Software Trustability Analysis

W. E. HOWDEN and YUDONG HUANG
University of California at San Diego

A measure of software dependability called *trustability* is described. A program p has trustability T if we are at least T confident that p is free of faults. Trustability measurement depends on *detectability*. The detectability of a method is the probability that it will detect faults, when there are faults present. Detectability research can be used to characterize conditions under which one testing and analysis method is more effective than another. Several detectability results that were only previously described informally, and illustrated by example, are proved. Several new detectability results are also proved. The trustability model characterizes the kind of information that is needed to justify a given level of trustability. When the required information is available, the trustability approach can be used to determine *strategies* in which methods are combined for maximum effectiveness. It can be used to determine the minimum amount of resources needed to guarantee a required degree of trustability, and the maximum trustability that is achievable with a given amount of resources. Theorems proving several optimization results are given. Applications of the trustability model are discussed. Methods for the derivation of detectability factors, the relationship between trustability and operational reliability, and the relationship between the software development process and trustability are described.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Reliability, Verification

Additional Key Words and Phrases: Analysis, dependability, detectability, failure density, statistical, testability, testing, trustability

INTRODUCTION

Many testing methods for software have been proposed. Although these methods have been found to be effective in revealing the presence of faults, they do not usually indicate the degree to which the tested software can be considered to be dependable.

Possible approaches to dependability estimation include reliability analysis, in which random testing is used to determine the probability of program failure. There are several ways of formalizing this concept. Examples include the simple model, used in Howden [1987], in which random testing over a

This research was supported by the Office of Naval Research.
Authors' address: Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093; email: howden@cs.ucsd.edu.

program's operational distribution is used to establish confidence in an upper bound on the program's failure density. A program's operational distribution is the distribution with which its input will appear when the program is used in its operational environment. A program's failure density is the probability of its failing when it is executed over its operational distribution. It is the weighted (by the operational distribution) fraction of a program's input domain over which the program will fail. Failure density analysis can be used to compute the number of tests needed to achieve a required level of confidence in a given failure density bound. Other approaches to reliability analysis include more-complex measures, such as estimates of mean time between failure [Musa et al. 1990].

The disadvantages of the failure-density/random-testing approach to dependability estimation are that it requires a knowledge of a program's operational distribution, and it requires a large number of tests for even a modest level of confidence in a modest failure density bound. This observation has been previously made by several authors, e.g., Hamlet and Voas [1993] and Howden [1987]. For example, 461 tests are needed for establishing a bound of 0.01 with confidence 0.99.

One of the problems with random testing, and hence dependability estimation based on random testing, is that it does not use information about fault classes or special properties of classes of programs and program functions. It is commonly accepted in software testing that "intelligent" or "guided" choices of tests are more likely to reveal faults than random sampling over an operational distribution. Guided testing involves rules that direct the selection of tests toward certain subdomains of a program's domain. It includes methods such as coverage-based testing (e.g., Clarke et al. [1985], Laski and Korel [1983], Ntafos [1988], Rapps and Weyuker [1985], and Woodward et al. [1980]), functional testing [Howden 1985; 1987; Marick 1992], fault-based testing [DeMillo et al. 1978; Foster 1980; Hamlet 1977a; 1977b; Howden 1982; White and Cohen 1980], and specifications-based testing (e.g., Gannon [1981]).

Guided dependability evaluation includes not only testing, but also program analysis and program development methods. Requirements analysis, for example, can be used to detect functional faults early in the development of a system. Proofs of correctness may be used to detect algorithm design faults, and static analysis to detect coding faults.

We describe an approach to dependability estimation that can be used to incorporate reliability-oriented methods, such as random testing, and guided testing and analysis methods into a common dependability framework. It is based on the concept of *trustability*, which is defined to be confidence in the absence of faults. Trustability measurement depends on the availability of *detectability* factors. The detectability of a method is the probability that it will discover a fault in a program, if a fault is present. The trustability/detectability framework can be used to define situations formally in which one method is better than another, to consider optimal mixes of methods, and to characterize the kinds of assumptions that have to be made in order to conclude that a program has a required level of dependability.

The article has three general sections. Previous theoretical work on testing included comparative analyses of the detectability of two very general methods, random and partition testing, for the general class of all faults. In Section 1 we introduce several original results in which we formally characterize and state circumstances under which one method is superior to another. These results are examples of the kinds of theoretical results that are possible in the study of detectability. Section 2 contains trustability results. Here we indicate the kinds of trustability properties that we can consider, independently of the sources of detectability information. In Section 3 we consider more-practical questions, such as the sources of information about fault classes and detectability factors and the relationship between trustability and operational reliability.

1. DETECTABILITY ANALYSIS

Failure Density and Detectability Analysis

Previous analyses of method detectability included comparisons of partition testing [Richardson and Clarke 1981] with random testing. In partition testing, the input domain of a program is decomposed into subdomains, and tests are distributed equally to each of the decomposition's subdomains, rather than randomly over the whole domain. Within a subdomain, tests are chosen randomly. The term "random" is normally associated with the uniform distribution, but other distributions are possible. Partition testing can be viewed as a general model for testing methods which involve disjoint decomposition of a program's input domain, such as informal functional testing.

Earlier work by Duran and Ntafos [1981; 1984] and Hamlet and Taylor [1990] used failure densities to investigate the conditions under which partition testing is superior to random testing. In Hamlet's work, the effectiveness of a testing method was equated with the probability that its use would cause at least one program failure to occur. In the case where the failure density is zero, the probability will be zero, so we can think of this probability as being a measure of detectability in the sense that it is the probability of revealing a fault, when a fault is present (i.e., when the failure density is nonzero). We will use p_r to refer to the effectiveness of random testing and p_p to refer to the effectiveness of partition testing. Hamlet gave examples that indicated that partition testing was not superior to random testing ($p_p \leq p_r$) under a variety of circumstances. He also informally described situations under which partition testing would be superior and gave tables of empirical results that supported his conclusions.

Later work by Weyuker and Jeng [1991] investigated the partition model further and made a number of observations. Using examples, they showed that partition testing could be worse than random testing. They also identified the worst kind of partition from among different partition possibilities. In another observation they proved that when all of a partition's elements are of the same size, and an equal number of tests is chosen from each, then

partition testing is at least as good as random testing. They also proved that if partition elements have the same individual failure densities, then partition and random testing are equally effective. In their paper they repeated the observation by Hamlet that partition testing will be more effective than random testing when it results in the creation of domain subdivisions with high failure densities and low frequencies.

The partition-testing model also appears in Frankl and Weyuker [1993], where they use it to compare different testing methods. Different possible relationships between test method partitions are defined, and the fault-revealing effectiveness of methods having those partitions are compared.

The simple analyses that we describe below require that input domain decompositions be partitions in the sense that they are complete and disjoint, i.e., all of a program's input domain must be considered, and the decomposition elements must be nonintersecting. This is the property that a decomposition must have to be classified as a partition. Much of the work on decomposition-oriented analysis uses this term and depends on a decomposition being a true partition, but we note that in some cases the word is loosely used to refer to decompositions that are not true partitions. We will use the word in its correct mathematical sense and will refer to disjoint element decomposition testing methods as *partition-testing methods*. The disjoint decomposition element property holds for methods such as functional testing and path testing. It does not hold for testing methods that involve coverage measures, such as statement and branch coverage, but there are ways in which disjoint element decomposition (i.e., partition) analysis can also be used to analyze their effectiveness [Frankl and Weyuker 1993; Howden and Huang 1993]. Weyuker and Jeng [1991] have previously documented the problem of intersecting decomposition subdomains and made several suggests on how this problem could be ameliorated.

Theoretical Detectability Results

Suppose that the input domain of a program p is decomposed into disjoint subdomains $D_1, D_2, \ldots, D_n$. Suppose that for each D_i, p has a failure density d_i when it is executed over randomly selected data from the subdomain D_i according to some distribution such as the uniform distribution. Suppose that when p is executed over its complete domain D using the same random distribution, D_i will be executed with frequency f_i. Then we can use the following formula to define the failure density d for the program p when it is tested over D using random testing:

$$d = \sum_{i=1}^{n} f_i d_i.$$

Suppose that we execute p over a sequence of N tests. Then the probability of finding at least one fault when random testing is used is given by the following [Duran and Ntafos 1981; 1984]:

$$1 - \left(1 - \sum_{i=1}^{n} f_i d_i\right)^N.$$

There are two possible partition-testing models: deterministic and statistical. In the deterministic model, each decomposition element is allocated an equal number of tests. In the statistical model, we assume that, with equal probability, we choose a decomposition subdomain and then (as in deterministic partition testing) randomly choose an element from that subdomain. If p is tested using statistical partition testing, then the probability of finding at least one fault during N tests is given by the formula

$$1 - \left(1 - \sum_{i=1}^{n} (1/n)d_i\right)^N.$$

If deterministic partition testing is used, then the probability of finding at least one fault is given by the following [Duran and Ntafos 1981; 1984]:

$$1 - \prod_{i=1}^{n} (1 - d_i)^{N/n}.$$

Note that in the above formula the probability of finding a fault will be zero if there is no fault, (i.e., the failure densities are zero), so that these probabilities are detectability measures.

In the following, p_{dp} denotes the effectiveness of deterministic partition testing and p_{sp} the effectiveness of statistical partition testing. In those cases where a theorem holds for both kinds of partition testing, we use the notation p_p.

We can use these formulae to prove theorems that compare the effectiveness of random and partition testing. For completeness, we first repeat a theorem that is found in both Hamlet and Weyuker. It applies to both deterministic and statistical partition testing.

THEOREM 1.1 [HAMLET AND TAYLOR 1990; WEYUKER AND JENG 1991]. *Suppose that the failure densities d_i for the subdomains D_i of a program P are all the same. Then partition and random testing are equally effective in discovering faults, i.e., $p_r = p_p$.*

In previous analysis of partition testing only the deterministic model was used. We can prove certain results using the statistical model that are not provable using the deterministic model. We will first compare partition and random testing using the statistical model and then give a theorem describing the relationship between the statistical and deterministic models. This is followed by theorems that give important results for the more traditional, deterministic model.

In previous work, as we have mentioned above, the observation was made that partition testing would be more effective in situations where there were partition subdomains with high failure densities and low frequencies, and random testing would be better when high failure densities were associated with high frequencies. Examples were used to support this claim. By characterizing idealized situations having these patterns of frequencies and failure densities, and adopting the statistical model of partition testing, we are able to prove that this observation is correct.

THEOREM 1.2. *Suppose that the domain D of program p is decomposed into subdomains* D_i, $1 \le i \le n$, *such that subdomains with higher failure densities have lower frequencies and subdomains with lower failure densities have higher frequencies (i.e., for all* $i, j, 1 \le i \le n$, $1 \le j \le n$, *if* $d_i \ge d_j$, *then* $f_i \le f_j$*). Then if not all failure densities are equal, statistical partition testing will be more effective than random testing, i.e.,* $p_{sp} > p_r$.

PROOF. In the following, and also in the proofs of Theorems 1.4, 1.5, and 1.6, we will assume that the number of tests, N, is equal to the number of partition subdomains, n. To generalize, we assume that we can divide N up to N/n sets of tests and apply the relevant testing methods repeatedly.

We prove the theorem by showing that

$$\frac{1}{n}\sum_{i=1}^{n} d_i > \sum_{i=1}^{n} d_i f_i$$

which implies that

$$p_r = 1 - \left(1 - \sum_{i-1}^{n} d_i f_i\right)^n < 1 - \left(1 - \frac{1}{n}\sum_{i=1}^{n} d_i\right)^n = p_{sp}.$$

By the conditions of the theorem, it is possible to number the partition subdomains so that their frequencies are in monotonic increasing order, and their failure densities in monotonic decreasing order. Because the frequencies sum to 1, there is some k, $1 < k \le n$, such that for $i > k$, $f_i \le 1/n$, and for $i > k$, $f_i > 1/n$. The case where all frequencies are $1/n$ corresponds to the case where $k = n$. When this is not the case, k must be less than n. Because the frequencies sum to 1, we can show that

$$\sum_{i=1}^{k} (1/n - f_i) = \sum_{i=k+1}^{n} (f_i - 1/n)$$

so that

$$\sum_{i=1}^{k} (1/n - f_i)d_i \ge \sum_{i=1}^{k} (1/n - f_i)d_k \ge \sum_{i=1}^{k} (1/n - f_i)d_{k+1}$$

$$= \sum_{i=k+1}^{n} (f_i - 1/n)d_{k+1} \ge \sum_{i=k+1}^{n} (f_i - 1)d_i$$

and hence

$$\sum_{i=1}^{n} (1/n - f_i)d_i \ge 0.$$

If the failure densities are not all equal, we can make the inequality strict, and hence

$$\frac{1}{n}\sum_{i=1}^{n} d_i > \sum_{i=1}^{n} d_i f_i. \qquad \square$$

THEOREM 1.3. *Suppose that the domain for p is decomposed into subdomains* D_i, $1 \le i \le n$, *such that subdomains with higher failure densities have*

higher frequencies and subdomains with lower failure densities have lower frequencies (i.e., for all i, j, $1 \le i \le n$, $1 \le j \le n$, if $d_i \ge d_j$, then $f_i \ge f_j$). Then if not all failure densities are equal, statistical partition testing will be less effective than random testing, i.e., $p_{sp} < p_r$.

PROOF. The proof is analogous to that for Theorem 1.2. □

A version of Theorem 1.3, but for deterministic partition testing, was previously introduced in a less-formal form by Hamlet and Taylor [1990] and discussed in more detail by Weyuker and Jeng [1991]. It describes some simple conditions under which deterministic partition testing will improve the effectiveness of a testing effort. We can relate their result to the above result by relating statistical and deterministic partition testing, using the following theorem.

THEOREM 1.4. $p_{dp} \ge p_{sp}$. (*Note that this is the same as the following theorem: if the frequencies for all subdomains are not all equal, then $p_p > p_r$ [Weyuker and Jeng* 1991].)

PROOF. Under the assumption of the theorem, we have

$$p_{dp} = 1 - \prod_{i=1}^{n} (1 - d_i) \quad \text{and} \quad p_{sp} = 1 - \left(1 - \frac{1}{n} \sum_{i=1}^{n} d_i\right)^n.$$

We can show that $p_{dp} > p_{sp}$ if

$$\prod_{i=1}^{n} (1 - d_i) < \left(1 - \frac{1}{n} \sum_{i=1}^{n} d_i\right)^n.$$

Let $E_i = 1 - d_i$. Then

$$\sum_{i=1}^{n} E_i = n - \sum_{i=1}^{n} d_i \quad \text{and} \quad \frac{1}{n} \sum_{i=1}^{n} E_i = 1 - \frac{1}{n} \sum_{i=1}^{n} d_i,$$

and what is required is to prove that

$$\prod_{i=1}^{n} E_i < \left(\frac{1}{n} \sum_{i=1}^{n} E_i\right)^n.$$

We ignore the case where $n = 1$, since in this case statistical and deterministic partition testing are equivalent to random testing, since there is no partition. The proof is by induction, with base case $n = 2$. The base case can be easily shown. Assume that the formula holds for $n = k$, i.e.,

$$\prod_{i=1}^{k} E_i < \left(\frac{1}{k} \sum_{i=1}^{k} E_i\right)^k,$$

and that $n = k + 1$. Without loss of generality, assume that E_{k+1} is a maximal E_i. Then

$$\prod_{i=1}^{k+1} E_i = \left(\prod_{i=1}^{k} E_i\right) E_{k+1} < \left(\frac{1}{k} \sum_{i=1}^{k} E_i\right)^k E_{k+1}.$$

Let

$$E = \frac{1}{k}\sum_{i=1}^{k} E_i \quad \text{and} \quad E_{k+1} = E + x.$$

Then

$$\left(\frac{1}{k}\sum_{i=1}^{k} E_i\right)^k E_{k+1} = E^k(E + x) = E^{k+1} + E^k x.$$

Alternatively,

$$\begin{aligned}\left(\frac{1}{k+1}\sum_{i=1}^{k+1} E_i\right)^{k+1} &= \left(\frac{kE + E_{k+1}}{k+1}\right)^{k+1} = \left(\frac{kE + E + x}{k+1}\right)^{k+1} \\ &= \left(E + \frac{x}{k+1}\right)^{k+1} \\ &= E^{k+1} + (k+1)E^k\left(\frac{x}{k+1}\right) \\ &\quad + \frac{(k+1)k}{2!}E^{k-1}\left(\frac{x}{k+1}\right)^2 + \cdots \\ &> E^{k+1} + E^k x,\end{aligned}$$

and hence $p_{sp} < p_{dp}$. □

Theorems 1.2 and 1.4 can be combined to prove that, under the conditions of Theorem 1.2, deterministic partition testing is more effective than random testing (i.e., $p_{dp} > p_r$). There are cases that satisfy the conditions of Theorem 1.3 under which either random or deterministic partition testing is more effective, so that the comparison of deterministic partition and random testing is more complicated.

We can approach the comparison of deterministic partition and random testing in a different way, by comparing the ratios of their effectiveness. Informal observations about ratios of effectiveness were made by Hamlet and Taylor [1990]. In the following theorems we give formal statements of provable effectiveness ratios.

THEOREM 1.5. *Combinations of frequency and failure densities can occur for which the ratio of effectiveness of deterministic partition to random testing is arbitrarily large.*

PROOF.

$$\frac{p_{dp}}{p_r} = \frac{1 - \prod_{i=1}^{n}(1 - d_i)}{1 - \left(1 - \sum_{i=1}^{n} d_i f_i\right)^n}.$$

Consider the case where $d_1 \neq 0$ and the other d_i are arbitrarily small. Assume that f_1 is arbitrarily small. Since the f_i sum to 1, this implies that

$$\sum_{i=1}^{n} d_i f_i$$

is arbitrarily small, so that

$$\frac{p_{dp}}{p_r} = \frac{d_1}{1 - \left(1 - \sum_{i=1}^{n} f_i d_i\right)^n}$$

will be arbitrarily large. □

THEOREM 1.6. *The ratio of effectiveness of random-to-deterministic partition testing is bounded by*

$$\frac{p_r}{p_{dp}} < n,$$

where n is the number of subdomain elements in a partition.

PROOF. Let $d_{\max}$ be the maximal value of the partition subdomain failure densities. Then

$$p_r = 1 - \left(1 - \sum_{i=1}^{n} d_i f_i\right)^n < 1 - (1 - d_{\max})^n$$

and

$$p_{dp} = 1 - \prod_{i=1}^{n} (1 - d_i) > 1 - (1 - d_{\max}) = d_{\max}$$

so that

$$\frac{p_r}{p_{dp}} = \frac{1 - \left(1 - \sum_{i=1}^{n} d_i f_i\right)^n}{1 - \prod_{i=1}^{n} (1 - d_i)} < \frac{1 - (1 - d_{\max})^n}{d_{\max}} = F(d_{\max}).$$

To obtain the extremal points of the functions $F(d_{\max})$ we need $F'(d_{\max}) = 0$, or

$$d_{\max} n (1 - d_{\max})^{n-1} - \left(1 - (1 - d_{\max})^n\right) = 0.$$

Hence, at the extremal points for $F(d_{\max})$, we know that

$$(1 - d_{\max})^n = \frac{1}{1 + \dfrac{d_{\max} n}{1 - d_{\max}}}$$

so at those extremal points

$$F(d_{\max}) = \frac{1 - \left(\dfrac{1}{1 + \dfrac{d_{\max} n}{1 - d_{\max}}} \right)}{d_{\max}} = \frac{n}{1 + (n-1)d_{\max}}.$$

This fraction obtains its maximal value of n as $d_{\max}$ converges to zero and hence

$$\frac{p_r}{p_{dp}} < n. \qquad \square$$

The above two theorems indicate that the potential "loss" in using partition rather than random testing is bounded, but its potential "gain" can be arbitrarily large. We carried out several experimental studies in which failure densities and frequencies were generated randomly which indicated that we can expect the "average" performance of partition testing to be superior to that of random testing.

Detectability Analysis and Standard Testing Methods

The above results can be used to help explain why methods such as functional testing may be more effective than simple random testing. In functional testing, programmers are advised to choose separate tests for extremal cases. It is common knowledge that the code for these special cases is often as fault prone as the code for cases corresponding to the nonextremal data. We can expect special-case code to have a relatively small corresponding program subdomain. This implies that during random test selection over the whole program domain it will be chosen with low frequency. It is also often true that extremal-case subdomains have the property that if a program fails using test data designed for those cases, it will fail over the whole extremal-case subdomain. It follows that the proportion of failure-inducing input in failure-causing extremal-case subdomains will be high. This indicates that functional testing is likely to be associated with decompositions in which there are subdomains with low frequencies and high failure densities, the conditions under which theory indicates that guided (i.e., partition) testing will be more effective than random testing.

It is noted that the generation of tests using random testing may be considerably cheaper than the generation of functional tests, which may require considerable analysis. In this case, the comparison of random and functional testing should include a cost benefit factor, since it may be possible to use a larger number of random tests, at the same cost, than functional tests. Additional detectability research might include considerations such as this. In many situations, however, it is the cost of checking output, of verifying correct behavior, for a set of tests that is the overwhelming cost factor, in which case the above analysis which assumes equal numbers of tests for random and functional testing is appropriate.

We note that the above argument is only of practical interest if it is associated with a domain partition in which there are a "small" finite number of elements, ruling out degenerate examples in which it is applied to partitions in which every element of an input domain is divided into its own extremal-case subdomain.

As pointed out by Weyuker and Jeng [1991], the above results are difficult to apply to coverage measure testing methods, such as branch testing, because the decompositions that such a method induces do not have disjoint subdomains (i.e., they are not partitions). There is, however, a way to apply the partition-based failure density results to coverage measure testing. Suppose that a sequence of tests has been carried out, and it is discovered that some part of a program, say some branch, has not been tested. Then consider a partition of the domain into two subdomains, one consisting of the subdomain that causes that branch to be followed, and the other subdomain its complement. Now if random testing had been used up to this point, the fact that one of the branch subdomains had not been tested would indicate that it has a low frequency with respect to random testing. If both branches lead to equally complex parts of a program, we might expect them to be equally likely to have faults, and hence, initially, expect them to be associated with partition subdomains that have equal estimated failure densities. However, suppose that during the initial testing that no failures occur for the tested subdomain. Then we will have a new situation in which the failure density for the executed part of the domain should be assumed to be lower than that of the subdomain corresponding to the unexecuted branch. This means that a situation has developed in which a low-frequency subdomain (the untested one) is associated with a (relatively) high failure density (estimation). The failure density theory developed above suggests that we should then use partition testing to force the testing of the untested subdomain, which coincides with the common use of coverage methods as a corrective measure for an otherwise random approach to testing, e.g., Howden [1987].

2. TRUSTABILITY ANALYSIS

In the study of detectability, we analyze the effectiveness of different methods in terms of factors such as failure densities and properties of a program's operational distribution. In trustability analysis, we characterize dependability properties in terms of method detectability factors. For generality, we also include fault class frequency factors, and program classes. It is possible to create different models of trustability. We first introduce and study properties of a very simple model, and then describe possible elaborations later in the article.

Detectability and Trustability

The simple concept of trustability that we will use here is based on hypothesis testing. In hypothesis testing, we have a hypothesis whose truthfulness we test with an experiment. The experiment is designed so that, on the basis of the experiment, we reject the hypothesis or accept it with a certain risk

factor, say q. The risk factor is the probability that the experiment will not result in rejection of the hypothesis when the hypothesis is false. When we accept the hypothesis, we say that it is true with confidence $1 - q$.

In our application of hypothesis testing, we model software testing and analysis as a process that involves the selection of a program from a program space, followed by the application of one or more evaluation methods to that program. The selection process may occur in stages, and the evaluation methods applied to intermediate program documents. Our hypothesis is that the selected program is free of faults. If we discover a fault we reject the hypothesis. If we do not discover a fault, we accept the hypothesis with a risk factor associated with the detectability of the methods for the program space. The *detectability* of a method M is the probability that M will detect the faults in a selected program, if that program contains a fault.

In the simplest case, suppose that a single method M with detectability D is applied to a selected program p. Then either the program has faults, or it does not. In the case where it has faults, the risk factor is $1 - D$. In the case where it does not have faults, the risk factor is zero. Suppose that M is used to evaluate p, and no faults are detected. The risk of our falsely concluding that p is free of faults is at most $1 - D$ so that our confidence in freedom from faults, i.e., the *trustability* of p, is at least T where

$$T = 1 - (1 - D).$$

The above formula is an example of a *trustability guarantee*. We can construct more powerful and more general formulae by taking different kinds of information into account.

In practice M may involve the repeated application of some step until no fault is discovered, and it may involve the correction of faults as they are discovered. At some point the "exit criteria" for the method is met, and then it terminates. If D is the detectability of a method, then $1 - D$ is the probability of the exit criteria being incorrect, i.e., the method is terminated when there are still faults in a program.

The detectability of a method may be *deterministic* or *probabilistic*. If it is deterministic, then it is equal to 1 for some associated fault class, since it is guaranteed to find faults of that type always. Static analysis methods such as those that look for uninitialized variables are examples of deterministic methods. If a method is probabilistic, it may or may not find a fault from some class, with a certain probability. Methods with probabilistic detectability involve some kind of sampling process. The sample space might be a space of possible programs, a space of program analyses, or the space consisting of a program's input domain. It may be observed, for example, that in a given environment that a method M has a certain probability of detecting faults for programs generated in that environment. In this case the sample space is the set P of possible programs that could be generated, and the development of a program is modeled as a process in which a sample from P is selected.

The detectability of random testing is probabilistic, with sample space equal to a program input domain. In this case detectability depends on the program under analysis and is determined by its failure density.

We note that probabilistic detectability may involve components from several sampling processes.

Trustability and Fault Class Frequencies

We can incorporate fault frequencies into the trustability approach, resulting in a more refined measure of trustability. Suppose that, for some program sample space, f is the probability of a program p occurring that has one or more faults and D is the detectability of an evaluation method M. The probability of our having a program with faults in it, and our not detecting one or more of those faults using M, is $f(1 - D)$. Hence, if we apply M to p, we can say that p has no faults with trustability at least T where

$$T = 1 - f(1 - D).$$

If fault frequencies are available, we can use them to establish a base-line level of program confidence, before any testing or analysis has been done. This *null-experiment* level of trustability is defined by the formula

$$T = 1 - f.$$

The incorporation of fault class frequencies into the trustability model is important since it allows us to model more accurately the ways in which programmers apply fault detection methods to programs: they identify the more commonly occurring classes of problems, and apply those methods which their experience indicates are most effective.

Trustability and Multiple Fault Classes

Suppose that we have a single method M with detectability D_j for fault classes F_j having occurrence frequencies f_j, $1 \leq j \leq s$. Let F be the union of the fault classes. Then we may wish to determine what level of trustability for F we can have if we apply the method to each fault class and find no fault. We can show that trustability will be at least T, where

$$T = 1 - \max_{1 \leq j \leq s}\{f_j(1 - D_j)\}.$$

The above formula is a special case of a more general formula whose correctness is discussed below.

Trustability and Multiple Methods

Here, we consider the possibility that we may have more than one method for some class of faults F with fault frequency f. Suppose that M_i, $1 \leq i \leq r$, are a set of methods with detectability D_i for F. Assume that we apply the methods "in parallel" to copies of the program drawn from the program sample space. Then the probability of there being a fault of type F in a program, and our not detecting it in the application of the r methods, is at most

$$f(\min_{1 \leq i \leq r}\{(1 - D_i)\}).$$

This means that trustability is at least T where

$$T = 1 - f(\min_{1 \leq i \leq r}\{(1 - D_i)\}),$$

i.e., is that which is achieved by the method with the highest level of detectability.

In the case of multiple methods, it would be nice if they had a "cumulative" effect so that if we had two methods M_1 and M_2 with detectability D_1 and D_2, and we did not see any faults when they were applied, we could say that the probability of this happening was at most

$$(1 - D_1)(1 - D_2).$$

In general, this formula will not hold since it may involve probabilistic detectability factors based on sampling over a program space. To get a cumulative effect we would have to sample twice, and D_1 and D_2 would be estimates for different programs. However, the desired cumulative effect can be achieved under certain circumstances. In general, it requires that the probabilistic detectability factors for the two methods involve independent sampling. For example, suppose that D_2 is the detectability of M_2 for finding faults that are not discovered by an application of M_1, i.e., D_2 is the detectability of M_2 *relative to* M_1. Then the probability of selecting a program with faults, and of not finding all of the fault by M_1, and then not finding all those faults not found by M_1 by using M_2, will be

$$f(1 - D_1)(1 - D_2).$$

The multiplicative effect can also be achieved under the following circumstances. Suppose that M_1 is a method whose probabilistic detectability factor depends on sampling over the program space, and that M_2 has detectability for which the probabilistic factor depends only on sampling over an input domain space.

The cumulative effect is also achieved when a repeatable method, such as random testing, is applied where repeated applications to a program can be assumed to have independent fault-revealing effectiveness. If D is the detectability of random testing for some class of faults, whose probabilistic aspect depends only on sampling over the input space, then the probability of not finding a fault in N independent samples will be

$$(1 - D)^N,$$

and the occurrence of N fault-free tests would give us trustability

$$1 - (1 - D)^N$$

for that class.

Trustability and Multiple Fault Classes and Methods

Here we consider the general case where we have multiple methods that are each effective for multiple fault classes.

Definition. Suppose that M_i, $1 \leq i \leq r$, is a program evaluation method, F_j, $1 \leq j \leq s$, a fault class, and P_k, $1 \leq k \leq t$, a class of programs. Then $D_{i,j,k}$, the detectability of method M_i for faults in fault class F_j occurring in programs of class P_k, is the probability that M_i will detect a fault of class F_j

in a program p from P_k, if p contains such a fault. If only a single program class P is involved, then $D_{i,j}$, $1 \le i \le r$, $1 \le j \le s$, will denote the detectability of method M_i for fault class F_j.

In the following definitions and theorems, we assume the existence of a set of methods M_i, $1 \le i \le r$, that can be applied to a program p from a class of programs P. The methods M_i are assumed to have detectability $D_{i,j}$ for detecting faults from fault class F_j in programs from P, where $1 \le i \le r$, $1 \le j \le s$. F is defined to be the union of the fault classes F_j, and f_j is the probability of occurrence for fault class F_j. The fault classes do not have to be disjoint.

Definition. A testing and analysis *strategy* S is a vector (N_i), $1 \le i \le r$, where N_i is an integer equal to 0 or 1, indicating whether or not method M_i is applied, called the *method usage factor* for M_i.

The choice of a value of 0 for a strategy component might occur for several reasons. If the estimated frequency for a fault class is low, then it is possible that the effort in using a method whose only purpose is to detect faults of that kind is wasted and could be better used for some other class.

THEOREM 2.1 (GENERAL TRUSTABILITY FORMULA). *Suppose that p is a program under analysis. Suppose that S is a testing and analysis strategy for p. If we see no (additional) faults in the application of the strategy S to p, then, for faults F, p has trustability at least T, where*

$$T = 1 - \max_{(1 \le j \le s)}\left\{\min_{(1 \le i \le r)}\left\{f_j(1 - D_{i,j})^N i\right\}\right\}.$$

PROOF. As above, when we have multiple methods for a fault class, we assume that they are all applied in parallel, or independently, to the program, if they have empirical detectability factors. The probability q_j that a fault from class F_j is present and that methods do not detect all the faults from that class is less than the probability that no single method detects the faults from that class and has the property

$$q_j \le \min_{(1 \le i \le r)}\left\{f_j(1 - D_{i,j})^N i\right\}.$$

This implies that the probability q that a fault from one or more fault class F_j is present, and is not detected, is given by

$$q \le \max_{(1 \le j \le s)}\{q_j\}.$$

Combining the above formulae, if no faults of type F are discovered by S then we can say that p has trustability at least T, where T is as defined in the statement of the theorem. □

Optimal Software Evaluation Strategies

Two basic situations will be considered. In the first, we have a cost budget B and a given set of fault classes and methods, and we want to establish the maximum trustability for a program p at cost less than or equal to B. In the

second situation, we want to establish a predetermined level of trustability L with minimum cost.

Definition. For a program p, and a method M_i, denote the cost of applying the method to p as $C_i(p)$. Suppose that S is a strategy for a set of methods M_i, $1 \le i \le r$, with usage factors N_i, $1 \le i \le r$. Then the *cost* of applying S to p is given by

$$C = \sum_{i=1}^{r} N_i C_i(p).$$

Suppose that S is a strategy with usage factors N_i, $1 \le i \le r$, and that T is a trustability guarantee formula. The strategy S is said to *support a trustability guarantee T at L*, if $T \ge L$ when it is evaluated using strategy S.

Definition. Suppose that B is a cost bound. Then an *optimal cost B trustability strategy* for a program p, and for some trustability guarantee T, is a maximum trustability guarantee strategy whose cost for p is less than or equal to B.

It is noted that for any nonnegative cost, there is a strategy that can achieve that cost, namely the *null strategy* (i.e., set all factors N_i, $1 \le i \le r$, to zero).

Definition. Assume that L is some desired level of trustability and that T is some trustability guarantee formula. Then an *optimal cost strategy for supporting T at L* is a minimum cost strategy that will support T at L.

Determination of Strategies for Optimal Trustability and Cost

In this section we assume that we have a finite set of possible methods. In the general case, where we have multiple methods and fault classes, trustability optimization problems are NP-hard [Howden and Huang 1993]. There are various heuristics, corresponding to simplified instances of the general case, that can be used to produce tractable optimization problems.

One possible heuristic corresponds to the idealized situation where, for each fault class, there is only one effective method, i.e., method with detectability > 0. A method may be effective for more than one fault class. If we have a situation where (1) we have a collection of potential methods and (2) more than one method is effective (i.e., detectability is nonzero) for one or more fault classes then we could use the following heuristic. For each fault class F_j, $1 \le j \le s$, we identify a method $M_{m(j)}$ for which $D_{m(j),j}$ is is maximal. We then generate modified detectability figures $D'_{i,j}$ for methods M_i and fault classes F_j, as follows. For $i = m(j)$ set $D'_{i,j} = D_{i,j}$, and for $i \ne m(j)$ set $D'_{i,j} = 0$, $1 \le i \le r$, $1 \le j \le s$.

If the detectability matrix is altered as described above, the general trustability guarantee formula can be replaced with the simplified, multiple method formula

$$T = 1 - \max_{(1 \le j \le s)} \left\{ f_j (1 - D'_{m(j),j})^{N_{m(j)}} \right\},$$

where $M_{m(j)}$ is the unique fault detection method having a nonzero detectability factor for fault class F_j.

Another possible heuristic corresponds to the idealized situation where each fault class is associated with a unique subset of methods that are effective for that fault class, and not for other fault classes. If we use the heuristic, for a general detectability matrix D we would define a new detectability matrix D' from D as follows. For each fault class j, identify a unique subset set(j) of the methods. For all indices i in set(j) let $D'_{i,j} = D_{i,j}$, and for all indices i not in set(j) let $D'_{i,j} = 0$. This heuristic will not result in a simpler trustability guarantee formula, but it will result in a tractable optimization problem.

The following theorems describe methods for determining an optimal strategy and cost bound for the first of the two idealized problems described above. Similar theorems can also be proved for the second of the simplified problems, in which each fault class is associated with a unique subset of methods.

In the following, we assume the existence of a detectability matrix having the properties of the first of the two tractable, idealized problems described above.

THEOREM 2.2 (OPTIMAL TRUSTABILITY). *Suppose that B is a strategy cost bound. Recall that F is the union of the fault classes under consideration. An optimal cost B strategy can be determined using the following simple algorithm:*

> Initially set all usage factors N_i, $1 \le i \le r$, to zero. Define a set of variables T_j, $1 \le j \le s$, called the trustability factors, and initialize $T_j = f_j$. Determine a value of j for which the trustability factor is largest, say k, and if $N_{m(k)}$ is 0 set it to 1. In addition, update T_k by multiplying it by $(1 - D_{m(k),k})$. If there is more than one value of j for which the trustability factor is largest, choose one arbitrarily. Continue this process while the sum of the cost factors $C_i(p)N_i$, $1 \le i \le r$, is less than the cost allocation bound B.
>
> In the above process, if the maximum trustability factor T_k at some stage is associated with a method that has already been used (i.e., its usage factor is already set to 1), if the maximum trustability factor is associated with a detectability factor equal to 0 (corresponding to fault classes for which we have no method), or if the maximum trustability factor is itself zero (due to detectability factors equal to 1 and/or frequency factors equal to zero), then no further improvements in the trustability guarantee can be made, and the algorithm can halt without continuing on until the cost bound is reached. The algorithm can also halt if the maximum trustability factors are such that the cost bound will be exceeded before all of the methods associated with the maximum trustability factors could be applied.

PROOF. The optimal allocation of resources will involve choosing or not choosing each of the r methods M_i. We can prove the theorem by proving the following property P_m of the algorithm: the choices made by the algorithm up to and including the mth choice, $1 \le m$, also occur in some optimal choice. We prove P_m by induction on m.

At each stage of the algorithm, the value of the trustability bound computed with choices made up to that stage is referred to as the partially

computed trustability bound. Initially it is computed from the formula with the strategy factors N_i, $1 \leq i \leq r$, all set to zero.

Consider the case where $m = 1$. We show that if the algorithm terminates at this stage it terminates correctly. If it does not terminate, the choice it makes is part of an optimal strategy.

Suppose that D_k, the detectability factor associated with a maximal trustability factor T_k, is zero. Then we have the situation where no method can increase the partially computed trustability bound so that the algorithm can terminate, and trivially produces an optimal allocation in which no methods are applied. This is also true for the artificial case where the maximal trustability factor is zero, which in this case would require that all fault frequencies be zero.

Consider now the other cases where the method detectability factors associated with the maximal trustability factors T_k are nonzero, and the trustability factors are nonzero. There are two possibilities. One is that we have enough resources to allow an application of each of the methods associated with all such factors, and the other is that we do not. Consider the first case. From the definition for the trustability formula, we see that we must eventually make an application of each of these methods in order to increase trustability beyond that achieved before any methods are applied, so that any such allocation in the first step must be part of an optimal allocation. This is because the maximal frequency factors f_k determine the partially computed trustability bound when no tests have been allocated, and applications of other methods M_i for $i \neq k$ will only, at best, decrease the nonmaximal trustability factors associated with those methods.

In the case where there are not enough resources to apply all of the methods associated with the maximal factors, then the trustability bound will be determined by the value of the maximal factors. No set of method applications that does not exceed bounds can increase the initial partially computed trustability bound achieved with no applications, so that the algorithm can terminate.

The proof of the inductive case for $m > 1$ is similar to that for $m = 1$. □

A theorem can also be stated for the related problem of determining the minimum cost strategy for achieving some required level of trustability. The proof of the theorem is simple and is not included.

THEOREM 2.3 (OPTIMAL COST). *If there is some strategy that will support a trustability bound T for a program p, then the cost C of the optimal cost strategy for achieving T is*

$$C = \sum_{i=1}^{n} C_i(p) N_i,$$

where $C_i(p)$ is the cost of applying method M_i to p and where the N_i can be determined as follows. For each fault class F_j, $1 \leq j \leq s$, determine the minimum integer value of 0 or 1 for n_j, $1 \leq j \leq s$, for which

$$T \leq 1 - \left(f_j(1 - D_{m(j),j})^n j\right)$$

is satisfied. The assumed existence of a supporting strategy guarantees the existence of a value of n_j *satisfying these constraints. For each method* M_i, $1 \leq i \leq r$, *let* $N_i = \max_{(1 \leq j \leq s,\, m(j)=i)}\{n_j\}$. *In addition to giving the minimum cost, the usage factors* N_i *also give the optimal cost strategy.*

3. APPLICATION ISSUES

Integration of Methods

Many authors have suggested the integrated use of a collection of methods for verification and validation, since each may be useful for different kinds of faults. The trustability/detectability approach is a possible framework for integration. It allows for the inclusion of both analysis and testing methods, and of techniques that are used early in a development effort such as design validation, as well as methods that are applied to code. It suggests that we may incorporate both formal and informal methods and tailor an approach to a particular class of programs and/or class faults.

Empirical Fault Frequency and Detectability Factors

It may be argued that fault frequency factors will not be readily available. It is possible to use the method without fault frequency factors, but even the simplest kind of information can be useful. For example, in a given environment, with observations of the kinds of faults that we expect to occur, we might develop a set of methods which deterministically detects those kinds of faults. At some point we may observe that these methods will work "most of the time," which corresponds to assuming that the frequency and the occurrence of faults which lie outside the scope of the methods is bounded by some probability. In this situation, all that is required is the willingness to be specific about this assumption, by making an estimate of this bound. In practice, these are the kinds of factors that programmers and maintainers use in making decisions about the choice and use of methods, and the requirement that such an estimate be made does not seem excessive. In any case, even when it is either impossible or unacceptable to make such an estimate, the trustability framework can be used to compute the frequency bound that is necessary for a certain level of trustability to be assumed.

It may also be argued that we will, in general, not know the detectability for probabilistic methods whose probabilistic factor depends on sampling over a program space. For example, it may seem unreasonable or imprecise to use empirical data about the effectiveness of methods gathered over multiple projects.

We first note that, in general, people will use methods that have been found to work for problems which are most likely to occur, so that the trustability model seems to be a good qualitative characterization of common practice. Its advantage is that it identifies estimates that would have to be made to permit quantification, and identifies valuable data whose collection should be part of any mature software development process. As in the case of fault frequency data, even in those cases where data is not available, it is

useful in allowing us to consider "what if" questions. For example, we can ask "if we have the following trustability requirement, then what are the minimal detectability factors that we have to assume for the following set of methods in order to achieve that trustability level?" or "if we make the following assumptions about method detectability levels and costs, what is the maximal level of trustability that can be achieved for a certain allowable maximal cost?"

Empirical data is used in a variety of software engineering metrics. It is, for example, the basis of all cost estimation models in which costs of previous, similar projects are used to predict costs of future projects. It is also used in standard reliability engineering models, such as those described by Musa et al. [1990]. In this case, past experience is used to justify the use of parameterized formulae which predict mean time between failures on the basis of the observed behavior of a program. The use of empirical information for trustability estimation is consistent with these uses of empirical data in other areas of software engineering. In this article we have described a very simple version of the trustability paradigm. Accurate empirical estimates may require more sophisticated measures of fault frequency or detectability that are more sensitive to a program or system under analysis. We briefly describe several possible alternatives here. Additional possibilities can be suggested that are based on research in defect classification such as that described by Chillarege et al. [1992].

Testability and Detectability. Approaches to the estimation of detectability factors may be customized for particular programs using the testability approach [Voas 1992; Voas and Miller 1992]. In the testability approach, a program p is seeded with faults, and then a method of interest is applied to determine its effectiveness in finding those faults. This information is used to provide a more accurate estimate of the effectiveness of a method M for p than would be available from estimates constructed entirely from information about the discovery of faults by M in programs other than p. When a standard method, such as random testing, is applied to a given program, the effectiveness of the method can be associated with the program and is referred to as the program's *testability* [Voas 1992]. A highly testable program is one for which the seeded errors are easily found and for which we then conclude that the probability of finding a fault when one is present is also high.

Parameterized Fault Frequency and Detectability. In our simple trustability model we can use program properties when estimating fault frequencies and detectability factors since we can include the consideration of classes of programs and classes of faults. In the parameterized approach we include measurable properties of individual programs in the detectability formula. For example, program complexity, using cyclomatic numbers, might be found to have an effect on fault frequency or detectability, and through empirical studies that correlate method effectiveness and complexity, we may be able to construct formulae that take such properties into account.

Process Feedback Fault Frequency. In this approach, formulae are used to incorporate factors into fault frequency formulae that are derived during the fault detection process. For example, it might be observed that the more faults that a module is discovered to have, the more undiscovered faults are likely to remain. It may be possible to construct fault frequency formulae which have the number of detected faults as a parameter.

Process Feedback Detectability. We may be able to construct detectability formulae that factor in the expenditure of resources. For example, suppose that inspections are used to detect faults. In the simple model of trustability described in the earlier sections of the article, we would have some stopping rule, such as "keep applying inspections until no faults are found." On the basis of experience, we would estimate the probability that such a process would find all faults, when faults are present. More-elaborate measures of detectability might factor in the number of inspections that were needed before a fault-free inspection occurred. If we use a stopping rule that requires repeated successful inspections, it might also have a parameter corresponding to the number of successful repeated inspections.

Fault Repair and Retest

Suppose that we use probabilistic detectability factors that are based on sampling from a program space. If we use a method that has such a detectability factor, apply it to a program, and then repair the program if a fault is found, it may seem that we can no longer use the detectability factor in reapplications of the method since the program under analysis is no longer a "random" sample from the program space. This problem is solved, as indicated above, by assuming that methods are associated with stopping rules, and that detectability measures the probability of finding a fault when the method is successively reapplied until the stopping criteria is satisfied. In the case where multiple methods are applied to the same fault class, or to nonindependent classes, we need either to assume that they are applied in parallel, or that detectability figures are constructed relative to a fixed process in which methods are applied in a given sequence. In this case, the detectability figure for a later method is measured relative to programs inherited from earlier methods.

It is noted that in the case where different methods are used for different, independent fault classes, then we can assume that the application of one method to a program, and the possible consequent modification of that program, will not decrease the detectability factor for other methods, and the problem described above will not occur.

Similar remarks can be made about fault frequencies. The frequency of a fault is the probability that a program (from the program sample space) will have a fault. If we apply a method M_2 to a program after the application of method M_1, we will need to use a frequency factor for the application of M_2 that corresponds to the probability of a fault occurring in a program that was selected from the program space and then analyzed (and possibly modified) using M_1. Again, if fault classes are disjoint and independent, we can assume

(1) that fixing a fault from one class does not change the frequency factor for faults for some other class and (2) that fault repair does not cause a problem.

Trustability and Operational Reliability

Operational reliability refers to the probability that a program will fail when it is executed over its operational distribution. Trustability can be viewed as an extreme case of operational dependability. Suppose that p is a program. Define a set of faults that will cause p to fail with probability larger than d, when the program is executed over its operational distribution, to be a *frequent-failure fault set* with *failure density factor* d. Reliability considerations demand that we have a certain level of confidence that the set of remaining faults in a program is not a frequent-failure fault set. If we have trustability T for faults of type F, then we can be at least T confident in the absence of a frequent-failure fault set of type F for any density factor d, $d \geq 0$, i.e., we can have confidence T that the failure density for p, due to faults of type F, is less than d for all $d \geq 0$.

We can combine the use of "absolute" trustability measurement methods with operational reliability measurement as follows. Suppose that we wish to conclude for a program p, with confidence T, that the failure density caused by faults of type F is less than some density bound d. Assume that we have a method or set of methods M for detecting faults of type F, and suppose that we can use M on p to establish trustability

$$T \geq 1 - x$$

for F. This means that when we use M, the probability of our making a mistake in asserting that p is free of faults of type F is less than x.

Suppose that we run N random tests on p, selected using the operational distribution, and see no failures of type F. On the basis of this event, we assume that we can say with confidence at least

$$1 - (1 - d)^N$$

that the failure density for p is less than d. The probability of incorrectly making this statement on the basis of the outcome of random testing, when it is false, is at most

$$(1 - d)^N.$$

Suppose that we now combine M with random testing. In the combined approach we will say that the failure density of a program is less than d if M indicates that is free of faults and if random testing is failure free. The probability of our making this statement when it is false is equal to the probability of coming to a false conclusion for both M and random testing, which is at most

$$x(1 - d)^N$$

so that, in the combined method, we can have confidence of at least

$$1 - x(1 - d)^N$$

that the failure density is less than d.

This formula can be used to determine the number of (failure-free) tests that are needed to guarantee a reliability level d. From this formula we see that trustability-oriented methods can be used to establish a reliability base-line, from which operational testing can be used to ensure, with fewer tests than would otherwise be required, a specified level of operational reliability. The trustability model allows us to consider ways in which testing may be made more efficient, through the use of such reliability base-lines.

Our technique for including operational dependability in the trustability model is based on ideas found in the simple reliability model mentioned earlier, in which we establish confidence in bounds on failure densities [Howden 1987]. In that approach, if we run a set of N tests that are selected over a program's operational distribution, and see no failures, then we say that we have confidence $1 - (1 - B)^N$ that the failure density is bounded above by B, since the risk of stating that the failure density is less than B is the probability of seeing N failure-free tests when in fact the failure density bound is larger than B. The number of tests needed to give a certain level of trustability for a given bound can be easily determined. The confidence-based model for bounds on failure densities has been recently discussed, along with several refinements and with work on techniques for describing operational distributions, in Woit [1993]. More-sophisticated reliability models, such as those that involve mean time between failure, will correspond to more-sophisticated ways of measuring trustability and of incorporating operational dependability into trustability measurement.

Trustability and the Software Process

In the previous sections, we focused on detectability factors for program analysis methods and associated detectability measures with program and fault classes. In general, we will need also to associate them with a software process. For example, the requirements and design methods that are used during the development of a program will affect the space from which the program is "selected" and hence the detectability factors for program analysis methods that are estimated over that space.

We need to consider methods for analyzing other products, such as designs and requirements. Suppose that we define a design fault in a program as one which can be traced back to a flaw in the design. One of the methods for detecting such faults will be design analysis. We cannot assume that design analysis is carried out in parallel with, and independently of, program analysis since it will be carried out before programming begins. One approach is to assume that development occurs in stages. In one stage, for example, we "select" a design that will have faults with certain frequencies, and for which evaluation methods will have certain detectability factors. In another, we "select" a program and assume similar information. As in the discussion of fault repair we could assume that empirical parameters like fault frequency and empirical detectability are determined relative to a process in which certain kinds of products are generated in different stages, and certain kinds of evaluation methods used in those stages.

Examples of the Application of the Trustability Model

The following simple examples illustrate ways in which the trustability model might be used. We first begin by showing how trustability can be related to the use of branch coverage, one of the most commonly used testing techniques.

Example 3.1 *Complete Coverage and Trustability.* Suppose that some form of test coverage method M is to be used, such as branch coverage. Let F be the set of faults whose presence in a program is guaranteed to be revealed if the program is tested over any set of tests which causes 100% coverage. Assume that we test a program, repair faults, and retest until all faults that can be found by a 100% coverage set are removed. Note that it is not necessary to reapply the method with different 100% coverage test sets, since F consists of faults guaranteed to be revealed by any 100% test set. Let F' be the complement of this set, and let f' be the frequency of occurrence of faults in F'. F' corresponds to the occurrence of programs which contain faults that are not (necessarily) revealed by a 100% coverage test set.

Now the detectability of the coverage-oriented testing method is 1 for the fault class F. The class of remaining faults F' can be thought of as being associated with the use of the null method, having detectability 0. The frequency of F' is f'. Applying the formulae given above, we find that we can have trustability of at least $1 - f'$ in a program that is analyzed using M.

Example 3.2 *Using Coverage to Make Statistical Testing More Efficient.* Suppose that it is desired to attain a 99% level of confidence in a 1% bound on the failure density for a program. In the simple reliability model referenced above in the section on trustability and operational reliability, we would need to run N failure-free tests, where N is defined by

$$0.99 = 1 - (1 - 0.01)^N.$$

From this formula we can see that N would need to be at least 461.

Suppose that, based on our experience with the combined use of informal functional and branch testing for a class of programs developed in the given environment, we estimate that we can be 90% confident that there are no faults of any kind. Then we can use the following formula to determine the additional number of acceptance tests needed to raise our confidence level to 0.99, for absence of faults that would cause a failure to occur with probability greater than 0.01

$$0.99 = 1 - (1 - 0.9)(1 - 0.01)^M = 1 - 0.1(1 - 0.01)^M.$$

This will require that M be at least 230, or only half as large as when random testing is used by itself.

Example 3.3 *Combinations of Life Cycle Methods.* In this speculative, artificial example, we illustrate how the trustability model might be used to combine trustability information from different phases of a software development process. The trustability calculations used here do not correspond

exactly to the trustability guarantee formulae of the earlier sections, but are derived directly from them.

We assume that a software company keeps track of problems that occur and tracks them to their source. Three broad classes of process-oriented fault classes are used: requirements, design, and programming. Faults tracked to requirements sources are generally functional in nature; design faults are interface or algorithm design problems; and programming faults are statement oriented but may include other problems such as initialization.

Within the class of programming faults, there is a further, method-oriented classification into mutation and nonmutation faults.

It is observed that there is a 0.01 chance that a requirements fault will not be detected in the requirements analysis phase, a 0.005 chance that a design fault will not be detected during design, and a 0.5 change that a requirements fault not detected during requirements will also not be detected during design. Since there are almost always both requirements and design faults (that usually get corrected) the fault frequency for these kinds of faults is 1. Programming faults are also common, but they are divided into two subclasses. It is observed that there is a 0.3 chance that a program will contain mutation faults and that for these faults mutation testing is 100% effective. For nonmutation faults, mutation testing has been found to be 0.7 effective, through some apparent coupling process between mutation and nonmutation faults. In addition to mutation testing, functional testing is used. It appears to be 0.98 effective for those faults not found by mutation testing. In addition, the combination of mutation and functional testing has been found to be 0.5 effective for requirements or design analysis, and also 0.5 effective for design faults not found in design analysis.

The information in Table I can be used to compute trustability, as follows. Trustability

$$
\begin{aligned}
&= 1 - \max\{(1 - 0.99)(1 - 0.5)(1 - 0.5), (1 - 0.995)(1 - 0.5), \\
&\quad \max\{0.3(1 - 1), 0.7(1 - 0.7)(1 - 0.98)\}\} \\
&= 1 - \max\{0.0025, 0.0025, 0.0042\} = 0.9958.
\end{aligned}
$$

In this case, the trustability figure is dominated by the effectiveness of the methods for programming faults.

Related Work

Previous work on detectability comparison for random and partition testing is described in the papers of Hamlet and Taylor [1990], Weyuker and Jeng [1991], and Frankl and Weyuker [1993].

Numerous papers have been published on reliability estimation and on statistical testing of software, e.g., Musa et al. [1990]. However, our primary concern here is the measurement of trustability, independently of whether the methods that are used are statistical or not, and for brevity we only cite very closely related work.

The two main ideas in the trustability model are the use of hypothesis testing for confidence estimation and the use of conditional probabilities that

Table I.

Fault Class	Probability of Occurrence	Method(s) for Detection	Detectability
requirements	1.0	requirements analysis	0.99
		design analysis*	0.5
		program testing*	0.5
design	1.0	design analysis	0.995
		programming*	0.5
programming (mutation)	0.3	mutation	1.0
programming (nonmutation)	0.7	mutation	0.7
		functional*	0.98

*These effectiveness factors are relative to faults not found by earlier methods.

indicate the effectiveness of methods. The first idea was used in Howden [1987] to estimate confidence in failure density bounds and was referred to as a measure of "probable correctness." The same term was used in Hamlet [1987], where confidence-based program dependability is also discussed. In this case, the goal was to derive confidence through sampling over the textual space for a program, rather than over its operational input domain.

The use of the confidence based approach to dependability estimation has also occurred more recently in, for example, Hamlet and Voas [1993] and Howden [1993]. The difficulty with all of the earlier, test-oriented applications of the confidence approach is the large numbers of tests needed for even a modest level of confidence. This problem is solved by the second idea, the use of conditional probabilities, but at the expense of having to make assumptions about a method's fault-revealing effectiveness.

The concept of detectability in the trustability model is similar to that of testability, which originated with hardware testing, e.g., Seth et al. [1990]. Hardware testability is the probability that a test of a circuit will reveal a fault, if the circuit contains a fault. Voas and Miller were the first to associate testability with software and to investigate different applications of the idea in depth [Voas 1992; Voas and Miller 1992].

The difference between testability (for a program) and detectability (for a method) is slight. However, the use of detectability has a significant effect on the kinds of analysis it enables. It makes it possible to consider combinations of testing and analysis methods and to incorporate fault class frequencies in trustability formulae. These kinds of results do not occur in the work of Voas.

Voas has been primarily concerned with the testability of an individual program, i.e., whether or not it is the kind of program that could easily "hide" faults when tested using some method. He also considers what we have defined to be trustability, but indirectly through the use of what he calls the "squeeze play" [Voas and Miller 1992], and only through the use of randomized testing. The squeeze play can be defined as follows. Suppose that we know the trustability t of a program with respect to some random testing method. Suppose that we apply the same testing method to establish an upper bound estimate f, with confidence C, on the program's failure density.

If $f < t$, then we can say with confidence C that the program is free of faults, since if it had a fault, t and f would be equal.

One of the principal ideas in the testability work is the development of methods for estimating the detectability of a method for a particular program. The approach, mentioned above, is to use seeded faults and to extrapolate to more-general fault classes. Although this may be more expensive than to use estimates based on previous applications of a method to different programs, it may be more accurate.

The words "trustability" for confidence-based analysis and "detectability" for method effectiveness were coined by the authors and first appeared in Howden and Huang [1993]. The use of both detectability factors and fault frequencies in computing trustability is original, as are the trustability guarantee formulae, the identification of different kinds of detectability factors, the relationship between trustability and reliability, the concept of a frequent failure fault set, the idea of combining methods within a single trustability framework, the formulation of the trustability optimization problems, and the optimization methods described in the theorems. A limited part of this work was previously described in less general form by Howden [1993] and Howden and Huang [1993].

CONCLUSIONS

The concepts of detectability and trustability provide a framework both for a theoretical analysis of important issues in testing and for characterizing the ways in which programmers make informal, practical decisions when attempting to prevent and detect faults in programs.

The trustability model identifies necessary conditions for levels of confidence in the absence of faults in a program. In those cases where numeric parameters associated with those conditions are not known, the model is useful in identifying assumptions that have to be made in order to establish levels of confidence. It identifies the kinds of measurements that can be made both across different projects and within a single project.

The theory and the examples given in the article indicate that in order to establish high levels of dependability, we need to rely on either low fault occurrence frequency, high detectability, or the use of methods whose effectiveness has a multiplicative effect. The latter can be achieved with cumulative methods, with methods having relative detectability factors, and with the joint use of empirical and probabilistic methods.

The trustability model which was presented is general, and it is possible to use restricted parts of it. The general model was created to allow a diverse set of situations to be accommodated within a single framework.

A variety of topics for further research and development can be identified. They include, for example, the use of the concept of detectability to characterize additional situations in which one method is more effective than another. We can also investigate the development of a refined trustability model, one in which methods are not identified with a single detectability coefficient, but with more-complex measures such as those suggested above in the section on

empirical detectability factors. Finally, research on methods for the determination of detectability factors can be pursued.

REFERENCES

CHILLAREGE, R., BHANDARI, I., CHAAR, J., HALLIDAY, M., MOEBUS, D., RAY, B., AND WONG, M.-Y. 1992. Orthogonal defect classification—A concept for in-process measurements. *IEEE Trans. Softw. Eng. 18*, 11 (Nov.), 943–946.

CLARKE, L., PODGURSKI, A., RICHARDSON, D., AND ZEIL, S. 1985. A comparison of data flow path selection criteria. In *Proceedings of 8th International Conference on Software Engineering* (Tokyo, Aug.). IEEE, New York, 244–251.

DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. 1978. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (Apr.), 34–41.

DURAN J. W. AND NTAFOS, S. C. 1981. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego). IEEE, New York, 179–183.

DURAN, J. W. AND NTAFOS, S. C. 1984. An evaluation of random testing. *IEEE Trans. Softw. Eng. SE-10*, 4 (July), 438–444.

FRANKL, P. G. AND WEYUKER, E J. 1993 A formal analysis of the fault-detecting ability of testing methods. *IEEE Trans. Softw. Eng. 19*, 3 (Mar.), 202–213.

FOSTER, K. A. 1980. Error sensitive test case analysis. *IEEE Trans. Softw. Eng. SE-6*, 3 (May), 258–264.

GANNON, J., MCMULLIN, P., AND HAMLET, R. 1981. Data-abstraction, implementation, specification and testing. *ACM Trans. Program. Lang. Syst. 3*, 3 (July), 211–223.

HAMLET, R. G. 1977a. Testing programs with finite sets of data. *Comput. J. 20*, 3 (Aug.), 232–237.

HAMLET, R. G. 1977b. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng. SE-3*, 4 (July), 279–290.

HAMLET, R. G. 1987. Probable correctness theory. *Inf. Process. Lett. 25*, 1 (Apr.).

HAMLET, D. AND TAYLOR, R. 1990. Partition testing does not inspire confidence. *IEEE Trans. Softw. Eng. SE-16*, 12 (Dec.), 1402–1411.

HAMLET, D. AND VOAS, G. 1993. Faults on its sleeve, amplifying software reliability testing. In *Proceedings, 1993 International Symposium on Software Testing and Analysis* (Monterey, Calif., June). ACM, New York, 89–98.

HOWDEN, W. E. 1982. Weak mutation testing and the completeness of program test sets. *IEEE Trans. Softw. Eng. SE-8*, 4 (July), 371–379.

HOWDEN, W. E. 1985. The theory and practice of functional testing. *Software 2*, 5 (Sept.), 6–17.

HOWDEN, W. E. 1987. *Functional Program Testing and Analysis*. McGraw-Hill, New York.

HOWDEN, W. E. 1993. Foundational issues in software testing and analysis In *Proceedings, 6th International Software Quality Week* (San Francisco, Calif., June). SRI International, Menlo Park, Calif.

HOWDEN, W. E. AND HUANG, Y. 1993. Theoretical foundations for formal and informal confidence based statistical dependability estimation. CSE Tech. Rep., Univ. of California at San Diego.

LASKI, J. W. AND KOREL, B. 1983. A data flow oriented program testing strategy. *IEEE Trans. Softw. Eng. SE-9*, 3, 347–354.

MARICK, B. 1992. The craft of software testing. In *Pacific Northwest Software Quality Conference* (Portland, Ore., Sept.). Tutorial notes.

MUSA, J., IANNNINO, A., AND OKUMOTO, K. 1990. *Software Reliability*. McGraw Hill, New York.

NTAFOS, S. C. 1988. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng. SE-14*, 6 (June), 868–874.

RAPPS, S. AND WEYUKER, E. J. 1985. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng. SE-11*, 4 (Apr.), 367–375.

RICHARDSON, D AND CLARKE, L. 1981 A partition analysis method to increase program reliability In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, Calif., Mar.). IEEE, New York, 244–253.

SETH, S. C., AGRAWAL, V. D., AND FARHAT, H. A. 1990. A statistical theory of digital circuit testability. *IEEE Trans Comput 39*, 4 (Apr.), 582–586.

VOAS, J. M. 1992. PIE: A dynamic failure-based technique. *IEEE Trans Softw Eng. 18*, 8 (Aug.), 717–727.

VOAS, J M. AND MILLER, K 1992. Improving the software development process using testability research. In *Proceedings 3rd ISSRE* (Research Triangle Park, Oct.). IEEE, New York, 114–121

WEYUKER, E. J. AND JENG, B. C. 1991. Analyzing partition testing strategies. *IEEE Trans. Softw. Eng. 17*, 7 (July), 703–711.

WHITE, L. J. AND COHEN, E. I. 1980. A domain strategy for computer program testing. *IEEE Trans. Softw Eng SE-6*, 3 (May), 247–257.

WOIT, D. M. 1993. Estimating software reliability with hypothesis testing. CRL, McMaster Univ. Apr.

WOODWARD, M R., HEDLY, D., AND HENNELL, M A. 1980. Experience with path analysis and testing of programs. *IEEE Trans. Softw. Eng. SE-6*, 3 (May), 278–286

Received May 1993; revised December 1993; accepted November 1994