ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 6, Issue 2, pp: (96-106), Month: April - June 2018, Available at: www.researchpublish.com

# Diferentation of the Software Test Techniques

Seyfali Mahini

Islamic Azad University, Khoy branch, Khoy, Iran

***Abstract:*** **The comparison of software test methods only makes sense if the sound theory relates the properties in comparison to the actual software quality. Existing comparisons typically use anecdotal fundamentals without the need for quality and compare methods based on technical terms that define the methods themselves. In the worst-case work, a method whose effectiveness is unknown is used as the standard for evaluating other methods! Causal Testing, as a method that can relate to quality, both in the conventional sense of statistical reliability and the stricter sense of Software Assurance, offers the possibility for a valid comparison.**

***Keywords:*** **software test, comparison of test methods, specification-based tests, Automatic test generation, white box test, black box test, partition testing.**

## 1. INTRODUCTION

It is well known that testing methods are difficult to compare because most testing have no theoretical basis. For example, it seems useful to test until every statement in a program has been executed. It would be stupid not to do this because a statement never executed could contain an arbitrarily bad mistake. But what do the testers know after they reach the statement coverage? It's easy to show that statement coverage can be achieved through many different sets of tests, some of which detect errors and others do not. How then can the value of statement tests be evaluated? As long as the relationship between the basis of a method (in the example that covers the statements) and the properties of the software (here the occurrence of errors) remains inaccurate, meaningful comparisons between methods, even comparisons between two applications of the same method can not be made. There are a number of seemingly reasonable analytical comparisons that can be made between test methods that are less useful in the analysis than they seem. The comparisons are precise, but they use parameters that have nothing to do with the actual effectiveness of the methods and can therefore be fundamentally misleading. For example The "subsumes" ordering is of this type. considerable Empirical comparison of methods seems the apparent solution to all these difficulties. Its two drawbacks are that it depends heavily on the particular programs and environments selected for the experiment, and often on the relative ability of human subjects participating in the assessment. once Again, the real mistake lies in our lack of theory: because we do not know how a method is related to the software properties of interest, we do not know how to control the important variables. Section 2 critically reviews past attempts to compare the test methods to show where they are in principle flawed. In Section 3, we propose a new basis for comparison.

## 2. PRIOR COMPARISON OF TESTING METHODS

In cases where testing resources are scarce or testing needs to be of genuine software quality, the practitioner is all too aware that methods need to be compared. Of the opposite side, the designer of a new testing method is called upon to explain it by comparison to existing methods. If the call for comparison is loud enough, comparisons are preferred, but without theoretical foundations you can not trust them.

### 2.1 EMPIRICAL COMPARISONS:

Empirical studies comparing the test methods must compete with two potentially unacceptable factors:

(1) (*programs*) A certain collection of programs needs to be used - it may be too small or too curious to trust the results.

(2) (*test data*) For each method, certain test data must be created - the data may have good or bad characteristic that are not related to the method.

The use of humans to generate test data is one way in which (2) can occur. Because the test generation requires the use of information about the program or its specification, a human can use that information in a way, which falsify the test. It is certainly unfair to assign a quality to the test method that actually stay to the human tester. An experimenter may try to control this "nose-rubbing effect" [11], but since human abilities vary widely and are poorly understood, control is not perfect.

The program selection (1) for an experiment can be influenced both by program attributes and by the type of errors present. When it comes to genuine programs that have been developed in practice, these difficulties merge; However, if the errors are set or the programs originated in a training environment, the distortion can be twofold. Because the program characterization is not well understood, no quantitative measurements of their properties are available. It certainly seems safer to trust the results of a study in which researchers have worked hard to control the environment [1], but an unknown factor can thwart the best of human endeavors if understanding is lacking.

Factor (2) can be treated in several ways:

a) *Worst-case analysis*. By examining all the test sets that meet a method, you can accurately predict the worst that this method can achieve.

The main disadvantage of worst-case analysis is that it is cumbersome and can not normally be automated. There have been few followers of Howden's careful study of path coverage in a number of textbook programs [13]. In addition, as we show in § 2.3, the inclusion of the worst-case analysis as a comparison technique makes all methods technically incomparable.

b) *Random selection*. If a method involves randomly dragging test points from a distribution, then no distortion should be 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. The random selection of test sets can also be used for methods that themselves have no probabilistic aspect. If the test sets have an even distribution, the sample will produce an average result. Although it does not give a true average case, using the random selection to meet a coverage method can control the nasal-rubbing effect.

c) *Automatic test generation*. Some methods can be used with algorithms or heuristics that generate data satisfying the method. The choice of programs can lead to hidden distortions on which, for example, a heuristic works. It must also be remembered that the results apply only to this close version of the method that uses the generation algorithm.

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

**2.2 THE SUBSUMES ORDERING:**

The majority of accurate published comparisons between test methods use the following order:

Method *A* subsumes method b when a test that satisfies *A* necessarily satisfies *b*.

For example, a method in which every test result is required to be correct includes one in which only 80% of the results must be correct. The inclusion is rigorous because one test that barely meets the 80% method can not satisfy the other.

Unfortunately, only a few common-ground methods can be compared sing subsums. Forms of path coverage can be compared, and these fall into a quite simple hierarchy [24, 19]. But many methods are incomparable - the other does not subsume [28]. Banal differences can make methods technically incomparable, as in the three mutation variants [8, 4, 30]. However, the subsumption relationship is more serious than its limited applicability. All algorithmic test methods are necessarily imperfect, since none can guarantee the correctness of the programs in general [14, Ch. 4]. When one imperfect method subsumes another, the relationship does not reflect the ability to disclose mistakes or ensure quality. It may happen that *A* subsumes *b*, but *b* actually addresses these real properties better. Intuitively, the problem with subsumes is that it rates methods in their own terms, regardless of what is really of interest. For example, consider

statement testing as Method *A*, and let *b* be random tests that have no requirement but that the test set is not empty and the results are 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 := 2 * sqr(x); end;

Suppose this program is intended to compute the function of t : →2 * √abs(t).

The Pascal Square function is used here inadvertently instead of the square root.

The test input set {1, 0, -1} reaches the instruction coverage, but does not reveal an error, while {3} is arbitrary and does not cover all instructions, but reveals an error.

Such examples are not really pathological. Since there is no necessary connection between covering statements and the finding of errors, the consideration of the former diverts attention from the latter. In this case, the random tester ignored the absolute value part of the program and focused on the square (root), with good results. Similarly, the path-flow variants of the path test [24, 15] may prove superior to the full path testing , since they focus attention on important cases that could be trivially treated in the chaos of all paths. The idea of the trivial satisfaction of methods is further investigated in §3.2.

When we include the human process of selecting test points or using mechanical means to generate test data, the subsumption order more clearly indicates their errors. A method can be so difficult to understand that in generating test points it is trivial to support decisions. Another method, which is easier to use, can lead to more representative data. Nothing prevents the poorer, more complex method from subsuming the better, simpler one. Coverage tools aggravate the problem rather than mitigating it: when a person tries to fill gaps in a hard-to-cover coverage method, as required by a test tool, trivial data naturally comes to mind, and the subsequent release of the tool fails because of the poor test quality.

**2.3. GOURLAY'S COMPARISON:**

In his excellent theoretical framework for testing [7] Gourlay defines a partial order $M \geq N$ between the methods *M* and *N*, which claims to capture the intuitive idea that one method is stronger than another. Its definition gives some of the same results as subsumes (for example, that BRANCH ≥ STATEMENT but STATEMENT ! ≥ BRANCH for the two methods with these names). We agree exactly with Gourlay's intuition about method comparison as motivation for its definition II.B-1). He explains that the method *M*, which is stronger than *N*, should mean that if *N* reveals an error, *M* must do likewise. However, his Theorem II.B-7 says that if one method subsumes another, its order holds. The results for BRANCH and STATEMENT are in line with Theorem II.B-7, but seem to conflict with the criticism of the subsumes in the previous section.

Since Gourlay's theoretical framework is good, it can be used to explain the problem and show that Definition II.B-1 does not capture his intuition. By better definition, we show that no algorithmic method is comparable to any other method. This formal result accurately captures the general case against the presented subsumes in §2.2.

It is Gourlay's "construction of choice" that causes the difficulty. In a test system that uses this construction, one method is the same as a collection of different tests, but the method is *OK* if a test in the collection is successful. For example, testing instructions for a particular program and specification is a method with many potential tests that execute all program statements, but some can be executed successfully, and some can detect an error. Gourlay will call Statement Testing *OK* if at least one of the tests is successful. It 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 the method *Q*. But for a selection system, Gourlay's success is not what we intuitively believe is success.

Referring to the discussion of §2.2, there is an erroneous comparison of the methods from the existence of multiple tests that satisfy every method, of which only a few detect current errors. For a wrong program, call a test that is successful and misleading. If a method is effective, that is, if it allows a mechanical decision as to whether a triplet (*T*, *p*, *s*) consisting of test, program, and specification is satisfactory, then this is a consequence of the undecidability of the program equivalence problem. For this method, there are misleading tests. At the same time, if a program is not correct, then some test shows this fact, and because most methods are monotone (Weyuker [29]), any procedure for including such a test can be

performed. Call a test that finds an exposure test this way. An exposure test and a misleading test are enough to intuitively refute a comparison between methods.

Suppose that method *A* is "better" than *b*. Let $t_A$ be a misleading test that satisfies method *A* and $t_b$ an exposure test that satisfies *b*. Then in fact *A* is no better than *b* using Gourlay's intuitive definition. However, reversing the relationship has exactly the same argument, so the methods can not be compared. However, reversing the relationship has exactly the same argument, so the methods can not be compared.

We can grasp this intuitively correct property of methods in Gourlay's theory by changing its definitions. First, define a test method as a relationship to tests, programs, and specifications, replacing its definition II.A-4. Call the method if the relationship is recursive. Second, replace his definition II.B-1 with one referring to tests *T*, not to test sets:

$M$ is stronger than $N$, $M \geq N$, iff:

for all $p$ for all $s$ (there is $T(\ N(T, p, s)\ )\ \neg(p\ OK_T\ s)) \rightarrow$

for all $T'(M(T', p, s) \rightarrow \neg(p\ OK\ T'(\ s)))$.

That is, if the weaker method has an exposure test, each test must expose the stronger method. This is exactly what Gourlay demands to motivate his definition. The construction of the election simplified the whole content of its definition II.B-1.

With the revised definitions and properties of the above-mentioned effective methods, we have the following theorem, unlike Gourlays II.B-7 and II.B-8:

**THEOREM**:

Alleffective monotone test methods are incomparable below the ordering ≤.

Especially the PATH methods (including STATEMENT and BRANCH) and the mutation method that Gourlay considers are incomparable and not arranged as he claims. The revised definition above captures some sort of "worst case subsumes" and requires that if a method finds an error, a stronger method must do so. With such a definition, no effective method would subsume any monotonic method With such a definition, no effective method would subsume any monotonic method.

It is a special case of the theorem that ≤ is not reflexive for an effective, monotonous method; that is, no such *M* has $M \leq M$ . The intuitive content of this special case is that a method can not be evaluated relative to itself; two applications of the same method can have different results. This certainly agrees with the experience and underlines the difficulties with empirical comparisons discussed in § 2.1.

Perhaps it is important to explain Gourlay's definition in the same words as above, to make the intuitive content visible. Take our (equivalent) replacement for Definition II.A-4. Then its definition II.B-1 reads for a selection construction system:

$M \geq N$ iff:

for all $p$ for all $s$(there is $T(M(T, p, s)\ (\ p\ OK_T\ s) \rightarrow$

there is $T'(N(T',\ p, s)\ (\ p\ OK_{T'}\ s))$.

In other words, if there is a successful test in the more efficient method, then there must be one in the weaker method. It is more instructive to call this what should not happen: It should not be a misleading test in the more powerful method, but all tests of the weaker method will be exposed. This feature is certainly desirable and perhaps an improvement of subsumems, but far from the intuitively correct idea that there should be no misleading test on the more pwerful method, but on the weaker an exposure test.

**2.4. USE ONE METHOD TO JUDGE ANOTHER:**

It is tempting to compare the test methods directly in a kind of mutual competition. If there is (1) a set of programs, (2) several test collections, each meeting a test criterion for those programs, and (3) analyzers for each criterion, the test collections can be exchanged and presented to the analyzers. The results have an intuitively appealing form. For example, one could learn that the data for method R always seems to satisfy method P, but the data for method P is usually not

sufficient to satisfy R. The conclusion that R is better than P seems inevitable. Assuming that a method (such as M) is the standard (call M the "touchstone" method), then P and R can be compared by seeing to what degree 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 unpredictable. They probably would not be made if they could not be automated, but they can be when test data is generated and a cover tool is available, they can be. Therefore, a mutation is often involved in FORTRAN programs (because such a tool is readily available [2]), and random test data is most frequently generated.

As an experiment, the results have meaning only insofar as the programs are representative and the test generations are free of distortions as described in § 2.1. Besides, the simplest experiments (satisfying P M?) Are not more than a taken to examine the subsumption ordering. In this regard many experiments show method P tests, which 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 are actually captured; the difficulties with this order, which are presented in §2.2, however, persist and are more fundamental.

The first published use of such method comparisons seems to be Ntafos's paper, which defines the testing of required elements [18]. In it he tries to compare this method with industry tests and a special version of samples, using mutation as a touchstone. The work suffers from the usual experimental difficulties described in § 2.1. A limited number of small programs were used, and the data for testing branches and required elements was "minimal" (generated by Ntafos in a way that not described). In addition, there have been few instances where all of the mutants were killed by any of the methods, so that the comparison is made technically with respect to the killed fraction without analyzing which mutants constitute each fraction. For example, one of the programs (an allegedly corrected version of) was the triangulation program [2]. The mutation system produced 286 incorrect mutants and the three methods (random, branching and required element) killed each (80%, 91% and 99%) of these. The implication is that testing required elements is thus the best of the three methods using the mutation test block.

Apart from the experimental difficulties, comparisons like these can be fundamentally misleading. Part of the reason is that the programs studied were chosen correctly. Although Ntafos does not give the information, the following is one possible interpretation of his data. Suppose that one of the four mutants not killed by testing the required elements is $P_{281}$, and let us further assume that $P_{281}$ was killed by branching tests. If $P_{281}$ were tested (instead of the correct triangle program), we have the exact case of a misleading test for testing the required elements and a suspension test for testing the branches. The presented error is based on the fact that not all mutants are killed by the "better" method, but extends to a more comprehensive comparison. As long as the test method is not perfect to guarantee correctness, programs like $P_{281}$ will exist, and the comparison may mislead us.

What is disturbing when exchanging across several methods is the fact that there is a hidden correlation between the methods to be compared based on a feature that is not related to actual program quality. Then the correlation misleadingly determines the winner. For example, Budd and Miller [3] hypothesize that the more often a statement is made through testing, the more likely it is that it will reveal errors in that statement. They experimented with this idea using mutation as a touchstone and programs that varied in their iteration. We believe in the hypothesis; In fact, the repeated execution of assessments is limited to the control point portion 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. If the state space is better covered, more mutants should be killed, so that the result of the experiment is predetermined and does not provide independent support for the hypothesis.

Two special cases of method comparison deserve a comment. It can be used to evaluate heuristics for generating data. A heuristic tries to approximate a method, and this method can then be used 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 seems impeccable to use a parent method to evaluate the quality of an approximation, the technique is suspect because it is based on the experimental determination of subsumes. It could happen that the heuristic randomly generates mutation-adequate data of a special kind, which is of little use compared to the actual program quality. Other mutation-appropriate data typically found by a human who does not use the heuristic could find bugs better. Then the observed result that the heuristic achieves a substantial portion of the mutation coverage is misleading. The reason is the one that has repeatedly cropped up in this section: methods should not be evaluated in their own terms if these terms can not be linked to program quality.

The second special case, where one method is used with another method, is a study that uses random sampling of structural methods. [27] proposes to calculate the number of samples required to achieve coverage with a certain probability. As a theoretical idea, this is seen in the difficult relationship between the input distribution and the state space coverage (see §3.2 below). However, as a comparative idea, it does not seem any better than the others considered in this section. The random test can, as in Section 3.1. described to have statistical validity the bigger it is. But there seems to be no necessary virtue in a method that resists satisfaction by accidental inputs.

Therefore, the comparison, applying methods to each other, suffers generally from the sins of both empirical and subsumed analysis.

### 2.5. AXIOMATIC COMPARASION:

Elaine Weyuker has attempted to develop axioms for test adequacy criteria [29]. Although there are philosophical differences, a reasonableness criterion and a test method are formally identical (a predicate is satisfied by a program, a specification, and a test), so that their axioms can be used to compare methods. The comparison of [29] has created some controversy; [31] suggests that the application of practical methods wrong them. Axiomatic methods may probably never provide satisfactory comparisons because of the unsolvable nature of the correctness problem. Since tests should be somewhat correct, this can not be done in general. Any axiomatic characterization will necessarily be weak, and if it can only be strengthened it is directed to external characteristics of test methods. So comparisons between two chairs will fall: either the methods are not distinguished by the axioms, or they are distinguished, but only on the basis of immaterial characteristics.

However, Weyuker's work aims to provide a framework for absolute evaluation, not comparison, and to explain the relationships between characteristics of testing methods. Inasmuch as their axioms pose problems, as we think about methods, they are a useful framework.

## 3. SUGGESTION TO COMPARE THE TEST METHODE

A comparison of the test methods can be criticized as in § 2: If the comparison shows a method superior, an example can still be found in which the superior method is misleading and the inferior one is not. Therefore, it is important that the comparison conditions are plausible, that they are based on the properties of programs that go beyond the random features of the test method. If the comparison conditions are not objectionable, it can not come unexpectedly and misleading correlations between trivial properties of methods and their assessed relative quality. Program correctness and its correlates such as performance are appropriate comparative properties, but except in special cases, these refer to no effective test method [14, Ch. 4]. The only other option is to treat a test as a sample of the program's behavior and perform a statistical analysis of that sample. This is the obvious way to handle the discrepancy between the infinity of points that define functional behaviors such as correctness and performance, and the finite collections that need to be tested.

### 3.1. COMPARISON WITH FAILURE RATE RELIABILITY:

The first published results, which can be more than an anecdotal method comparison, are those of Duran and Ntafos, [6] who compare random and partition tests according to the error rate reliability model. Her work raises questions about her many assumptions, but she is unique in that she compiles a comparison of properties of programs of real interest and the assumptions needed to derive the result.

Partition testing is a class of methods that includes most of those that are usually called "systematic" in the sense that the method performs partitioning of the input space and requires testing within each subdomain. The archetype example is Path Test. The relationship between two inputs: "They execute the same path" divides the input domain into a program into equivalence classes. Path testing is the method that requires selection of data from each of these classes. Actually, the partition is the relationship or division into classes; however, the classes themselves are loosely called partitions. However, not all schemes for subdividing the input are equivalence relations. The subdomains may overlap, which can result in test data that simultaneously lies in several. Statement tests (using the subdivision based on two inputs that execute the same statement) and mutation tests (inputs that kill the same mutant) are two overlapping schemes. The technical results apply only to partition testing methods with true equivalence classes without overlap.

Duran and Ntafos tried to characterize partition tests in general by assuming only enough properties to allow comparison with random tests. They took the conventional reliability model in which a program is characterized by an error rate *R*. When tests are taken from the operational distribution, it is assumed that the program fails such that the long-term average

of the ratio of failed tests to total tests approaches *R*. This sample defines (in total) random tests. They also assumed that the input area should be partitioned in any manner, with each subdomain being characterized by its own error rate. If you assume the same distribution of operations but now set a fixed number of points in each partition, the partition test is defined. The two schemes can be analytically compared by relating the number of tests in each individual, the likelihood that the total test points will fall in the various subdomains, and the total to partitioned error rates. Obviously, the total test should use as many points as the sum of the points used in the partitions. To handle the error rate relationship, Duran and Ntafos assumed a distribution of partition values. They tried a uniform distribution - the likelihood that a partition will have an error rate in [0,1] is equally likely - and a distribution that is nearly homogeneous to the partitions - the probability that a partition will have an error rate close to 0 or close to 1 much higher than the rate take an intermediate value. Finally, the probabilities of all random test points that fall into 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 plot of the partition error rate distribution and the distribution of the total samples among the partitions.

The results showed that partition tests are slightly superior in the likelihood of finding an error, the ratio being about 0.9, and the variance across different experiments is indicated by a standard deviation of about 0.1 in ratio.

In reviewing and extending the findings of [6], Taylor and Hamlet [10] took the analysis a step further. They varied the assumptions that characterize partition tests to understand the preposterous results that partition tests are not much better than random tests. Their work is thus a comparison of different types of partition tests, with traditional reliability theory being the standard.

Since Duran and Ntafos only present two examples of their comparison, Taylor [10] first checked the stability of their results by varying parameters such as the number of partitions, the error rate, and the number of points per partition. The results were remarkably stable: partition tests are slightly superior, but the superiority can not be improved. Taylor [10] then performed two experiments involving a few "small" partitions. High error rates and low probability of being hit by general random tests were reported. These experiments revealed significant differences between the methods. Under the same circumstances, he also varied the homogeneity of the "small" partitions. These experiments lead to an examination of what allows partition testing, with the general random reliability test being the standard. They differ from those performed by Duran and Ntafos, mainly in that the assumed error rate is low, a situation in which it is possible to observe variations in the effectiveness of partition tests. The results in [10] can be summarized as follows:

1) If partitions are not perfectly homogeneous, then the degree of homogeneity not very important.

2) The partition check is improved if one or more partitions have a much higher probability of failure than overall.

3) Finer partitions are disadvantageous if their error rates are uniform.

Overlaps must also be taken into account in comparison methods. For example, in statement testing, the natural subdomains consist of inputs that cause the same statement to be executed; these subdomains are not disjoint. However, a real partition can be formed by the intersection of the natural subdomains. Points selected from overlapping natural subdomains may be considered to be selected from the true partition, but with an increased sampling density in the overlap. Insofar as the error rate in these parts is low, one method will suffer relatively general random tests; as it is high, the method will be better. If there is no reason to believe that the overlapping regions are prone to error, then a method of overlapping comparisons that would appear would be slightly worse 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 developed early in the development cycle. Before code is written, or even designed, the software requirements / specifications can be parsed, and the possible inputs are divided into equivalence classes by the required functionality. A good example of this type of test is shown in [20] where the system under test has a number of instructions each with a number of parameters and the partitions are first defined by a command and then refined by the parameter value.

Design-based tests can also be developed before code is written. If the program has been divided into modules of some kind, and defines its overall structure in terms of their interfaces and dependencies, partitions can be defined according to a kind of large-scale statement. Two inputs are in the same subdomain when they call the same module.

Design based testing is the better method after the reliability theory comparison. The above results 2) and 3) apply. In specification-based partitions, there is less reason to believe that error rates are different than in design-based partitions. In specification-based partitions, there is less reason to believe that error rates are different than in design-based partitions. The subjective assessments of the designers about the difficulty of implementing each module and the ability of the assigned program staff are very good indications where to look for errors in the design-based partition. Moreover, specification-based partitions strive for coverage of functionality, which is obviously a refinement criterion that is not related to the error rate. On the other hand, the design-based partitions can be refined as the implementation progresses, using improved information. Code metrics may indicate error-prone modules or parts thereof, and most importantly, programmers may give subjective estimates of where problems might occur because the coding task was perceived as of insecure quality. Therefore, the theory predicts that the testing effort should be used for design-based rather than specification-based testing.

We conclude this section with a collection of assertions about practical methods and their comparison, which supports the theory for the reasons given in brackets. Some of them are counterintuitive and point to further analysis and experimentation:

a) it is a mistake to refine a partition just to better match their tests.

b) Combining test methods to create intersections of partitions, for example, by refining a structure partition with a specification-based one as suggested in [25] no improvement over both methods alone.

c) Dataflow test methods, especially those containing more program context [15], should be superior to the full path test.

d) Specification-based tests that do not consider possible development and deployment difficulties are not very good. The "cleanroom" project [26] used and could be more specification-based testing with design-based testing.

e) All tests used in walkthroughs or as typical design and implementation cases are useless for later debugging.

f) Special value tests are valuable because the special values represent cases that are likely to fail. Therefore, a specific value matching the above items d) or e) is not so useful.

**3.2 COMPARITSON WITH ERROR RATE ASSURANCE:**

The statistical reliability is a better comparison test measure than the subsumed relation considered in §2.2. It is universally applicable and its intended meaning - how likely the software fails in normal use - seems to be clearly related to quality. However, the plausibility of traditional software reliability has been called into question. The errors in the software reliability theory are:

R) It assumes a long-term average failure rate for which there is little supportive evidence

Rather, it is observed that software failures occur in bursts with stable periods between. This phenomenon is probably related to the following distribution used in (D).

D) It is based on an operational distribution of input values that are assumed to describe the normal use of the software, which may not exist in reality. The usage can have so many : modes that no distribution is typical. More importantly, the lack of a distribution reveals a major conceptual flaw: The software quality does not depend intuitively on normal usage. We judge the quality of how well the software behaves in unusual circumstances.

I) It is based on independent samples from the operational distribution. Intuitively, making unbiased input choices does not increase confidence in the results because it can happen that the internal behavior of the program is nearly the same for quite different inputs. Since software is deterministic, repeating tests in this way can not provide security.

W) Whatever one thinks of the assumptions of the theory, the derived results are not intuitively correct. For example, the size and complexity of programs, specifications, and their input ranges do not occur in any way. For example, the size and complexity of programs, specifications, and their input ranges do not occur in any way. The operational distribution (D) given above is at the heart of the difficulty of the conventional theory.

Thus, the comparison proposed in § 3.1 is flawed - methods are compared only in their ability to detect an error in normal use. Insofar as this basis misses the mark, this also applies to the comparison.

It is common practice to express software quality in a way different from the reliability view. For example, specifying the quality of the space shuttle flight software is said to have less than 0.1 errors per thousand lines of code. Such a measure is inherently quality-related because its expression is independent of use and because it is directly related to the process that produces software and the mistakes made in that process. Based on the error rate, a sampling theory [9] can be constructed. This theory, in contrast to reliability theory, is referred to as the assurance theory of § 3.1.

In the insurance theory, the assumption (R) is replaced by the assumption of an error rate. The assumptions (D) and (I) are replaced as follows:

D´) It assumes that text errors are evenly distributed across the state space of the program, which consists of all checkpoints and all values of variables at those points. That is, an error is defined as a state space value that results in an error. This is an improvement over (D) because the distribution is fixed and because it is plausible that programmers will make mistakes with equal probability at such a point. The objection to this view is that the term error is not clearly defined [22, 17].

I´) To search for defects, the state space must be scanned independently. Precisely because samples from the input domain do not penetrate uniformly into the state space, (I) is called into question. However, tests are necessarily carried out across the input domain and all the way to require that they obey an unknown distribution is impractical.

Insurance theory argues that (D ') is often meaningful in practice, and that (I') is correct but unattainable. The insurance theory improves (W): Program size and cardinality of the input domain are entered accordingly. The number of tests $N$ required to guarantee a given error rate is approximately proportional to the program size $M$ and the domain size $D$ and inversely proportional to the number of errors per line $f$. For a confidence of 99% and a large $D$ / $f$, the proportionality constant is about $10^{-4}$ error / line [2]: $N \approx 10^{-4} * MD / f$. For example, a 50K line program with one million point domain requires $N \approx 5 * 10^{10}$ tests to guarantee less than 0.1 errors per thousand lines.

Assurance theory does not suggest a practical test method, either because the required independent samples can not be identified in the input domain or because too many tests are required. However, if the theory is correct, it provides a standard for comparing other methods. Hamlet [10] tries such a comparison. Assuming that tests are selected to uniformly sample the state space, it is shown that partition tests are far inferior to general random testing and that the disadvantage is increased by more partitions or lower error rates.

Again, we need to examine the theory and our intuition about partition tests to find the assumptions that influence those results. Tests in general do not directly take state space, as the theory assumes, and intuitively, selecting inputs that stimulate different program states is most difficult when the entire domain is used. Partitions representing an input distribution into the low-distortion state space would exceed any general test scheme. This observation has implications for the level at which the reliability test should be carried out.

A distortion in state space sampling occurs because program control points can not be reached there with all the values of the variables. Control predicates are part of the reason for limited state values; assignment to one part of the state are the others. If instruction-level specifications were available, ideal backup tests could be performed by first finding the range of possible variable values in a statement and then sampling them with an even distribution over that range. Howden has proposed "functional" testing in which each piece of code is individually tested against a local specification [14, Ch. 5]; an insurance test could be carried out for these units. A system designed for "testability" would aim 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 be an ideal safety test. Even without low-level features and design-in-state space access and reliability testing, unit testing should be more likely than integration testing [12]. At the integration level reliability testing may be more appropriate.

We apply the insurance theory to two (overlapping) partition methods, industry tests and mutation tests.

The natural input range subdomains for the industry test are those in which inputs execute the same condition with the same branching result. We ask how well tests from these subdomains cover the state space. The branch cover includes at least two points in the state space of each conditional expression; that is, two points in the state space of the basic block precede this state. However, the state at the location immediately following the branch must take only a single value. There is no necessary state space connection between the subdomains; it may happen that the subdomains together force no more than the minimum coverage of two states for each condition. There is no necessary state space connection

between the subdomains; it may happen that the subdomains together force no more than the minimum coverage of two states for each condition. This situation was described in §2.2 as a "trivial" satisfaction of a method. Intuitively, the letter of the method is met, but through data that has a chance to expose an error. 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. 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 tests contain such inputs that kill the same mutant. Intuitively, some mutants can only be killed by unusual conditions and different mutants may require different states. So to kill all mutants at a checkpoint, the condition needs to be explored in detail there. Failures that arise only through a combination of errors defeat the mutation because the mutants do not require at one point of failure that the state varies sufficiently; Only when another error point is reached can the earlier variation be considered too narrow. Therefore, mutation should be a better backup method than industry tests, but it is not perfect.

This discussion proposes a tool to measure the expected software security from test data, regardless of the origin of the test. The program under test can be instrumented in the same way profiles are obtained to indicate state space coverage. If the coverage is poor, the test may be improved, or perhaps the state space is limited in some way. The tester has to make the decision, since the problem is generally unsolvable. However, if two tests are compared, the better is the insurance viewpoint is the one with the better state space coverage.

The concept of state space coverage proposes a modification of the subsumed relationship that is better in line with reality: one method is better than another if its state space coverage is better. This state-space subsume has the property that each of a The poorer method must also be revealed by a better one, since the responsible state must be covered by the latter. It therefore does not suffer from the defect described in §2.2-2.3. However, only methods that are not monotonic could be worse, unless the better method would be perfect for detecting errors, so the analytic application of this idea is not promising. On the other hand, with the tool proposed in the previous section, an experiment could be developed to examine state space subsumption. If the fault tolerance theory is correct, such experiments would measure the actual effectiveness 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 test methods on a reliability or assurance sample must be based on standards. Reliability theory suggests partitioning methods that focus on errors at the expense of partition coverage and homogeneity. The Assurance theory states that tests must be performed at or below the unit level to control state space coverage. The qualitative comparisons presented here suggest a further investigation of methods with these standards.

## 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 testingstrategies, *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. The´venod-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, Comparisonof 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.