can the worth of statement testing be assessed? So long as the relationship between a method's basis (in the example, covering statements) and properties of the software (here, the occurrence of bugs) remains imprecise, meaningful comparisons between methods, indeed comparison between two applications of the same method, cannot be made.

There are a number of apparently sensible, analytic comparisons that *can* be made among testing methods, which on analysis are less useful than they seem. The comparisons are precise, but they utilize parameters unrelated to true efficacy of the methods, and so can be fundamentally misleading. The "subsumes" ordering is of this kind.

Empirical comparison of methods seems the obvious solution to all these difficulties. Its two drawbacks are that it depends heavily on the particular programs and environments selected for trial, and often on the relative ability of human subjects who participate in the evaluation. Again, the real fault is with our lack of theory: because we do not know how a method is related to software properties of interest, we do not know how to control for the important variables.

In §2 below we critique past attempts to compare testing methods, showing where these are flawed in principle. In §3 we suggest a new basis for comparison.

2. PRIOR COMPARISON OF TESTING METHODS

In situations where testing resources are scarce or where tests must demonstrate real software quality, the practitioner is all too aware of the need to compare methods. From the opposite side, the inventor of a new method is called upon to justify it by comparison to existing methods. When the call for comparison is loud enough, comparisons will be brought forward, but in the absence of theoretical foundations they cannot be trusted.

2.1. Empirical Comparisons

Empirical studies comparing testing methods must contend with two potential invalidating factors:

(1) (*programs*) A particular collection of programs must be used—it may too small or too peculiar for the results to be trusted.

(2) (*test data*) Particular test data must be created for each method—the data may have good or bad properties not related to the method.

The use of human beings to generate test data is one way in which (2) can occur. Because test generation requires the use of information about the program or its specification, a human being may use this information in some (unconscious) way, distorting the test. It is then certainly unfair to attribute a quality to the test method that in fact resides in

† Work supported by NSF grant CCR-8822869.

the human testers. An experimenter can attempt to control for this "nose-rubbing effect" [11], but since human skill varies widely, and is little understood, control is imperfect.

The program choice (1) for an experiment may be biased both by program attributes and by the kind of faults present. If the programs are real ones that arose in practice these difficulties merge; however, if the faults are seeded, or the programs arose in a training environment, the bias can be twofold. Since program characterization is not well understood, no quantitative measures of their properties are available. It certainly seems safer to trust results of a study in which the researchers worked hard to control the environment [1], but some unknown factor can confound the best human efforts when understanding is lacking.

Factor (2) can be handled in several ways:

a) *Worst-case analysis.* By investigating *all* test sets that satisfy a method, one can accurately predict the worst that method might do. The main drawback of worst-case analysis is that it is tedious and usually cannot be automated. There have been few followers for Howden's careful study of path coverage on a set of textbook programs [13]. Furthermore, as we show in §2.3 below, capturing worst-case analysis as a comparison technique makes all methods technically incomparable.

b) *Random selection.* When a method involves drawing test points at random from a distribution, there should be no bias introduced. However, if the required distribution is unknown, then using programs for which it is given biases an experiment in favor of an actually impractical method. Random selection of test sets may also be used for methods that themselves have no probabilistic aspect. If the test sets have a smooth distribution, sampling them yields an average-case result. (That any useful real situation could be known to have such a distribution seems unlikely, however.) Even though it may not yield a true average case, using random selection to satisfy a coverage method (instead of employing human subjects) can control the nose-rubbing effect.

c) *Automatic test generation.* Some methods (notably path testing [23] and recently mutation testing [5]) can be used with algorithms or heuristics that generate data to satisfy the method. There can be a hidden bias through the choice of programs, those on which a heuristic happens to work, for example. Also, it must be remembered that the results apply only to that narrow version of the method using the generation algorithm. (Further discussion appears in §2.4.)

It is not unfair to say that a typical testing experiment uses a small set of toy programs, with uncontrolled human generation of the test data. That is, neither (1) nor (2) is addressed. Such studies can be helpful in understanding methods, but they do not provide comparisons. It is interesting to note that an exceptional recent study [16] in which real programmers tested real programs, showed that the most significant factor in finding bugs was not the method, but the skill of the human subject.

2.2. The "Subsumes" Ordering

The majority of precise, published comparisons between testing methods use the following ordering:

> Method *A subsumes* (or *includes*) method *b* iff a test that satisfies *A* necessarily satisfies *b*.

For example, a "method" in which every test result is required to be correct subsumes one in which only 80% of the results must be correct. The inclusion is strict because a test barely satisfying the 80% method must fail to satisfy the other.

Unfortunately, only some methods, those with a common basis, can be compared using *subsumes.* Variants of path coverage can be compared, and these fall into a fairly simple hierarchy [24, 19]. But many methods are incomparable—neither subsumes the other [28]. Trivial differences can make methods technically incomparable, as in the three mutation variants [8, 4, 30].

The subsumes relation is more seriously flawed than by limited applicability, however. All algorithmic testing methods are necessarily imperfect in that none can guarantee correctness of programs in general [14, Ch. 4]. When one imperfect method subsumes another, the relation expresses nothing about the ability to expose faults or assure quality. It can happen that *A* subsumes *b*, but *b* actually addresses these real properties better. Intuitively, the trouble with subsumes is that it evaluates methods in their own terms, without regard for what's really of interest. For example, consider statement testing as method *A*, and let *b* be "haphazard" testing in which there is no requirement but that the test set be nonempty and the results be correct. Obviously *A* subsumes *b*. But *b* may find faults that *A* does not. Consider the Pascal function:

```
function Q(X: real): real;
  begin
    if X < 0 then X := -X;
    Q := sqr(X)*2
  end
```

Suppose this program is intended to compute the function of t: $2\sqrt{|t|}$. (The programmer has inadvertently used the Pascal square function instead of the square root.) The test input set: {1, 0, -1} achieves statement coverage yet uncovers no failure, while {3} is haphazard (and does not cover all statements), yet does expose a failure.

Such examples are not really pathological. Because there is no necessary connection between covering statements and finding faults, paying attention to the former diverts attention from the latter. In this case the haphazard tester ignored the absolute value part of the program, and concentrated on the square(root), with good results. In a similar way, the dataflow variants of path testing [24, 15]

may prove superior to full path testing (which subsumes them), because they concentrate attention on important cases that might be trivially treated in the welter of all paths. The idea of "trivial" satisfaction of methods will be further examined in §3.2.

If we include the human process of selecting test points, or use mechanical means to generate test data, the subsumes ordering shows its flaws more clearly. One method may be so difficult to understand that in generating test points, trivial choices are encouraged. Another method, easier to use, may lead to more representative data. Nothing stops the poorer, more complex method from subsuming the better, simpler one. Coverage tools exacerbate rather than alleviate this problem: when a person trys to fill gaps in a hard-to-understand coverage method, as required by a test tool, trivial data comes naturally to mind, and the tool's subsequent approval glosses over the poor test quality.

2.3. Gourlay's Comparison

In his excellent theoretical framework for testing [7], Gourlay defines a partial ordering $M \geq N$ between methods M and N, which claims to capture the intuitive idea that one method is more powerful than another. His definition gives some of the same results as *subsumes* (for example that BRANCH ≥ STATEMENT but STATEMENT ≱ BRANCH for the two methods with those names). We agree exactly with Gourlay's intuition about method comparison (presented as motivation to his Definition II.B-1). He states that for method M to be stronger than N should mean that if N exposes a failure, then M must necessarily do so as well. However, his Theorem II.B-7 states that when one method subsumes another, his ordering holds. The results about BRANCH and STATEMENT are consistent with Theorem II.B-7, but appear to conflict with the critique of *subsumes* in the previous section.

Because Gourlay's theoretical framework is a good one, it can be used to explicate the problem and to show that Definition II.B-1 does not capture his intuition. With a better definition we show that no algorithmic method is ever comparable to any other such method. This formal result exactly captures the general case against *subsumes* presented in §2.2.

It is Gourlay's "choice construction" (Definition II.A-3) that causes the difficulty. In a testing system using this construction, a method corresponds to a collection of different tests, but the method "is *OK*" if any test in the collection executes successfully. Thus, for example, statement testing for a given program and specification is a method with many potential tests, all executing every program statement, but some may execute successfully, and some may expose a failure. Gourlay calls statement testing *OK* if at least one of the tests is successful. He then takes $M \geq N$ to mean that for all programs p and specifications s

$$p\ OK_M\ s \Rightarrow p\ OK_N\ s,$$

where $p\ OK_Q\ s$ stands for the success of method Q. But for a choice system, Gourlay's "success" is not what we intuitively think of as success.

Recalling the discussion of §2.2, erroneous comparison of methods results from the existence of several tests that satisfy each method, only some of which expose present bugs. For an incorrect program, call a test that succeeds (and thus misleads the tester) a *misleading* test. If a method is effective, that is, if it allows a mechanical decision as to whether a triple (T, p, s) consisting of a test, program, and specification is satisfactory, then it is a consequence of the undecidability of the program-equivalence problem that misleading tests exist for that method. At the same time, if a program is not correct, then some test exposes this fact, and because most methods are monotonic (in the sense of Weyuker [29]), any method can be made to incorporate such a test. Call a test that finds a failure in this way an *exposing* test. An exposing test and a misleading test are enough to intuitively invalidate any comparison between methods. Suppose method A were "better" than b. Let t_A be a misleading test satisfying method A and t_b be an exposing test satisfying b. Then in fact A is not better than b using Gourlay's intuitive definition. However, turning the relation around, exactly the same argument applies, hence the methods cannot be compared.

We can capture this intuitively correct property of methods in Gourlay's theory by altering his definitions. First, define a *testing method* to be a relation over tests, programs, and specifications, replacing his Definition II.A-4. (In a choice-construction testing system, the single value of Gourlay's method definition would be the set of all tests for which this relation holds. Thus the definitions are equivalent.) Call the method *effective* if the relation is recursive. Second, replace his Definition II.B-1 with one that refers to tests T, not to sets of tests:

M is stronger than N, $M \geq N$, iff:
$\forall p \forall s (\exists T (N(T, p, s) \langle \neg(p\ OK_T\ s)) \Rightarrow$
$\forall T' (M(T', p, s) \Rightarrow \neg(p\ OK_{T'}\ s)))$.

That is, if the weaker method has any exposing test, then every test of the stronger method must be exposing. This is precisely what Gourlay calls for in motivating his definition. The choice construction simplified all the substance out of his Definition II.B-1.

With the revised definitions, and the properties of effective methods noted above, we have the following theorem in contrast to Gourlay's II.B-7 and II.B-8:

Theorem:

> All effective, monotonic test methods are incomparable under the ordering ≤.

In particular, the PATH methods (including STATEMENT and BRANCH) and the mutation method Gourlay considers, are incomparable, not ordered as he claims. The

revised definition above captures a kind of "worst-case subsumes," requiring that when one method finds a bug, a stronger method must also. With such a definition, no effective method would subsume any monotonic method.

It is a special case of the Theorem that $\leq$ is not reflexive for an effective, monotonic method; that is, no such M has $M \leq M$. The intuitive content of this special case is that a method cannot be evaluated even relative to itself; two applications of the same method may have different outcomes. This certainly agrees with experience, and underlines the difficulty with empirical comparisons discussed in §2.1.

Perhaps it is worth explicating Gourlay's definition in the same terms as above, so that its intuitive content can be seen. Take our (equivalent) replacement for Definition II.A-4. Then his Definition II.B-1 reads for a choice-construction system:

$$M \geq N \text{ iff:}$$
$$\forall p \forall s (\exists T (M(T, p, s) \langle\ p\ OK_T\ s) \Rightarrow$$
$$\exists T' (N(T', p, s) \langle\ p\ OK_{T'}\ s)).$$

That is, if there is a successful test in the more powerful method, then there must be one in the weaker method. It is more revealing to state this as what should *not* happen: there should not be a misleading test in the more powerful method, but all tests of the weaker method are exposing. That property is certainly desirable, and perhaps an improvement on *subsumes*, but a far cry from the intuitively correct idea that there should not be a misleading test in the more powerful method, but an exposing test in the weaker one.

2.4. Using One Method to Judge Another

It is tempting to compare test methods directly, in a kind of mutual coverage competition. Given (1) a set of programs, (2) several test collections, each satisfying a test criterion for these programs, and (3) analyzers for each criterion, the test collections can be interchanged and submitted to the analyzers. The results have an intuitively pleasing form. For example, one might learn that the data for method R always seems to satisfy method P, but the data for method P usually fails to satisfy R. The conclusion that R is better than P seems inescapable. When one method (say M) is assumed to be the standard (call M the "touchstone" method), then P and R can be compared by seeing to what extent they satisfy M; or, a single method P can be evaluated by the extent to which it satisfies M.

Such comparisons are analogous to validating an electronic instrument by checking its readings against another known to be erratic. They would probably not be made if they could not be automated, but when test data is generated and a coverage tool is available, they can be. Thus mutation in FORTRAN programs is often involved (because such a tool is easily available [2]), and random test data is the most commonly generated.

As experiments the results have meaning only so far as the programs are representative, and the test generations free from bias, as described in §2.1. Furthermore, the simplest experiments (does P satisfy M?) are no more than a sample taken to investigate the *subsumes* ordering. Insofar as many experiments exhibit method-P tests that are also method-M tests and none that are not method-M tests, it is more likely that P subsumes M. The quality of the experiment determines to what extent *subsumes* is actually being captured; however, the difficulties with this ordering presented in §2.2 remain, and are the more fundamental.

The first published use of such method comparisons appears to be Ntafos's paper defining required-element testing [18]. In it he attempts to compare this method with branch testing and a peculiar version of random testing, using mutation as a touchstone. The work suffers from the usual experimental difficulties described in §2.1, because a limited number of small programs were used, and the data for branch- and required-element testing were "minimal" (generated by Ntafos in a way that is not described). Furthermore, there were few cases in which all mutants were killed by any of the methods, so the comparison is technically in terms of the fraction killed, without analyzing which mutants constitute the fraction in each case. For example, one of the programs was (a supposedly corrected version of) the triangle-classification program of [23]. The mutation system generated 286 incorrect mutants, and the three methods (random, branch, and required-element) killed respectively (80%, 91%, and 99%) of these. The implication is that required-element testing is thus demonstrated to be the best of the three methods, using the mutation touchstone.

Aside from the experimental difficulties, comparisons like this can be fundamentally misleading. Part of the reason is that the programs investigated were chosen to be correct. In practice, the whole point of testing is to learn about a program that is probably not correct. Although Ntafos does not give the information, the following is a possible interpretation of his data. Suppose that one of the four mutants not killed by required-element testing is P_{281}, and suppose further that P_{281} was killed by branch testing. (It is easy to construct such a case, even though mutation subsumes branch testing, as indicated in §2.2.) If P_{281} were being tested (instead of the correct triangle program) we have exactly the case of a misleading test for required-element testing and an exposing test for branch testing. The flaw presented relies on not all mutants being killed by the "better" method, but it extends to a more extensive comparison, too. So long as the touchstone method is not perfect in the sense of guaranteeing correctness, programs like P_{281} will exist, and the comparison can mislead us.

What's disturbing about cross-method exchanges is that there may be a hidden correlation between the methods being compared, a correlation which is based on some feature unrelated to actual program quality. Then the

correlation determines the winner, in a misleading way. For example, Budd and Miller [3] suggest the hypothesis that the more times a statement is executed by tests, the more likely they are to expose faults in that statement. They experimented with this idea using mutation as a touchstone, and programs that varied in their degree of iteration. We believe the hypothesis; indeed, repeated statement execution is assurance testing restricted to the control-point part of the state space, as described in §3.2 below. However, the experiment is flawed because there is a direct correlation between mutation and state-space coverage. When the state space is better covered, more mutants should be killed, so the experiment's outcome is predetermined, and it provides no independent support for the hypothesis.

Two special cases of cross-method comparison deserve comment. It can be used to evaluate heuristics for generating data. A heuristic attempts to approximate some method, and that method may then serve to judge the quality of the heuristic. DeMillo and Offutt [5] use this technique to evaluate a constraint-based heuristic for generating (sometimes) mutation-adequate data. Although it may seem unexceptionable to use a parent method to evaluate the quality of an approximation, the technique is suspect because it is based on experimental determination of *subsumes*. It could happen that the heuristic happens to generate mutation-adequate data of a peculiar kind, which is of low utility relative to actual program quality. Other mutation-adequate data, which might be typically found by a human being not using the heuristic, could be much better at finding failures. Then the observed result that the heuristic does achieve a substantial fraction of mutation coverage, is misleading. The reason is the one that has appeared again and again in this section: methods should not be evaluated in their own terms, when those terms cannot be connected to program quality.

The second special case of using one method with another is an investigation using random testing on structural methods. [27] suggests calculating the number of random tests required to attain coverage with a given probability. As a theoretical idea this gets at the difficult relationship between input distribution and state-space coverage (see §3.2 below). As a comparison idea, however, it seems no better than the others considered in this section. The random test may have statistical validity as described in §3.1 below, the more so the larger it is. But there seems no necessary virtue in a method that is or is not resistant to satisfaction by random inputs.

Thus comparison by using methods on one another in general suffers from the sins of both empirical study and *subsumes* analysis.

2.5. Axiomatic Comparison

Elaine Weyuker has attempted to devise axioms for test adequacy criteria [29]. Although there are philosophical differences, an adequacy criterion and a test method are formally identical (a predicate satisfied by a program, specification, and test), so her axioms can be used to compare methods. The comparison of [29] has generated some controversy; [31] suggests that the application to practical methods does them an injustice. Axiomatic methods probably can never produce satisfactory comparisons, because of the unsolvable nature of the correctness problem. Since tests should in some way relate to correctness, but this cannot be realized in general, any axiomatic characterization will necessarily be weak, and if strengthened can only be directed at extraneous features of test methods. Thus comparisons will fall between two stools: either the methods will not be distinguished by the axioms (when they deal with essential features), or they will be distinguished, but only on the basis of inessential features.

However, the point of Weyuker's work is to provide a framework for absolute evaluation, not comparison, and to explicate the relationships between (perhaps less-than-obvious) properties of test methods. Insofar as her axioms expose problems in the way we think about methods, they are a useful framework.

3. PROPOSAL FOR COMPARING TESTING METHODS

Any comparison of testing methods can be criticized as in §2: if the comparison shows one method to be superior, an example can nevertheless be found in which the superior method is misleading and the inferior is not. Therefore it is crucial that the terms of comparison be plausible, that they be founded in properties of programs that go beyond incidental features of test-method definition. If the terms of comparison are unobjectionable, there cannot be unexpected and misleading correlations between trivial properties of methods and their judged relative quality. Program correctness and its correlates such as performance are the appropriate comparison properties, but except in special cases these are not related to any effective testing method [14, Ch. 4]. The only other possibility is to treat a test as a sample of the program's behavior, and to perform statistical analysis of that sample. This is the obvious way to handle the mismatch between the infinity of points that define functional behaviors like correctness and performance, and the finite collections that tests must be.

3.1. Comparison Using Failure-rate Reliability

The first published results that can claim to be more than an anecdotal comparison of methods are those of Duran and Ntafos [6] comparing random and partition testing according to the failure-rate reliability model. Their paper raises questions about its many assumptions, but it is unique in basing a comparison on properties of programs that are of real interest (how likely is a test to excite a failure, for example), and stating the assumptions needed to derive the result.

Partition testing is a class of methods that includes most of those usually called "systematic," in the sense that the

method makes a subdivision of the input space, and requires testing within each subdomain. The archetype example is path testing. The relation between two inputs: "they execute the same path" divides a program's input domain into equivalence classes. Path testing is the method that requires selection of data from each of these classes. Properly speaking, the partition is the relation or the division into classes; however, the classes themselves are loosely referred to as partitions. Not all schemes for subdividing the input are equivalence relations, however. The subdomains may overlap, which can result in test data that simultaneously lies in several (and thus the defining relation fails to be transitive). Statement testing (using the subdivision based on two inputs executing the same statement) and mutation testing (inputs killing the same mutant) are two schemes with overlap. The technical results apply only to partition testing methods with true equivalence classes, without overlap. (The overlap case will be discussed following presentation of the results.)

Duran and Ntafos attempted to characterize partition testing in general, assuming just enough of its properties to allow a comparison with random testing. They took the conventional reliability model in which a program is characterized by a failure rate R. When tests are drawn from the operational distribution, the program is assumed to fail in such a way that the long-term average of the ratio of failed tests to total tests approaches R. This sampling defines (overall) random testing. They further assumed the input space to be partitioned in an arbitrary way, with each subdomain characterized by its own failure rate. Drawing from the same operational distribution, but now forcing a fixed number of points to lie in each partition, defines partition testing. The two schemes can be analytically compared by relating the number of tests in each, the chance that the overall tests's points will fall in the various subdomains, and the overall- to partitioned failure rates. Evidently the overall test should use as many points as the sum of those used in the partitions. To handle the failure-rate relationship, Duran and Ntafos assumed a distribution of partition values. They tried a uniform distribution—the chance that a partition have a failure rate in [0,1] equally likely—and a distribution that corresponds to the partitions being nearly homogeneous—the chance that a partition have a failure rate close to 0 or close to 1 much higher than that the rate take some intermediate value. Finally, the probabilities of the overall random test points falling in each partition were taken from a uniform distribution. The use of distributions for some of the parameters means that experiments are required, each defined by a drawing from the partition failure-rate distribution and the distribution of overall samples among the partitions.

The results showed that partition testing is slightly superior to random testing in the probability of finding a failure, the ratio being about 0.9, and the spread over various experiments indicated by a standard deviation of roughly 0.1 in the ratio.

In replicating and extending the results of [6], Taylor and Hamlet [10] carried the analysis one step farther. They varied the assumptions that characterize partition testing, in the attempt to understand the counterintuitive results that partition testing is not much better than random testing. Their work is thus a comparison of different kinds of partition testing, with conventional reliability theory as the standard.

Because Duran and Ntafos present only two examples of their comparison, Taylor [10] first checked the stability of their results, by varying parameters like the number of partitions, the failure rate, and the number of points per partition. The results were found to be remarkably stable: partition testing is slightly superior, but the superiority cannot be improved by varying these parameters. Taylor [10] then conducted two experiments in which some "small" partitions were given high failure rates and low probability of being hit by overall random tests. These experiments produced significant differences between the methods. Under the same circumstances, he also varied the homogeneity of the "small" partitions. These experiments amount to an investigation of what makes partition testing work, with overall random reliability testing as the standard. They differ from the ones conducted by Duran and Ntafos primarily in that the assumed failure rate is low, a situation in which it is possible to observe variations in the efficacy of partition testing.

The results presented in [10] can be summarized as follows:

1) If partitions are not perfectly homogeneous, then the degree of homogeneity is not very important.
2) Partition testing is improved when one or more partitions have a substantially higher probability of failure than that overall.
3) Finer partitions are disadvantageous if their failure rates are uniform.

Overlap must also be considered in comparing methods. In statement testing, for example, the natural subdomains are composed of inputs that cause execution of the same statement; these subdomains are not disjoint. (In particular, since all inputs execute the first statement, one subdomain is the entire input space.) A true partition can be formed by intersection of the natural subdomains, however. Points selected from overlapping natural subdomains can be thought of as selected from the true partition, but with increased sample density in the overlap. Insofar as the failure rate is low in these parts, a method will suffer relative to overall random testing; as it is high, the method will fare better. If there is no reason to believe that the overlapping regions are prone to failure, then a method with overlap will come out a bit worse in comparisons that it would appear, because its true partitions are a disadvantageous refinement.

We give a comparison in which the theory suggests a clear preference: specification-based *vs.* design-based testing. It is the virtue of specification-based tests that they can be devised early in the development cycle. Before code is written, or even designed, the software requirements/specifications can be analyzed, and the potential inputs broken into equivalence classes by required functionality. A good example of this kind of testing is presented in [20], where the system to be tested has a number of commands, each with a number of parameters, and the partitions are defined first by command, then refined by parameter value. "Design-based" tests (a method just invented for this comparison, although variants are in practical use) can also be devised before code is written. When the program has been divided into modules of some kind, and its overall structure defined in terms of their interfaces and dependencies, partitions can be defined according to a kind of large-grain "statement" coverage. Two inputs are in the same subdomain if they invoke the same module. (This is an overlapping subdivision.)

Design-based testing is the better method according to the reliability-theory comparison. Results 2) and 3) above apply. In specification-based partitions, there is less reason to believe that the failure rates will differ than in design-based partitions. The designers' subjective estimates of the difficulty of implementing each module, and the ability of the programming staff assigned, are very good indications of where to look for failures in the design-based partition. Furthermore, specification-based partitions strive for coverage of functionality, which is evidently a refining criterion not related to failure rate. The design-based partitions on the other hand can be refined as implementation proceeds, using improved information. Code metrics may point to fault-prone modules or parts thereof, and most important, programmers can make subjective estimates of where there might be trouble because the coding task was perceived to be of uncertain quality. Thus the theory predicts that testing effort should be devoted to design-based tests rather than specification-based ones.

We end this section with a collection of assertions about practical methods and their comparison, which the theory supports for the reason given in parentheses. A number of these are counterintuitive and suggest further analysis and experimentation:

a) It is a mistake to refine a partition just to make its tests more like each other. (The finer partitions will be more homogeneous but will likely have uniform failure rates.)

b) Combining testing methods to create intersections of partitions, for example by refining a structural partition with a specification-based one as suggested in [25], will not be an improvement over either method used alone. (There is no reason to believe that the finer partitions include any with high failure rates, and a high-failure partition may be diluted).

c) Dataflow testing methods, particularly those that involve more of the program context [15], should be superior to full path testing. (The dataflow partitions are failure prone because they concentrate on difficult programming situations; full path testing refines the dataflow partitions without any evident increase in the expected failure rate.)

d) Specification-based testing that takes no account of potential difficulties in design and implementation is not much good. (See the comparison with design-based testing immediately above.) The "cleanroom" project [26] used specification-based testing, and so could be improved with design-based testing.

e) Any tests that have been used in walkthroughs or as typical cases for design and implementation, are useless for later finding failures. (The partitions they define are likely to have low failure rates because they have been thought through in the development process.)

f) Special-values testing is valuable because the special values represent cases that are likely to fail. Hence a special value that fits item d) or e) above is not so useful. (To be valuable, the singleton special-value partition should be an extreme case of high failure probability.)

3.2. Comparison Using Defect-rate Assurance

Statistical reliability is a better comparative testing measure than the subsumes relation considered in §2.2. It is universally applicable, and its intended meaning—how likely the software is to fail in normal usage—seems clearly related to quality. However, the plausibility of conventional reliability for software has been called into question. The flaws in software reliability theory are:

R) It assumes a long-term average failure rate, for which there is little supporting evidence. Rather, it is observed that software failures come in bursts with stable periods between. This phenomenon is probably related to the usage distribution (D) below.

D) It assumes an operational distribution of input values, presumed to be the one describing normal usage of the software, which may have no existence in reality. Usage may have so many "modes" (for example, novice/expert users) that no distribution is typical. More important, the lack of a distribution exposes a larger conceptual flaw: software quality does not intuitively depend on normal usage. We judge quality by how well the software behaves *under unusual circumstances.*

I) It assumes independent samples drawn from the operational distribution. Intuitively, making unbiased input choices does not guarantee increasing confidence in the results, because it can happen that the program's internal behavior is nearly the same for quite different inputs. Because software is deterministic no assurance is gained by repeating tests in this way.

W) Whatever one may think of the theory's assumptions, the derived results are not intuitively correct. For example, the size and complexity of programs, specifications, and their input domains, do not enter in any way.

The operational distribution (D) above is at the heart of the difficulty with the conventional theory. The results must be expressed in the form, "...the probability of failure *in normal usage* is...". Thus the comparison suggested in §3.1 is also flawed—methods are compared only in their ability to expose a failure under normal use. Insofar as this basis misses the mark, the comparison does too.

It is usual to express software quality in a way quite different from the reliability view. For instance, in touting the quality of the space-shuttle flight software, it is said to have less than 0.1 defect per thousand lines of code. Such a measure is intrinsically related to quality, because its expression is independent of usage, and because it is directly related to the process that produces software, and the mistakes that might be made in that process. A sampling theory can be constructed [9] based on defect rate. This theory will be called an *assurance* theory in contrast to the *reliability* theory of §3.1. (Parnas [21] calls software *trustworthy* when it meets an assurance test in this sense. He argues that trustworthiness is the only appropriate measure of quality, but that it requires an impractical number of tests to guarantee.)

In the the assurance theory, assumption (R) is replaced by the assumption of a defect rate. Assumptions (D) and (I) are replaced as follows:

D´) It assumes that textual faults are uniformly distributed over the state space of the program, which consists of all control points, and all values of variables at those points. That is, a fault is defined as a state-space value that leads to a failure. This is an improvement on (D) because the distribution is fixed, and because it is plausible that programmers make mistakes at any such point with equal probability. The objection to this view is that the notion of "fault" is not well defined [22, 17].

I´) To sample for defects means to sample the state space independently. It is precisely because samples from the input domain do not penetrate uniformly to the state space that (I) is called into question. However, tests are necessarily conducted through the input domain, and to require that they obey some unknown distribution there is impractical.

The assurance theory argues that (D´) is often sensible in practice, and that (I´) is correct however unattainable. The assurance theory does improve on (W): program size and cardinality of the input domain enter appropriately. The number of tests N required to guarantee a given defect rate is roughly proportional to the program size M and domain size D, and inversely proportional to the number of faults per line f. For 99% confidence and large $\frac{D}{f}$, the proportionality constant is about 10^{-4} fault/line^2: $N \approx 10^{-4} \frac{MD}{f}$. For example, a 50K-line program with a million-point domain requires $N \approx 5 \times 10^{10}$ tests to guarantee less than 0.1 fault per thousand lines.

The assurance theory does not suggest a practical testing method, both because the independent samples it requires cannot be identified in the input domain, and because there are far too many tests required. However, if the theory is correct, it does provide a standard for comparing other methods. Hamlet [10] attempts such a comparison. Assuming that tests are selected so that the state space is uniformly sampled, it is shown that partition testing is far *inferior* to overall random testing, and that the disadvantage is magnified by more partitions or by lower fault rates.

Again we must examine the theory and our intuition about partition testing, to find the assumptions that influence these results. Tests in general do not directly sample the state space as the theory assumes, and intuitively, the selection of inputs that excite varied program states is most difficult when the whole domain is used. Partitions that reflect an input distribution into the state space with little distortion, would outperform any overall testing scheme (because the latter would produce such bad coverage of the state space). This observation has implications for the level at which assurance testing should be done.

Distortion in state-space sampling occurs because program control points cannot be reached with all values of the variables there. Control predicates are part of the reason for restricted state values; assignments to part of the state are the other. If specifications were available at the statement level, ideal assurance testing could be conducted by first determining the range of possible variable values at a statement, then sampling with a uniform distribution over that range. Howden has suggested "functional" testing in which each code unit is individually tested against a local specification [14, Ch. 5]; an assurance test could be conducted for these units. A system "designed for testability" would strive for units with their own specifications, whose input spaces provide a direct mirror of their state spaces. Uniformly distributed random tests for such units would constitute an ideal assurance test. Even in the absence of low-level functional specifications and designed-in state-space access, assurance testing should probably be unit testing rather than integration testing [12]. At the integration level, reliability testing (with partitions defined by interface parameters!) may be more appropriate.

We apply the assurance theory to two (overlap) partition methods, branch testing and mutation testing.

The natural input-space subdomains for branch testing are those in which inputs execute the same conditional with the same branch outcome. We ask how well tests chosen from these subdomains cover the state space. Branch coverage involves at least two points in the state space of each

conditional expression; this means two points in the state space of the basic block preceding that conditional. However, the state at the location immediately following the branch is only required to assume a single value. There is no necessary state-space connection between the subdomains; it can happen that together the subdomains force no more than the minimum coverage of two states for each conditional. This situation was referred to in §2.2 as "trivial" satisfaction of a method. Intuitively, the letter of the method is satisfied, but by data that has little chance of exposing a failure. In any kind of path testing, the program is forced down the path, but with values of the variables that do not necessarily explore a large state space on that path.

The natural subdomains for mutation testing contain those inputs that kill the same mutant. Intuitively, some mutants can be killed only by unusual states, and different mutants may require different states. Thus to kill all mutants at one control point means that the state there must be extensively explored. (It might be made a design criterion for mutation operators that they force a wide exploration of the state space.) Failures that arise only from a combination of faults defeat mutation, because the mutants at one fault point do not require the state to vary quite enough; it is only when another fault point is reached that the earlier variation can be seen to be too narrow. Thus mutation should be a better assurance method than branch testing, but it falls short of perfection.

This discussion suggests a tool to measure the software assurance to be expected from test data, whatever the test origin. The program under test can be instrumented, in the same way that profiles are obtained, to show the state-space coverage. When the coverage is poor, it may be that the test could be improved, or perhaps the state space is constrained in some peculiar way. The tester must make the decision, since the problem is unsolvable in general. However, when two tests are compared, the better one from the assurance point of view is the one with the better state-space coverage.

The notion of state-space coverage suggests a modification of the *subsumes* relation that may better correspond with reality: one method is better than another if its state-space coverage is better. Precisely, method A is better than method B iff for all programs the states covered by any B test are necessarily covered by all A tests. This "state-space subsumes" has the property that any failure exposed by a poorer method must also be exposed by a better one, since the state responsible must be covered by the latter. Thus it does not suffer from the deficiency described in §2.2-2.3. However, only methods that are not monotonic could be poorer, unless the better one were perfect at exposing failures, so the analytic application of this idea is not promising. On the other hand, using the tool proposed in the previous paragraph, an experiment could be devised to investigate state-space subsumption. If the defect-rate assurance theory is correct, such experiments would be measuring real efficacy of the methods, although of course the results are still subject to the difficulties described in §2.1.

4. SUMMARY

We have argued that a plausible comparison of testing methods must be based on reliability or assurance sampling as the standard. Reliability theory suggests partition methods that concentrate failures, at the expense of partition coverage and homogeneity. Assurance theory indicates that testing must be done at or below the unit level in order to control the state-space coverage. The qualitative comparisons presented here suggest further investigation of methods using these standards.

Acknowledgements

Elaine Weyuker noticed that "partition" methods are usually not true partitions. Bruce Gifford suggested examination of Gourlay's work in this context, and he recognized that the Theorem of §2.3 applies to a single method.

References

1. V. Basili and R. Selby, Comparing the effectiveness of software testing strategies, *IEEE Trans. Software Eng.* SE-13 (December, 1987), 1278-1296.
2. T. A. Budd, The portable mutation testing suite, TR 83-8, Department of Computer Science, University of Arizona, March, 1983.
3. T. A. Budd and W. Miller, Testing numerical software, TR 83-18, Department of Computer Science, University of Arizona, November, 1983.
4. R. DeMillo, R. Lipton, and F. Sayward, Hints on test data selection: help for the practicing programmer, *Computer* 11 (April, 1978), 34-43.
5. R. A. DeMillo and A. Jefferson Offutt VI, Experimental results of automatically generated adequate test sets, *Proceeding 6th Pacific Northwest Software Quality Conference*, Portland, OR, September, 1988, 210-232.
6. J. Duran and S. Ntafos, An evaluation of random testing, *IEEE Trans. Software Eng.* SE-10 (July, 1984), 438-444.
7. J. Gourlay, A mathematical framework for the investigation of testing, *IEEE Trans. Software Eng.* SE-9 (November, 1983), 786-709.
8. R. Hamlet, Testing programs with the aid of a compiler, *IEEE Trans. on Software Eng.* SE-3 (July, 1977), 279-290.
9. R. Hamlet, Probable correctness theory, *Info. Proc. Letters* 25 (April, 1987), 17-25.
10. R. Hamlet and R. Taylor, Partition testing does not inspire confidence, *Proceedings Second Workshop on Software Testing, Verification,* and Analysis, Banff,

Canada, July, 1988, 206-215.

11. R. Hamlet, Editor's introduction, special section on software testing, *CACM* 31 (June, 1988), 662-667.
12. R. Hamlet, Unit testing for software assurance, *Proceedings COMPASS 89*, Washington, DC, June, 1989, 42-48.
13. W. Howden, Reliability of the path analysis testing strategy, *IEEE Trans. Software Eng.* SE-2(1976), 208-215.
14. W. Howden, *Functional Program Testing and Analysis*, McGraw-Hill, 1987.
15. J. Laski and B. Korel, A data flow oriented program testing strategy, *IEEE Trans. Software Eng.* SE-9 (May, 1983), 347-354.
16. L. Lauterbach and W. Randall, Experimental evaluation of six test techniques, *Proceedings COMPASS 89*, Washington, DC, June, 1989, 36-41.
17. J. D. Musa, "Qualitytime" column, Faults, failures, and a metrics revolution, *IEEE Software*, March, 1989, 85,91.
18. S. C. Ntafos, An evaluation of required element testing strategies, *Proc. 7th Int. Conf. on Software Engineering*, Orlando, FL, 1984, 250-256.
19. S. C. Ntafos, A comparison of some structural testing strategies, *IEEE Trans. Software Eng.* SE-14 (June, 1988), 868-874.
20. T. J. Ostrand and M. Balcer, The category-partition method for specifying and generating functional tests, *CACM* 31 (June, 1988), 676-687.
21. D. L. Parnas, A. van Schouwn, and S. Kwan, Evaluation standards for safety critical software, TR 88-220, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada.
22. D. Parnas, personal communication.
23. C. V. Ramamoorthy, S. F. Ho, and W. T. Chen, On the automated generation of program test data, *IEEE Trans. Software. Eng.* SE-2 (Dec., 1976), 293-300.
24. S. Rapps and E.Weyuker, Selecting software test data using data flow information, *IEEE Trans. Software Eng.* SE-11 (April, 1985), 367-375.
25. D. Richardson and L. Clarke, A partition analysis method to increase program reliability, *Proc. 5th Int. Conf. on Software Engineering*, San Diego, 1981, 244-253.
26. R. W. Selby, V. Basili, F. Baker, Cleanroom software development: an empirical evaluation, *IEEE Trans. Software Eng.* SE-13 (Sept., 1987), 1027-1038.
27. P. Thévenod-Fosse, Statistical validation by means of statistical testing, Dependable Computing for Critical Applications, Santa Barbara, CA, August, 1989.
28. M. Weiser, J. Gannon, and P. McMullin, Comparison of structural test coverage metrics, *IEEE Software* (March, 1985),80-85.
29. E. J. Weyuker, Axiomatizing software test data adequacy, *IEEE Trans. Software Eng.* SE-12 (December, 1986), 1128-1138.
30. S. J. Zeil, The EQUATE testing strategy, *Proceedings Workshop on Software Testing*, Banff, Canada, July, 1986, 142-151.
31. S. H. Zweben and J. S. Gourlay, On the adequacy of Weyuker's test data adequacy axioms, *IEEE Trans. Software Eng.* SE-15 (April, 1989), 496-500.