Evaluating Testing Methods by Delivered Reliability[*]

Phyllis Frankl[†]

CIS Dept.
Polytechnic Univ.
6 Metrotech Center
Brooklyn, NY 11201
USA
phyllis@morph.poly.edu

Dick Hamlet

Dept. of CS
Portland State Univ.
PO Box 751
Portland, OR 97207
USA
hamlet@cs.pdx.edu

Bev Littlewood
Lorenzo Strigini [‡]

Centre for Software Reliability
City University
Northampton Square
London EC1V OHB
UK
{b.littlewood,strigini}@csr.city.ac.uk.

[*] This work was carried out in part during visits of Hamlet and Frankl to the Centre for Software Reliability, with support from EPSRC visiting fellowship grants GR/K68134 and GR/L00445.

[†] Supported in part by NSF grant CCR-9206910.

[‡] Littlewood and Strigini were funded in part by the European Commission via the ESPRIT Long Term Research Project 20072 "DeVa".

ABSTRACT

There are two main goals in testing software: (1) To achieve adequate quality (*debug testing*); the objective is to probe the software for defects so that these can be removed. (2) To assess existing quality (*operational testing*); the objective is to gain confidence that the software is reliable. The names are arbitrary, and most testing techniques address both goals to some degree. However, debug methods tend to ignore random selection of test data from an operational profile, while for operational methods this selection is all-important. Debug methods are thought, without any real proof, to be good at uncovering defects so that these can be repaired, but having done so they do not provide a technically defensible assessment of the reliability that results. On the other hand, operational methods provide accurate assessment, but may not be as useful for achieving reliability.

This paper examines the relationship between the two testing goals, using a probabilistic analysis. We define simple models of programs and their testing, and try to answer theoretically the question of how to attain program reliability: Is it better to test by probing for defects as in debug testing, or to assess reliability directly as in operational testing, uncovering defects by accident, so to speak? There is no simple answer, of course.

Testing methods are compared in a model where program failures are detected and the software changed to eliminate them. The "better" method delivers higher reliability after all test failures have been eliminated. This comparison extends previous work, where the measure was the probability of detecting a failure. Revealing special cases are exhibited in which each kind of testing is superior. Preliminary analysis of the distribution of the delivered reliability indicates that even simple models have unusual statistical properties, suggesting caution in interpreting theoretical comparisons.

Keywords

Reliability, debugging, software testing, statistical testing theory

1 INTRODUCTION – RELIABILITY VS. DEBUGGING

There are two main goals in testing software. On the one hand, testing can be seen as a means of achieving reliability: here the objective is to probe the software for bugs[1] so that these can be removed and its reliability thus improved. Alternatively, testing can be seen as a means of gaining confidence that the software is sufficiently reliable for its intended purpose: here the objective is reliability *evaluation*.

We begin by taking the point of view of a developer who tests to find and correct bugs and improve the delivered software. A systematic testing method includes a criterion for selecting test cases and a criterion for deciding when to stop testing. Most common approaches to systematic testing are directed at finding as many bugs as possible, by either sampling all

[1]We deliberately use this informal term in this introductory discussion: in later sections we shall discuss the problems in finding a formal interpretation of the notion of "fault."

situations likely to produce failures (e.g., methods informed by code coverage or specification coverage criteria), or concentrating on situations that are considered *most likely* to do so (e.g., stress testing or boundary testing methods). The choice among such testing methods will depend on hypotheses about the likely types and distributions of bugs at the point in the software development process when testing is applied. We shall call all these approaches, collectively, "debug testing."

A completely different approach is "operational testing," where the software is subjected to the same statistical distribution of inputs that is expected in operation. Instead of actively looking for failures, the tester in this case waits for failures to surface spontaneously, so to speak.

In comparing the relative advantages of operational testing and debug testing, important points are:

- Debug testing may be more effective at finding bugs (provided the intuitions that drive it are realistic), but if it uncovers many failures that occur with negligible rates during actual operation, it will waste test and repair efforts without appreciably improving the software. Operational testing, on the other hand, will naturally tend to uncover earlier those failures that are most likely in actual operation, thus directing efforts at fixing the most important[2] bugs.

- The cost of testing with the various approaches varies widely with the characteristics of the program and of the application problem. These latter determine the relative costs of the components of the testing process (generating test cases, executing the software, checking for correct results), in particular through the extent to which they can be automated.

- The fault-finding effectiveness of a debug testing method hinges on whether the tester's assumptions about bugs represent reality; for operational testing to deliver on its promise of better use of resources, it is necessary for the testing profile to be truly representative of operational use.

- Operational testing is attractive because it offers a basis for reliability assessment, so that the developer can have not only the assurance of having tried to improve the software, but also an estimate of the reliability actually achieved.

Previous comparisons of the effectiveness of testing techniques have used the failure-finding probability, the probability that a testset will detect *at least one failure*, as a measure of effectiveness. This measure was used in simulations comparing "partition-testing" techniques to random testing by Duran and Ntafos [8], and Hamlet and Taylor [12]; in analytical treatments by Weyuker and Jeng [16], and Chen and Yu [4]; in analytical comparisons of various testing techniques by Frankl and Weyuker [10]; and in experimental comparisons by Frankl and Weiss [9], and Mathur and Wong [26].

[2]Notice that, throughout the paper, we treat the "importance" of a bug solely in terms of its contribution to unreliability. We do not take any account of the consequences of failure. In practice, of course, these can vary greatly from one bug to another. The results of the paper could, of course, be applied to suitably defined subclasses of failures, representing particular levels of severity of consequences.

Failure-finding probability may be a good measure for evaluating test data adequacy criteria (stopping criteria). The best stopping criterion may be the one that is most likely to detect at least one failure, for then when it detects nothing, the tester has the most confidence that nothing has been missed. However, failure-finding probability sheds little light on how the detection and elimination of failures during the testing process affects the delivered reliability. Different failures may make vastly different contributions to the (un)reliability of the program. Thus, testing with a technique that readily detects "small" faults, may result in a less reliable program than would testing with a technique that less readily detects some "large" faults. Examples of this situation in which failure-finding probability and better reliability do not go together are given in section 3.5, Multiple Failure Regions, Debugging with Subdomains, below.

Several papers have considered the expected number of failures during test as a measure of effectiveness [10, 5]. Chen and Yu [5] argue that, although ideally one would like to assess the number of faults detected, and although there is no general relation between number of failures and number of faults, if more failures occur during test, it will be easier to find and remove more faults. Several experiments have compared the number of faults detected by different testing techniques [14, 26], but this issue has not been addressed analytically. In addition several reliability growth models [15, 19]are based on the number of faults detected.

This paper studies testing effectiveness based on the reliability of a program after it is tested. This measure is used to compare debug testing to operational testing, exploring circumstances under which each technique is likely to yield superior reliability. Li and Malaiya [18] consider a similar question, but in the narrower context of altering the test profile to emphasize particular subdomains of the program. Their model assumes that failures that occur in these subdomains have a fixed detection probability.

Our model is more realistic: the probability of a test case detecting a fault depends not only on the subdomain hit by the test case, but on the way test cases are selected in the subdomain. We also model a wider range of testing methods: operational testing, and two forms of debug testing, one driven by the consideration of subdomains in the input space, and one ignoring the subdomains. Last, we clarify the difficult problem of modeling "the fault" that is responsible for a failure.

1.1 The Debugger's Intuition

There is a deeply rooted belief among program testers and debuggers that the process of probing software for bugs is a cost-effective way of achieving sufficient reliability. That is, employing testing methods that are designed to expose failures is believed to be a better alternative than simulating normal operation and letting the failures appear. Indeed, the latter method is used by only a small minority of industrial organizations. This paper examines the validity of that belief. (Detailed definitions of "debug testing" and "operational testing" are given in sections below.)

The validity of testers' trust in debug testing is not an academic question. Software whose reliability must be high could be tested in a number of different ways, and because testing is expensive and time-consuming, developers and regulatory agencies would like to choose among alternatives, not use them all. Thus if debug testing is not effective, it should not be used at all. In particular, there is a currently popular position that can be paraphrased as follows:

> Reliable software can best be developed using formal methods. When properly applied, these methods eliminate at source those failures normally exposed at the unit and subsystem levels by debug testing. Therefore, unit debug testing should be reduced in favor of additional system-level random testing.

In the "Cleanroom" development methodology [6, 24], to give an extreme example, debug testing is generally not used at all, particularly by those doing the development. Apart from its ability to provide reliability estimates, it is argued that operational testing detects any remaining failures that could occur, with probabilities that are in proportion to their seriousness. However, experienced developers, say of flight-control software, are profoundly disturbed by the suggestion that they abandon debug testing. As an indication of the depth of traditional testers' reaction to this position, Beizer [2] has attacked Cleanroom as "lead[ing] to false confidence."

Attempts to support or refute beliefs about debug testing have been inconclusive:

Empirical studies. Case studies comparing software development methods are difficult to conduct. Attempts to establish a correlation between the degree of debug testing (usually measured by some structural "coverage" of unit tests) and the resulting software quality are at best preliminary [7, 13, 22, 9, 17]. On the other side, case studies using formal methods development show great variation, both in the care with which the method is defined and applied and in the results [11]. Neither side has any real claim to establishing its case.

Analysis of "partition testing." A number of theoretical studies have compared random testing with debug ("partition") testing [8, 12, 16, 25, 4, 5]. The original motivation for these studies was a belief that random testing might be a real alternative to partition testing for finding failures. However, no such conclusive result was obtained. Although random testing is a surprisingly good competitor for partition testing, it is seldom better, and scenarios can be constructed (although their frequency of occurrence in practice is unknown) in which partition testing is much better at failure exposure. Thus our question remains.

In this paper we take a new analytical approach to comparing debug testing with operational testing. This approach was devised to study theoretically the question of delivered reliability, without prejudice to the outcome of comparisons.

1.2 Analytical Approach

We believe that analytic, probabilistic methods are the best tools for studying software reliability. Basing an important choice on intuition, without much supporting evidence, is clearly dangerous. Analytical studies help by giving clear representations of the competing intuitive beliefs and of their actual implications, and also by indicating which empirical measurements could provide indirect evidence that, in a particular project and phase of development, a certain test method is best. We consider the situation in which software fails under test, then is changed so that the failure no longer occurs. We compare testing methods according to the probability that the corrected software will subsequently fail in operation (that is, the *delivered software reliability*). This measure is expressed as a random variable, and we mainly focus on its expected value, although the distribution is also of interest.

A simple program model is used; this simplifies the analysis, and focuses attention on the question of reliability. The testing-failure-fix process must also be abstracted and simplified for analysis. We believe that the notion of a software "fault" is central to this abstraction, and that a meaningful, formal treatment of "faults" is not available. Instead, we introduce the notion of a "failure region" of the input space, a set of failure points that is eliminated by a program change.

For our simple abstractions, we compare operational testing to debug testing, and present revealing special cases in which each technique yields better reliability after some failures are eliminated. For a single failure region, the results are similar to those obtained by analyzing the probability of finding a failure. But for multiple failure regions new phenomena are captured. For example, for some programs the testing technique that best finds failures may not lead to the best reliability, because it finds trivial problems with little operational impact.

1.3 Statistical Nature of Analysis

If methods of achieving reliability are to be assessed, probabilistic analysis must be used. Statistical questions must be framed and answered by calculation, to inform a debate that so far has little content beyond strongly held beliefs on both sides. The partition vs. random studies suggest that if we can frame the questions, a combination of mathematical analysis and simulation can answer them, with a marked improvement in fundamental understanding.

The questions here need to be posed in ways that acknowledge the inevitable underlying uncertainty. In other words, we need to be aware that probability and statistics are appropriate tools for expressing the problem. This means that our answers will inevitably be couched in these terms. Thus, for example, we shall not be able to make deterministic claims for the superiority of one testing regime over another. Instead, we shall be looking for evidence that one type of testing is likely to be superior to another, or that it is on average better.

Such observations involve us in some subtleties which may not be obvious to the unwary. For example, we may sometimes prefer a testing procedure that is inferior to another on average (e.g., in its ability to increase reliability most cost-effectively) if its efficacy shows less variation from one application to another. We might prefer the near certainty of a modest gain in reliability for a particular outlay from the first procedure, to a mere possibility of a high gain from the other. Such considerations will not loom large in the following, which we realize represents only the beginnings of an understanding of these issues, but they must be addressed in future work.

2 TERMINOLOGY AND ASSUMPTIONS

In formal work, it is important to have precise definitions and to explicitly state assumptions. In this preliminary work, these must be particularly simple.

2.1 Tests and Failures

A *test* or *test case* is a single value of program input, which enables a single execution of the program. A *testset* is a finite collection of tests. These definitions implicitly assume a simple programming context: a program with a pure-function semantics. The program is given a single input, it computes a single result and terminates. The result on another input in no way depends on prior calculations. In particular, if an input is repeated, the result is always the same. Although many programs do not behave in this manner, the relevant issues about reliability arise for pure-function programs.

This simple program model abstracts reality, but it is more general than it may appear. Real programs may have complex input tuples, and produce complex outputs. But we can imagine coding each tuple into a single value, so that the simplification to one input value is not a transgression in principle. Some interactive programs, programs that read and write permanent data, and real-time programs, do not fit the pure-function model. However, it is possible to treat these more complex programs as if they used testsets of independent inputs, at the cost of some artificiality. For example, an interactive or real-time program can be thought of as having artificial testsets whose members (single tests) are *sequences* of the real input elements, starting from some standard "reset" state. Each such sequence is one abstract input in the pure-function model.

Each program has a specification that is an input-output relation. That is, the specification S is a set of ordered input-output pairs describing allowed behavior. A program P *meets* its specification for input x if and only if (iff) the following is true: if $x \in dom(S)$ then on input x, P produces output y such that $(x, y) \in S$. When $x \notin dom(S)$, that is, when an input does not occur as any first element in the specification, the program may do anything, even fail to terminate, yet still meet the specification. S defines the input domain as well as behavior on that domain. Many real specifications can be recursively extended to be everywhere defined, by adding required "ERROR" responses; but some, notably involving unbounded searches with uncertain outcome, cannot be effectively extended.

A program P with specification S *fails* on input x iff P does not meet S at x. When a program fails, the event is called a *failure*, and the input responsible is a *failure point*. The program's *failure set* is the collection of all failure points. Hence a program that meets its specification has an empty failure set. The opposite of fails is *succeeds*; the opposite of a failure is a *success*; the complement of the failure set is the *success set*.

We assume that the specification of a program does not change during testing and corrective changes to the program.

2.2 So-called "Faults"

Program testing methods are often designed to find "faults." But it is a strong, unjustified assumption that "a fault" is an objective characteristic of a program. Although *fault* is an IEEE standard term for "bug" (or "defect," or "error"), this idea is not precise, and is difficult to make precise. The IEEE glossary states that a fault is the part of a source program that causes a failure. However appealing and necessary this intuitive idea may be, it has proved extremely difficult to define formally. The difficulty is that "faults" have no unique characterization. In practice, software fails for some testset, and is then changed so that it succeeds on that testset.

The (not necessarily true) assumption is made that the change does not introduce any new failures. The "fault" is then defined by the "fix," and is characterized, for example "wrong expression in an assignment", by what was changed. But the change is by no means unique. Literally an infinity of other changes would have produced the same effect.

Some fixes do appear to be unique and easily localized: for example, a wrong operand – perhaps a typo – in an expression. But "faults of omission" are common, and for these it is difficult for even reasonable programmers to agree on a fix. In addition, two changes that both fix a given set of failure points may differ in the remainder of their effects on program behavior. The complications of a "partial fix" that removes fewer failure points than it might have done, and a "least fix" that is in some textual way minimal for the effect it has, are extremely difficult to capture.

An operational method for identifying "the set of faults" in a program, as this term is commonly understood, might be as follows.

> Give the program to a debugging team to be tested and corrected until no more failures are detected, then analyze the history of program changes. Every change is motivated by a fault perceived by the debugging team in response to some failure(s). Some of these faults were introduced by the team itself by mistake. The others are the set of faults in the original program.

The problem with this method is that if we gave the same original program to a different debugging team, or even to the same team under different circumstances, we might end up identifying a different set of faults. So, any model that depends on a program having a uniquely identifiable set of faults, or even a unique number of faults, in order to predict how these may be removed by testing, cannot use this method to identify them.

Thus "the fault" is not a precise idea, and the usual intuitive meaning of the word cannot be used here.

On the other hand, "failure" is well defined, and so is a change in failure behavior resulting from a program change. Most of what we need to say can be phrased in these terms, as follows:

> A program change may alter the failure set; that is, the changed program's failure set will in general be different from that of the original program. A change is a *fix* for a collection of failure points F (the change *fixes* F) if it is conservative in the sense that (1) the failure set of the changed program no longer includes any member of F, (2) the failure set of the changed program is a subset of the original failure set.

Thus a fix for a set of failure points F may eliminate failure points outside F, but it may not introduce new failures[3].

[3]Our models only consider changes that are fixes, i.e., successful changes. We could have avoided this

In these terms, the closest we can come to speaking of a "fault" is to talk of a *failure region*, a collection of failure inputs that some change fixes exactly. Every change that does not introduce new failure points has such a region (if no more than the empty one). It is tempting to begin thinking of such a fix as the basis for defining "fault," but this would not satisfy the intuition behind the IEEE definition. We can hardly say that an elaborate change tailored to some failure region bears any relation to a mistake made by a programmer; nor does the failure region indicate or constrain a fix that might remove it.

We believe that one should try to avoid the term "fault" in discussing testing and the dependability of software. Thus one should say, "testing exposed a failure," not, "testing found a fault." One should say, "source change A led to a failure set strictly contained in the failure set resulting from change B," not, "A fixed more faults than B" (much less, "B didn't fix the bug, but A did"). Suppose a fix is found for a certain collection of failure points B_1, and another fix for other points B_2, which seem unrelated. However, a clever programmer then finds a completely different fix for $B_1 \cup B_2$ (and there is always such a fix, whatever arguments it causes among programmers). One should describe the situation in that neutral way, saying nothing about which are the "real bug(s)."

With the usual flawed assumption that each failure is due to one well defined "fault" in the program source, the process of testing and fixing a program appears to be affected by only two sources of uncertainty: which "faults" the testers will find and how effective their attempted fixes will be. (Perfect fixes are usually assumed.) Our contrary viewpoint recognizes three sources of uncertainty: which failure points will be found, which fixes the testers will try (hence which failure regions they expect to remove), and how effective the fixes will be (which failure regions will actually be removed).

The modeling in this paper uses the conventional assumption that all testers will react to a given observed failure with the same, successful fix. We wish to show how wide a spectrum of situations is possible, even under this restrictive assumption. However, we think that in many situations of interest, especially with highly reliable programs, this restrictive assumption is unrealistic, as the failure set may be determined by rare, complex patterns of program behavior.

Specifically, we will assume that all testers, upon observing a test failure, choose fixes that eliminate exactly the same failure region, irrespective of which test method they are using. We can thus talk of failure regions as characteristics of the program – as people usually talk about "faults" being characteristics of the program – rather than of the fixing process. This is a useful simplification in this initial analysis. The reason why it is unrealistic is that the way a debugger chose the test case that caused a failure may affect the debugger's guess about an underlying "fault" and thus the way he/she will proceed to fix the problem. Such "cues" may be beneficial or misleading depending on both the test method and the failure set of the program. So, following a failure on a given test case (or on test cases that appear equivalent to some tester) different testers may perceive the existence of different defects in

assumption by adding appropriate parameters to our models, describing the probability that fixes are completely or partially unsuccessful. In practice, we have avoided this, like other possible "improvements" to the models, to avoid an overwhelming number of degrees of freedom in the scenarios that can be modeled. When seeking insight into the effects of some specific factors – the fault-finding abilities of different testing strategies, in our case – it is better at first to avoid refinements that might obscure these effects in complex ways.

the code, and their changes may eliminate different sets of failure points, and even add new failure points.

We also assume that failure regions are disjoint, and all test failures are noticed (that is, there is a perfect oracle). So, each test failure deterministically causes one failure region to be removed.

2.3 Operational Testing

To define operational testing requires two main concepts: the operational profile that determines the likelihood of selection of the different points of the input domain, and an allocation of labels "ϕ" and "σ" (for failure and success) to the points.

The operational profile is a probability distribution Q over the input domain D, i.e., to each point is allocated a probability of selection, and these probabilities sum to one over the points of the domain. That is, $Q : D \to [0,1]$, and $\sum_{t \in D} Q(t) = 1$. Operational testing[4] then proceeds by independently selecting points from the input domain with these probabilities. In many applications, a point-by-point operational profile is far too detailed to obtain, and even a crude approximation requires considerable developer effort [21]. However, for our theoretical treatment, the profile Q is a central concept.

Informally, the operational profile can be thought of as characterizing the nature of the use to which the program is put, and will in general be determined by the system(s) (including people) that interact with the software. In itself it does not tell us about the reliability of the software. We need in addition that all points in the input domain have associated with them either a label ϕ (to indicate that such a point, when selected, results in a failure), or σ (for success). Define the indicator variable

$$\delta(t) = \begin{cases} 1 & \text{if } t \text{ has label } \phi \\ 0 & \text{if } t \text{ has label } \sigma \end{cases} .$$

Then the *failure probability* for a test point drawn randomly from the operational profile is

$$\theta = \sum_{t \in D} Q(t)\delta(t) = \sum_{t \in \text{failure set}} Q(t).$$

Of course, in practice we do not know what the labelings of the points in the input domain are: if we did, we could simply fix things without any testing! Thus estimation of θ will have to be statistical, and come from the results of a testset randomly selected from the operational profile. One simple approach would use the proportion of failures within such a sample of tests as an estimate of θ.

The *reliability* of the program is then the probability of it surviving N executions on inputs drawn from the operational profile:

$$R(N) = (1 - \theta)^N.$$

[4]Operational testing is sometimes called random testing, but the latter term is wider and could be used for statistical testing from *any* distribution, rather than one, as is intended here, that reflects operational use. Indeed, random testing is often taken to mean uniform random testing, where all points in the input domain are equally likely to be selected.

The probability of failure on a randomly selected input, and thus the reliability of a program, is determined partly by the probabilities of selection of the different points in the input domain (the operational profile), and partly by the way in which these points are labeled ϕ and σ. Operational testing only takes account of the operational profile in the selection of tests. Debug testing, on the other hand, seems mainly to take account of the labeling: it seems implicit that testers have knowledge (or at least believe they have) of which points in the input space are more likely to have ϕ labels, and testers give such points a greater chance of being selected than in operational testing; the points that are believed to be more likely to be σ points are given correspondingly smaller chances of selection.

There is a subtle interplay between the two contributions to (un)reliability, and how the two testing approaches treat them. Consider a single point in the input domain, x_i, with probability of selection in operation p_i. The operational tester says "I don't know anything about the chance that x_i will have label ϕ, so I will select it with probability p_i; that way, if it has a label ϕ, I at least have a chance of detecting it that is proportional to its contribution to the unreliability of the program." The debug tester says "I don't know anything about the operational profile (or if I do I don't care!), but I do have a good intuition about which points are likely to cause failure, and x_i is one of them, so I will select it with high probability and thus have a good chance of improving the reliability."

2.4 "Debug" Testing

Whereas the operational tester focuses attention on developing an input profile that closely approximates the distribution that the software will encounter in the field, the debug tester seeks to develop a distribution that will be likely to find the points labeled "ϕ". A perfect debug testing strategy would assign probability zero to all points labeled "σ". In practice, debug testers develop distributions based on heuristics that they hope will give high selection probabilities to failure points. Many such heuristics divide the program's input domain into (possibly overlapping) regions called *subdomains* and require that at least $T_i \geq 1$ test cases be drawn from the i^{th} subdomain. (The earlier discussion of "partition testing" refers to subdomains that do not overlap[5].)

In a number of practical testing methods, the subdomains are based on analysis of the specification (*specification-based* or *black-box* methods). The primary such method is *functional* testing, in which a number of program "functions" are identified (roughly, things the software should do), and the subdomains are defined as those inputs that result in its doing each thing. A second important collection of debug-testing methods are *program-based*, or *structural*, or *clear-box* methods. The archetype structural testing method is "statement testing," in which the subdomains correspond to the execution of individual program statements, and a test point selected from each and every subdomain forces every program statement to have been executed. These statement-testing subdomains therefore overlap, as do the subdomains of most structural testing methods and of many functional methods.

[5] "Partition" is a good word to avoid, not only because it technically does not include the important practical case of overlapping subdomains, but also because in common parlance "partitions" refer to the subdomains themselves, while in the technical mathematical usage "partition" refers to the relation that induces a set of equivalence classes (the subdomains).

Subdomains may be used either 1) as a means of evaluating whether enough testing has been done, or 2) the basis for test selection. In approach 1, testers select test cases by some independent means, such as use of a different subdomain testing strategy, random testing according to some well-defined input distribution, or "haphazard" selection (random testing in which the input distribution is difficult to characterize precisely). They then check whether the requisite number of points has been selected from each subdomain and, if not, select additional test cases. In approach 2, testers systematically look for test points that lie in the subdomains. They may give preference to certain types of points, such as those close to the boundary of a subdomain, or those that for some other reason are believed to be more "failure-prone." Clear-box testing techniques are usually more amenable to approach 1, whereas functional testing techniques are usually more amenable to approach 2. For clear-box methods, particularly the more abstruse, it is not easy to force test points to fall in the defined subdomains. However, since automatic tools exist to measure structural coverage and report deficiencies by subdomain, the tester can obtain a list of untested subdomains and find test points in the missed structural subdomains. In contrast, for functional methods it is usually relatively easy to identify the subdomains and select test cases from them, but harder to check which test requirements are covered by an arbitrary test case.

We consider two models of debug testing, which roughly correspond to the two ways debug-testing techniques are used. The first model, which we call *debug testing without subdomains*, describes the case in which a tester aims to select ϕ points, without considering subdomains. The probability distribution is defined on the entire input domain and the tester selects inputs independently until some stopping criterion is satisfied. If the stopping criterion is that some pre-determined number T of test cases has been selected, then debug testing without subdomains differs from operational testing only in the input profile used, which the tester hopes will produce more frequent failures during testing. This model captures only part of the first way of using subdomains, in that it does not require test points in each subdomain as a stopping criterion. In the second model, *debug testing with subdomains*, which models the second method of using debug testing, there is a probability distribution on each subdomain and the tester independently selects T_i test cases from each subdomain i.

In experimental comparisons among structural methods [9, 14] a somewhat different selection procedure is used:

> Test points are selected from a profile over the entire input domain (usually a uniform profile, although an arbitrary profile poses no difficulty). Each such test point falls in some subdomain(s), and by selecting enough overall points, one obtains "random" points in each subdomain. This procedure is adopted to eliminate possible human bias in selecting test points within subdomains.

The experimental procedure illustrates the difficult connection between any overall profile and structural parts of the program. Not only can it happen that a profile neglects some part (subdomain) so that an excessive number of choices of overall random points is needed to reach it, but the pattern that does reach it may not be appropriately "random" on the subdomain.

These models are only approximations of how testing is done in practice. In particular, a tester may use knowledge of previous test cases when selecting new ones, thereby violating the independence assumption. Nevertheless, we believe that they provide a fairly general and reasonably accurate starting point for our investigations.

Two practical problems must be accounted for. Some testing strategies may produce empty subdomains. That is, an apparently sensible subdomain (e.g., "inputs that make this branch condition TRUE") may in fact be empty (the TRUE branch is then *infeasible*). In our analysis of debug testing with subdomains, we assume that all empty subdomains have been eliminated from consideration. Very small subdomains also cause problems in practice, either because the subdomain size is smaller than the required number of elements from the subdomain, or the subdomain cannot be easily "hit" by tests. We do not require that the T_i test cases drawn from subdomain i be distinct. We ignore difficulties in hitting a subdomain, while in practice testers may have to stop testing before 100% coverage has been achieved.

3 DEBUGGING VS. OPERATIONAL TESTING

Exercising a program, whether in test or in operational use, involves selecting a succession of inputs to be presented for execution. The selection mechanism distinguishes between different types of test and of use.

3.1 The Analytical Context

Reliability in the technical sense is characterized by the failure probability when inputs are selected according to the operational profile. Failure points will be encountered at random, and there is a certain probability that the program will fail in use. If a testset is selected by sampling according to the operational profile, then direct estimates of the failure probability may be obtained. If a testset is selected in any other way, then the probability of encountering a failure region bears no necessary relation to the failure probability in operational use. But there is still a probability that the program will fail under test, which we call the "detection rate." In debug testing one tries to arrange that the detection rate is high. It is the "debugger's intuition" that the way to achieve reliability is through clever testing with high detection rates.

Reliability improves under either testing scheme when failures are found, the software is successfully changed, and the operational failure probability decreases.

The precise question we wish to study is the following:

> Under which conditions (on the program, and the testing method) will debug testing deliver better reliability than operational testing?

Certainly conditions exist favoring each alternative. If many debug tests fail and the corresponding fixes substantially decrease the overall failure probability, then debug testing may be superior to operational testing in which fewer tests happened to fail. However, it may happen instead that many fixes originated by debug testing are less effective, in terms of improving reliability in operation, than a few fixes originated by operational testing.

The case of ultra-reliability is of particular interest. When the failure set has a very small chance of being encountered in operation [20, 3], operational testing has a correspondingly very small chance of inducing failures and thus allowing the removal of failure regions. Debug testing is therefore the only option that allows some hope of further improving reliability. However, simply choosing debug testing is no guarantee that the results will be better than with operational testing. It may still happen that debug tests encounter only failure points whose probability in the operational profile is so low that fixes are worthless, or simply that it does not encounter any failure region. That is, the debug test regime chosen, or the tester's experience, may be ill-matched to the failure regions present in the software. Furthermore, even if debug testing does achieve ultra-reliability, it cannot demonstrate that ultra-reliability has been achieved; only an infeasible amount of operational testing can demonstrate that [20, 3].

We assume that all testers, upon observing a test failure, choose fixes that eliminate exactly the same failure region; that failure regions are disjoint; and that all test failures are noticed (that is, there is a perfect oracle). The limitations of these assumptions have been discussed in section 2.2.

Note that we are not considering the cost of removing a failure region; in practice, this may depend on the testing method that was used to detect the failure and on the phase of the development cycle in which the failure occurred.

In practice, a debugger may use information about the subdomain D_i from which a failed test case comes in order to figure out how to "fix" the problem. This information may make it easier to locate the problem, but may also lead to an inadequate fix, for example, one that only removes $D_i \cap F$, rather than all of a failure region F. This situation is not captured by our model, in which we assume that fixes and their corresponding failure regions are uniquely determined by failure points, independent of the testing strategy.

The *failure rate* of a failure region is the probability that an element of that region will be selected when selecting an input according to the operational distribution. The *detection rate* of a failure region is the probability that an element of that region will be selected when one input is selected during debug testing. These are the probabilities that the program will fail *because of this particular region* under the operational profile and the debug profile, respectively.

We will study the program failure probability after a testset of size T tests has been applied, as a random variable Θ. In this section, we focus mainly on the expected value of Θ; later we explore other aspects of the distribution of Θ." The simplest form of comparison assumes that equal effort is spent on both testing methods, and that the effort is measured by T. The comparison can be generalized to account for different costs per test case between methods. Although our examples only scratch the surface of the analysis possible in our models, we believe that they show the formalism to be reasonable and useful, and they provide insight into the process of testing to achieve reliability.

3.2 Single Failure Region, Debug Testing without Subdomains

Consider a program with failure probability q and only one failure region F. (Thus F's failure rate for operational testing is q as well.) Initially, we take debug testing as being conducted according to some overall test profile V. That is, tests are selected just as in operational testing, but with a different profile. The detection rate is thus a constant given by

$$d = \sum_{t \in F} V(t). \tag{1}$$

After a testset of size T has been tried, what is the distribution of the failure probability Θ of the final debugged program? Under the assumptions above, Θ will be 0 if the test encountered the region (which is then eliminated by a fix), and still q otherwise. Thus for debug testing:

$$\begin{aligned}
P(\Theta = 0) &= 1 - (1-d)^T && (2)\\
P(\Theta = q) &= (1-d)^T && (3)\\
E(\Theta) &= 0 \cdot P(\Theta = 0) + q \cdot P(\Theta = q) && (4)\\
&= q(1-d)^T. && (5)
\end{aligned}$$

With operational testing:

$$\begin{aligned}
P(\Theta = 0) &= 1 - (1-q)^T && (6)\\
P(\Theta = q) &= (1-q)^T && (7)\\
E(\Theta) &= q(1-q)^T. && (8)
\end{aligned}$$

So we get the obvious result that debug testing is superior iff $d > q$.

As a simple case, assume that a fixed budget is available for testing and allows different numbers of tests depending on the type of testing, say, T_D tests for debug testing and T_R for operational testing. In other words, a test case in debug testing costs on average T_R/T_D times what it costs in operational testing. Then, debug testing is superior iff

$$T_D \log(1-d) < T_R \log(1-q), \tag{9}$$

that is, for small q and d, iff

$$dT_D > qT_R. \tag{10}$$

That is, debug testing is superior if, compared with operational testing, it improves the effectiveness of a single test more than it raises its average cost.

3.3 Single Failure Region, Debug Testing with Subdomains

Let the input domain be divided into subdomains $D_1, D_2, \ldots, D_n$. T_i test cases are selected independently from each D_i according to test profile V_i on subdomain $D_i, 1 \leq i \leq n$. The single failure region F may be spread across the subdomains in an arbitrary way. Let d^i be the debug detection rate[6] for subdomain D_i:

$$d^i = \sum_{t \in F \cap D_i} V_i(t). \tag{11}$$

[6]The somewhat peculiar use of a superscript anticipates a different usage for subscripts to follow.

Then

$$P(\Theta = 0) = 1 - \prod_{i=1}^{n}(1 - d^i)^{T_i} \tag{12}$$

and

$$E(\Theta) = q\prod_{i=1}^{n}(1 - d^i)^{T_i}. \tag{13}$$

For comparison with operational testing, equation (8) can be compared with (13) by taking $T = \sum_{i=1}^{n} T_i$.

Here $E(\Theta)$ depends on the extent to which the subdomains "concentrate" the failure points. In comparing the probability of detecting at least one failure using random testing and partition testing, Weyuker and Jeng [16] and Hamlet and Taylor [12] observed this "concentration" effect. In the case of a single failure region, we are considering almost the same question that they did. Weyuker has noted that failure detection probability may not be the right parameter to study, and here we go beyond it to study the delivered reliability. Our explicit use of failure region(s) makes our model capable of analyzing more complex situations.

Several straightforward special cases explore failure concentration:

- At one extreme, suppose that for some i, subdomain $D_i \subset F$. Then $d^i = 1$, and consequently $E(\Theta) = 0$, so debug testing is superior for any $0 < q < 1$.

- At the other extreme, the failure region might be uniformly "spread out" over all the subdomains weighted by their profiles and test counts, in the sense that the chance $\bar{d}$ of finding a failure in each subdomain is the same. Then the results of the previous section apply, with $d = \bar{d}$ in equation (5). Operational testing is superior iff $q > \bar{d}$, which is to be expected if the operational profile has a peak within F.

- More generally, the expected failure probability after debugging is smaller for debugging with subdomains than for operational testing if and only if there is a collection $D_{i_1}, \ldots, D_{i_k}$ of k subdomains such that

$$(1 - d^{i_1})^{T_{i_1}}(1 - d^{i_2})^{T_{i_2}} \cdots (1 - d^{i_k})^{T_{i_k}} < (1 - q)^T. \tag{14}$$

  As in the previous extreme, one way this can happen is if some subdomain is completely contained in the failure set, in which case its detection rate is 1 and the left-hand side of (14) is 0. It is also possible for some collection of subdomains which individually have moderately, but not exceptionally, high detection rates to collectively yield a high enough detection rate to make debug testing superior. On the other hand, operational testing will be superior if the above condition does not hold. For instance, because of poor choice of the input distributions within subdomains, a failure region with high failure rate may have low detection rate for every subdomain; or the tester may select large numbers of test cases from subdomains with low detection rates and small numbers of test cases from the "good" subdomains.

By considering the failure region F to be a strict subset of a single subdomain, it is possible to capture two intuitively appealing special cases, one in which debug testing is superior, the other in which operational testing is superior. Suppose that $F \subset D_k$ for some k, but some points of subdomain D_k are not failure points: $D_k \not\subset F$; and that no possibly overlapping subdomain touches F: $F \cap D_i = \emptyset, i \neq k$. Further suppose that within D_k the two testing techniques (on average) are equally likely to encounter F. That is, the detection probability d^k is just the fraction of the operational-distribution inputs in D_k that encounter F:

$$d^k = \frac{\sum_{t \in F} Q(t)}{\sum_{t \in D_k} Q(t)}. \tag{15}$$

Finally, take the debug testing points as equally spread among subdomains, so since there are n subdomains, and T test points for comparison with operational testing, $T_k = T/n$.

The intuitive situation in which debug testing should be superior is the one in which operational testing with profile Q is relatively neglectful of D_k, that is, $T \sum_{t \in D_k} Q(t) \ll T_k$, or substituting $T_k = T/n$,

$$\sum_{t \in D_k} Q(t) \ll \frac{1}{n}. \tag{16}$$

Under these assumptions, the expected value of failure probability for debug testing is:

$$\begin{aligned} q \prod_{i=1}^{n} (1 - d^i)^{T_i} &= q(1 - d^k)^{T_k} && (17) \\ &= q\left(1 - \frac{\sum_{t \in F} Q(t)}{\sum_{t \in D_k} Q(t)}\right)^{T/n} && (18) \\ &< q\left(1 - \frac{\sum_{t \in F} Q(t)}{1/n}\right)^{T/n} && (19) \\ &\approx q(1 - T \sum_{t \in F} Q(t)) && (20) \\ &\approx q(1 - q)^T, && (21) \end{aligned}$$

where the last term is the expected value of the failure probability for operational testing. (The approximations in (20) and (21) require that d^k and q are small, using $(1+x)^y \approx 1+yx$ for small x.)

Thus when debug testing is used, the failure probability delivered will be less than the failure probability delivered when operational testing is used. That is, debug testing delivers the better reliability. To paraphrase, we have captured the situation where a subdomain includes the only failure region, and under plausible assumptions debug testing leads to the better reliability. Intuitively, the subdomain D_k is chosen to be "failure prone," and it is then given more test cases than the operational profile would imply.

A similar analysis yields the opposite result when many operational tests fall in D_k. If there are many other subdomains, debug testing "wastes" most of its tests on them (still assuming that $T_k = T/n$). That operational sampling of D_k is much greater than its debug sampling is expressed as $T \sum_{t \in D_k} Q(t) \gg T_k$, or substituting $T_k = T/n$,

$$\sum_{t \in D_k} Q(t) \gg \frac{1}{n}. \tag{22}$$

Then using (22) instead of (16) in equation (19) above reverses the inequality and gives the result that operational testing delivers better reliability than debug testing.

Although these two cases are intuitively obvious, and can be obtained using the failure-detection measure of [16], they demonstrate that our model is useful, and in section 3.5 on "Multiple Failure Regions, Debugging with Subdomains" below they will be combined to demonstrate that good failure detection does not imply good delivered reliability.

3.4 Multiple Failure Regions, Debugging without Subdomains

Suppose a program contains m non-overlapping failure regions $\{F_1, F_2, ..., F_m\}$, with failure rates $q_1, q_2, ..., q_m$ and detection rates $d_1, d_2, ..., d_m$. Then its expected failure probability after T tests is

$$E(\Theta) = \sum_{i=1}^{m} q_i(1 - d_i)^T \tag{23}$$

for debug testing, and

$$E(\Theta) = \sum_{i=1}^{m} q_i(1 - q_i)^T \tag{24}$$

for operational testing. These formulas are justified by considering that the contribution of each failure region to the failure rate after testing is a random variable, taking on the value 0 if the failure region is eliminated during testing, and the value q_i otherwise. Its expected value is obtained by multiplying q_i times the probability of the failure region not being detected. The failure probability of the debugged program is the sum of these random variables, and its expected value is thus the sum of their expected values.

If, for instance, $d_i \geq q_i$ for $i = 1, ..., m$, debug testing is superior to operational testing. This seems natural, as the hypothesis means that debug testing performs better than operational testing on each failure region. This belief is probably the usual basis of the "debugger's intuition." However, it is a very strong assumption. If it is false, the main factor affecting the delivered reliability is the relationship between the failure rates and the detection rates.

We can analyze the effect of this factor in isolation by assuming that, for each randomly chosen test case, debug testing has the same probability of finding a failure region as operational testing, i.e., $\sum d_i = \sum q_i$. In the simplest case that all the failure regions have the same failure rate q, operational testing is superior, because to minimize

$$q \sum_{i=1}^{m} (1 - d_i)^T, \tag{25}$$

under the condition that $\sum d_i = mq$, requires $d_i = q$. More generally, we can compare a set of testing methods such that they all have the same probability per test of finding a failure (i.e., for all methods $\sum d_i = K$). A way of looking at this question is to imagine that we can freely "transfer" a certain amount of detection rate from one failure region to another – say, from F_1 to F_2, leaving the detection rates of all the other failure regions unchanged – and ask under which conditions this would decrease the expected value of the program's failure probability after T tests. The new value of the failure probability would be

$$q_1(1 - d_1 + \epsilon)^T + q_2(1 - d_2 - \epsilon)^T + \sum_{i=3}^{m} q_i(1 - d_i)^T, \tag{26}$$

and by differentiating it with respect to ϵ we obtain that the derivative is negative, i.e., the change is beneficial, iff:

$$\frac{q_1}{q_2} \leq \frac{(1-d_2)^{(T-1)}}{(1-d_1)^{(T-1)}}. \tag{27}$$

This inequality implies, in the first place, that it is never beneficial to increase the detection rate of a failure region above the detection rate of another region with a higher failure rate. This is an intuitively plausible extension of the results obtained by Hamlet and Taylor [12].

However, other consequences also hold:

- If we can run only one test, then the method which gives the best expected reliability after debugging is one that has detection rate K for the failure region with the largest failure rate, and 0 for the others;
- As the number of tests that one can run increases, it becomes beneficial to increase the detection rates of regions with increasingly small failure rates. For a given number of tests T, the optimal detection rates for failure regions F_i and F_j must satisfy the equation:

$$q_i(1-d_i)^{(T-1)} = q_j(1-d_j)^{(T-1)}; \tag{28}$$

- As T tends to infinity, the optimal detection rates d_i all tend to K/m.

Of course, these results only describe optimality for the *expected* failure probability after debugging. If, for instance, we wished instead to maximize the probability of finding the failure region with the highest failure rate, it would clearly be best to have a method that has detection rate K for this failure region, and 0 for the others.

These considerations only apply under the artificial constraint $\sum d_i = K$; however, they help to clarify the issue of how the "allocation" of detection rates to failure regions affects the effectiveness of testing, separately from the issue of how effective a testing method is at finding failure regions.

With the model used in this section, it is easy to model cases in which the probability of finding a failure is a bad indicator of how well a testing method improves reliability. Consider a program with a large number m of failure regions, such that $d_i = d, i = 1, ..., m$, and that $md > \sum_{j=1}^{m} q_i$, and that $q_1 > d \gg q_i, i \neq 1$. Then debug testing ensures the higher probability of finding a failure, yet produces a much worse $E(\Theta)$ than operational testing, since most of the failure regions that debugging detects have a negligible effect on reliability. This case is the "debugger's nightmare," a situation in which debugging goes on and on with apparent success, but really does no good at all.

Interestingly, one can also show that cases exist in which operational testing has the better failure detection, but debug testing delivers better reliability, if few tests are run. The actual numerical differences are negligible for practical purposes, but we describe such a scenario to illustrate the subtlety of the problem. Consider a program in which failure region F_1 has a much greater failure rate than any other, but also a detection rate greater than its failure rate, $d_1 > q_1 \gg q_i, i \neq 1$. Let the failure regions 2, ..., m have negligible detection rates, but failure rates that together outweigh the imbalance between d_1 and q_1. These conditions ensure that $\sum_{i=1}^{m} d_i < \sum_{i=1}^{m} q_i$. Then the probability of failure per test is greater for operational testing than for debug testing. Yet, during the early phase of testing, debug testing will have a greater probability of eliminating F_1, and will thus offer a (marginally) better $E(\Theta)$ to a tester who can afford only a few tests.

3.5 Multiple Failure Regions, Debugging with Subdomains

The m failure regions F_j may be arbitrarily spread across the n subdomains D_i. The detection rates are now:

$$d_j^i = \sum_{t \in F_j \cap D_i} V_i(t). \tag{29}$$

As in the case of a single failure region, there are some straightforward observations:

- The detection of a particular failure region F_j is guaranteed if there is a subdomain D_i that is completely contained in F_j. More generally, the probability of detecting F_j is high if for some i, the probability of selecting an element of F_j from D_i is high.

- However, in contrast to the analysis of operational testing and to debug testing without subdomains, there is some interesting non-independence between different failure regions. A simple illustration of this dependence arises when there are two failure regions contained within the same subdomain, and no other subdomains that intersect either failure region. In subdomain testing with one test case per subdomain, at most one of these failure regions can be detected.

- If a high-failure-rate failure region is spread out across several big subdomains, it may be hard to detect. If, moreover, these subdomains have moderately high concentrations of small (low-failure-rate) failure regions, it will be fairly easy to detect a lot of those. This is again the debugger's nightmare: detection and removal of many minor problems, while failing to detect the serious problems.

A general formula for the expected failure probability after debug testing with subdomains follows from the discussion of Θ's distribution in section 4, below. In the remainder of this section, we use examples to illustrate some of the phenomena that can occur.

3.5.1 Detecting failures vs. delivering reliability – operational testing superior

The two special cases described in section 3.3 above for a single failure region in which debug testing (*resp.* operational testing) is superior when the failure region lies within a subdomain, can occur simultaneously with multiple failure regions. It is possible to use this situation to construct a special case with the properties that: (a) Debug testing is much more likely to find a failure, but (b) Operational testing is superior in reducing the ultimate failure probability under our assumption that all detected failure regions are removed.

Two disjoint subdomains suffice to construct this example: D_1 strictly containing F_1 for which debug testing is more likely to find a failure and D_2 strictly containing F_2 in which operational testing is better. Assuming $F_1 \subset D_1$, $F_2 \subset D_2$ implies $d_1^2 = d_2^1 = 0$. To account for operational testing being better on F_1 than on F_2, let $q_2 \approx q \gg q_1$. Debug testing is made much better than operational testing at finding F_1 by setting $d_1^1 \gg q$; and taking $d_2^2 \approx q$ makes operational testing better at finding F_2, because it places most of its T test points in D_2. Take $T_1 = T_2 = T/2$. We have thus a scenario in which debug testing looks – on a test-by test basis – intuitively better than operational testing. However, operational testing delivers better reliability. Here are the calculations:

1. The *probability of finding a failure* with debug testing is about
$$1 - (1 - d_1^1)^{T/2}(1 - q)^{T/2}, \tag{30}$$
while with operational testing it is
$$1 - (1 - q)^T. \tag{31}$$
So since $d_1^1 \gg q$, debug testing is much better at finding a failure.

2. However, if we look at the *failure probability delivered* after fixing the failure regions uncovered, the situation is different. For operational testing,
$$E(\Theta) = q_1(1 - q_1)^T + q_2(1 - q_2)^T \tag{32}$$
(as explained at the beginning of section 3.4). Let us consider small values of T, such that the first summand is much smaller than the second one, because $q_1 \ll q_2$. Then since $q_2 \approx q$, for operational testing:
$$E(\Theta) \approx q(1 - q)^T. \tag{33}$$
On the other hand, debug testing will likely result in F_1 being fixed, but F_2 will be fixed with lower probability than in operational testing. For debug testing,
$$\begin{aligned} E(\Theta) &= q_1(1 - d_1^1)^{T/2} + q_2(1 - q_2)^{T/2} && (34) \\ &\approx q(1 - q)^{T/2}. && (35) \end{aligned}$$
Comparing (33) and (35), operational testing results in much better delivered reliability of the software.

This example straightforwardly captures the intuitive situation in which debug testing finds the "wrong" bugs, from the standpoint of better delivered reliability.

Another way to measure the effectiveness of testing in this situation is the probability that a preset reliability target θ_R is reached. Then, the precise value of θ_R matters in determining which testing regime is better. Let us simplify calculations by setting T=2. Then,

- if $q_1 < \theta_R < q_2$, the target is reached if F_2 is detected, i.e., with probability $d_2^2 \approx q$ by debug testing and with probability $(1 - (1 - q_2)^2) \approx (1 - (1 - q)^2) \approx 2q$ by operational testing, so operational testing is superior.

- If $\theta_R < q_1 < q_2$ then the target is attained if both failure regions are detected, i.e., with probability $d_1^1 d_2^2 \approx d_1^1 q$ by debug testing and with probability $2q_1q_2 \approx 2q_1q$ with operational testing. This expression is obtained because the event of interest is the union of two disjoint events: finding F_1 with one test and F_2 with the second test, and the inverse sequence. So, since we assumed $d_1^1 \gg q \gg q_1$, debug testing is better in this case.

3.5.2 Detecting failures vs. delivering reliability – debug testing superior

We have also been able to construct an example of the opposite case, in which operational testing is better at detecting failures, yet debug testing yields better reliability. However, the intuitive situation is more subtle, and the advantage for debug testing only marginal.

Consider m subdomains, each with a strictly contained failure region F_i. Assume that debug testing is very good at detecting one failure region F_1, which has a high failure rate, but debug testing is unlikely to detect many other failure regions, with smaller failure rates. That is, $d_1^1 \gg q_1 > q_i > d_i^i \approx 0, i \neq 1$. F_2, F_3, etc., correspond to bugs that are both "small" – they have low failure rates – and "subtle" – intuitive debug testing strategies are unlikely to discover them. Then, operational testing may be better at producing a failure early, because debug testing wastes most test cases on those subdomains where it has negligible probability of finding a failure. Yet, if debug testing does reveal a failure, it will cause the most important failure region F_1 to be removed; hence debug testing can yield better delivered reliability.

For convenience let all the failure rates other than q_1 be q_2. The probabilities of causing a failure in the first T tests are:

$$1 - (1 - (q_1 + (m-1)q_2))^T \tag{36}$$

for operational testing, and approximately

$$1 - (1 - d_1^1)^{T/m} \tag{37}$$

for debug testing. The expected values of the delivered failure probability of the tested program are:

$$q_1(1-q_1)^T + q_2(m-1)(1-q_2)^T \tag{38}$$

for operational testing, and approximately

$$q_2(m-1) + q_1(1-d_1^1)^{T/m} \tag{39}$$

for debug testing.

The following is a typical numerical example: $m = 20$ subdomains and failure regions, $q_1 = 10^{-3}, d_1^1 = 0.05$, and $q_i = 10^{-4}, d_i^i = 10^{-5}, i \neq 1$, with a test run of 400 tests. Operational testing is more likely to detect a failure (by 0.69 to 0.64), yet debug testing has a better $E(\Theta)$ (by 0.00226 to 0.00250).

However, if we keep testing, then both testing methods will soon reach a practical certainty of eliminating F_1, and debug testing will have a smaller chance of finding other failure regions. So, operational testing will catch up. In the example above, the cross-over is at about 1318 tests, when both methods achieve $E(\Theta) = 0.00193$. After 10000 tests, $E(\Theta)$ with operational testing is down to 7×10^{-4} (that is, on average only seven failure regions remain), against 0.0019 for debug testing (that is, on average none of the 19 "small" failure regions have yet been found).

This example illustrates a general rule: the number of tests that we are willing to spend on a given test regime affects which test regime we should choose. As testing and fixing proceed and alter the failure set of the program, which test regime is better will also change (but the tester will not know when, because the change depends on which (unknown) failure regions are left in the program).

We emphasize that the case described in this section, in which operational testing is better at failure detection, but debug testing may deliver better reliability, is quite unlike the previous one favoring operational testing (section 3.5.1). The previous case has an obvious intuitive meaning, and the analysis shows a substantial difference between the methods. The present case appears contrived, to get the result requires careful adjustment of the parameters, yet still the difference exhibited between methods is not substantial. Of course, our failure to discover a satisfying, simple example does not mean one does not exist, but we believe that the debug tester is more likely to be misled by considering failure-finding probability, than is the operational tester.

3.5.3 Detecting failures vs. delivering reliability – complex cases

With different assumptions about the detection rates of failure regions with different failure rates, it is possible to describe more complex situations. For instance, we could consider the effect of having three classes of failure regions, with failure rates that are an order of magnitude apart, and such that their detection rates are larger than failure rates for all failure regions except those in the intermediate class. Then, we may have a scenario in which debug testing yields the better expected reliability with a small test run, $T < T_1$, then operational testing becomes better while $T_1 < T < T_2$, and then again debug testing is preferable if $T > T_2$. This points, again, at the opportunity of using combinations of different test strategies in different phases of the testing.

These cases illustrate the extra complexity of the situations that can be analyzed using failure regions and the expected value of the delivered reliability. Our examples mostly involve the unrealistic (but easier to analyze) case of failure regions strictly contained in subdomains. This is not a limitation of the model, but it is the simplest situation to conceive, and it suffices to demonstrate most of the properties that concern us.

4 THE DISTRIBUTION OF Θ

Up to this point, our investigation of the random variable Θ has focused on its expected value. It is sometimes useful to consider other statistical properties of Θ, such as its variance or the probability that Θ is less than some given (un)reliability target θ_t. One might prefer testing method M_1 over testing method M_2 if Θ_{M_1} has a smaller variance than Θ_{M_2}, so that the results of using M_1 are more predictable than those of M_2. Alternatively, one might prefer M_1 if $P(\Theta_{M_1} \leq \theta_t)$ is relatively large, so that one can be more confident that the reliability target will be achieved.

In this section we investigate the distribution of the random variable Θ. We then present an example illustrating how small changes in the detection rates can influence the tail probability $P(\Theta \leq \theta_t)$. This example shows that while, under ideal circumstances, debug testing can result in a high probability of reaching given reliability target, under less ideal circumstances, debug testing can perform as badly as, or worse than, operational testing.

4.1 Computation of the Distribution of Θ

In order to compute the distribution of Θ, we must derive the probability mass at each possible value of Θ. Let $\mathcal{F} = \{F_1, \ldots, F_m\}$ denote the collection of failure regions of a program and let $q_1, \ldots, q_m$, be their respective failure rates. For each sub-collection $X \subset \{1, \ldots, m\}$, let $\theta_X = \sum_{j \notin X} q_j$. Thus θ_X is the failure probability after removal from $\mathcal{F}$ of only the failure regions indexed by X.

There are at most 2^m values of θ_X that Θ may assume, and there may be fewer values if two different subsets X_1 and X_2 have the same total failure rate. To investigate the probability mass at each value of Θ, we initially assume, for simplicity, that each subset of $\mathcal{F}$ has a different total failure rate, i.e., that Θ has 2^m possible values. We will then relax this assumption.

Let p_X denote the probability that the failure regions denoted by X (and no others) are detected. That is,

$$P(\Theta = \theta_X) = p_X.$$

For a given collection X, the probabilities p_X depend, perhaps in a complicated way, on the details of the test selection strategy.

Consider a sequence of n test cases. We can represent the results of executing such a sequence by an n-tuple, $(r_1,\ldots,r_n)$ where $r_i = 0$ if no new faults are discovered with the i^{th} test case and $r_i = j$ if F_j is discovered with the i^{th} test case. For example, the 5-tuple $(0, 3, 0, 1, 0)$, represents execution of 5 test cases with discovery of failure region 3 with the second test case, discovery of failure region 1 with the fourth test case, and no failures on the other three test cases. We can then compute p_X by finding the probability of each sequence in which those failure regions belonging to X (and no others) are discovered, then summing those probabilities.

To compute the probability of a sequence $(r_1,\ldots,r_n)$, let p_j^i denote the probability that failure region F_j will be detected by the i^{th} test case. Let p_0^i be the probability that no *new* failure region is discovered by the i^{th} test case. That is, p_j^i denotes the probability that $r_i = j$. The probability of sequence $(r_1,\ldots,r_n)$ occurring is then the product of the corresponding p_j^i. For example, the above 5-tuple has probability $p_0^1 p_3^2 p_0^3 p_1^4 p_0^5$.

The values of the p_j^i depend on the particular testing strategy used, as well as on the detection rates or failure rates. For operational testing, $p_j^i = q_j$ if F_j has not been detected by test cases $1, \ldots, i-1$, and 0 otherwise. For debug testing without subdomains, $p_j^i = d_j$ if F_j has not been detected by test cases $1, \ldots, i-1$, and 0 otherwise. For debug testing with subdomains, with one test case per subdomain, assuming the i^{th} test case is selected from D_i, $p_j^i = d_j^i$ if F_j has not been detected by test cases $1, \ldots, i-1$, and 0 otherwise. Note that the p_j^i are not constants, but have values dependent on the history of the testing process. Also note that with these testing scenarios, any sequence in which the same positive value of j occurs more than once (representing rediscovery of failure region F_j after it has been discovered and removed) has probability zero. More complicated situations such as partial removal of a failure region can be modeled in a similar manner.

Finally, p_X is the sum of the probabilities of all those sequences in which the collection of failure regions represented by X is detected (and no others).

More generally, we may have several different subsets of $\mathcal{F}$ that yield the same total failure rate. If $\theta_{X_1} = \theta_{X_2} = \ldots = \theta_{X_k}$, then

$$P(\Theta = \theta_{X_1}) = \sum_{i=1}^{k} p_{X_i}.$$

4.2 Example

The previous sub-section shows how one can derive the exact probabilities for the different values of Θ for the testing scenarios considered earlier. This situation is too complicated for algebraic analysis, so we wrote a program to compute these probabilities from input values of q_i and d^i_j, for small numbers of test cases, subdomains, and failure-regions. From these probabilities we can obtain the expected value and variance of Θ, and the probability that a given reliability target is reached.

In this subsection, we present an example in which, for a single hypothetical program, we examine the distribution of Θ and the resulting tail probabilities under several different testing scenarios. Consider a program with six failure regions, with $q_1 = 0.01$ and $q_2 = q_3 = q_4 = q_5 = q_6 = 0.001$. Before testing, the failure rate is $\theta = 0.015$. Call F_1 the "big" failure region and $F_2 \ldots F_6$ "small" failure regions. In this example the possible values of Θ are determined by whether the big failure region is detected, along with the number of small failure regions detected, so there are 12 possible values, rather than $2^6 = 64$ of them. These 12 values are shown in the second column, θ, of Table 1. The remaining columns give the corresponding probabilities for operational testing and for several debug testing scenarios, which are described below. The last four *rows* of the table give the expected value, the variance, the probability of reaching the reliability target $\theta = 0.01$ (P($\Theta \leq 0.01$)), and the probability of detecting at least one failure region (P($\Theta < 0.015$)). To reach the reliability target 0.01, it is necessary to either detect the big failure region or to detect all five of the small failure regions.

Table 1 shows four scenarios for debug testing with subdomains. In each, there are six disjoint subdomains, with $F_i \subset D_i$. Thus $d^i_j = 0$ for $i \neq j$. The four scenarios have been selected for their illustrative value, although of course they are not necessarily representative of any real-world situation. The debug testing results are based on one test case per subdomain and the operational testing results are also based on a total of six test cases.

In each scenario, $d^1_1 = q_1 = 0.01$. The debug scenarios differ *only* in the values of d^i_i for the small subdomains, i.e., in how likely it is to detect failure region i with a test case from subdomain i. In all four scenarios, *all of the detection rates assumed for the small failure regions are at least an order of magnitude greater than the failure rates of these failure regions.* Thus, one might expect that these scenarios would strongly favor debug testing over operational testing. We shall see that, as we move away from the ideal debug testing scenario, the advantage of debug testing (measured by the probability of reaching the reliability target) quickly diminishes.

operational: Of the 12 possible values for Θ, many have negligible probabilities. For example $\Theta = 0.003$ requires detection of the large failure region and two of the small ones, which occurs with probability approximately 10^{-8}. The probability of reaching the reliability target is quite low, essentially equal to the probability of detecting the

		$P(\Theta = \theta)$				
	θ	operational	debug-1	debug-2	debug-3	debug-4
Distribution of Θ	0.015	0.9133	0.0000	0.0024	0.0309	0.0000
	0.014	0.0279	0.0000	0.0281	0.1547	0.0000
	0.013	0.0003	0.0000	0.1310	0.3094	0.0000
	0.012	0.0000	0.0000	0.3056	0.3094	0.0000
	0.011	0.0000	0.0000	0.3565	0.1547	0.9801
	0.010	0.0000	0.0000	0.1664	0.0309	0.0099
	0.005	0.0570	0.0000	0.0000	0.0003	0.0000
	0.004	0.0014	0.0000	0.0003	0.0016	0.0000
	0.003	0.0000	0.0000	0.0013	0.0031	0.0000
	0.002	0.0000	0.0000	0.0031	0.0031	0.0000
	0.001	0.0000	0.9900	0.0036	0.0016	0.0099
	0.000	0.0000	0.0100	0.0017	0.0003	0.0001
Expected value $E(\Theta)$		0.0144	0.0010	0.0114	0.0124	0.0109
Variance		0.0123	0.0010	0.0029	0.0029	0.0104
$P(\Theta \leq 0.01)$		0.0584	1.0000	0.1764	0.0409	0.0199
$P(\Theta < 0.15)$		0.0866	1.0000	0.9976	0.9691	1.0000

Table 1: Distributions of Θ with means and variances for several testing scenarios.

big failure region in six tries. The probability of detecting at least one failure region is only a bit better, with the slight improvement reflecting the possibility of detecting one of the small failure regions.

Debug-1: This scenario is as favorable as possible for debug testing, under the above constraints. Each small failure region is guaranteed to be detected: $d_i^i = 1.0$ for $i = 2, \ldots, 6$. Since all five small failure regions are guaranteed to be detected, there are only two non-zero values of $P(\Theta = \theta)$, distinguished by whether or not the big failure region is detected. The variance is low, the reliability target is guaranteed to be met and it is guaranteed that at least one failure region will be detected. So, under these ideal circumstances, debug testing is a clear winner. But in the remaining scenarios, we investigate what happens when these most favorable conditions do not hold.

Debug-2: In this scenario, the small failure regions are fairly likely to be detected: $d_i^i = 0.7$ for $i = 2, \ldots, 6$. Although the probability of detecting at least one failure region is still very high, and the variance is quite small, $E(\Theta)$ is close to that of operational testing and the probability of reaching the reliability target has fallen substantially (compared to Debug-1). This is because with $d_i^i = 0.7$, the probability of finding all five small failure regions is low.

Debug-3: Continuing in this manner, in the next scenario the small failure regions are even less likely to be detected: $d_i^i = 0.5$ for $i = 2, \ldots, 6$. Note that the detection rates are still much higher than the failure rates. $E(\Theta)$ is still slightly better than for operational testing, but the probability of reaching the target is less than for operational testing.[7]

This is quite surprising. This scenario models a situation in which the debug tester has very good, but not perfect, intuition as to where the small failure regions are – for $i > 1$, half of each subdomain D_i consists of failure points from F_i. This good intuition pays off by giving the debug tester a high probability of detecting at least one failure region. On the other hand, the good intuition is useless in terms of enhancing the probability of reaching the reliability target!

Debug-4: Lastly, we consider another scenario in which the debug tester has very good intuition. Detection of four of the small failure regions is guaranteed, but one small failure region has a detection rate equal to that of the big failure region: $d_2^2 = 0.01$, $d_i^i = 1.0$ for $i = 3, \ldots, 6$. Detection of at least one failure region is guaranteed, $E(\Theta)$ is slightly better than for operational testing, the variance is similar to that of operational testing, but, again, the probability of reaching the target is very low. Like the Debug-3 scenario, this shows that with imperfect, but still very good intuition, the debug tester may perform worse than operational testing, in terms of probability of reaching the reliability target.

One cannot draw sweeping conclusions from this small and somewhat contrived example. In it the failure rates are much higher, and the reliability target is much looser, than for systems that purport to be highly reliable. However, several points are worth noting:

- Testing scenarios that have similar values of $E(\Theta)$ may differ widely in other important statistical measures. The implication is that analysis much more detailed than that attempted to date is required to compare testing methods.
- Small deviations from the optimal debug-testing scenario lead to severe degradations in behavior, especially in the probability of reaching a reliability target. Operational testing cannot be used to attain stringent reliability targets [20, 3], but it seems unlikely that debug-testing is an alternative under realistic assumptions.
- The distribution of Θ is very different from the usual "textbook" ones (binomial, etc.). This calls into question any simple assumptions about the behavior of Θ.

5 SUMMARY AND FUTURE WORK

We have considered the question of whether low operational failure probability (and hence better reliability) may be better obtained by looking for failures (debug testing), or by sampling from expected usage (operational testing). The testing models we considered can be analyzed in two ways, with and without identifying subdomains for debug testing. This paper generalizes and extends the "random vs. partition" studies that followed from the work of Duran and Ntafos [8]. We have analyzed a number of special cases, showing that

[7]This may seem counter-intuitive, given that $d_1^1 = q_1$. However, debug testing with subdomains only has one chance to find the big failure region (when using a test case from D_1), whereas operational testing gets six chances to find it.

the theory can capture and inform our intuition about the strengths and weaknesses of the two testing schemes.

Debug testers always have the potential advantage that by adjusting the test profile and subdomain definitions they might improve the behavior of debug methods. While operational testers have no such freedom, they do have the advantage that the operational profile, and operational testing, *define* the desired result. Studies like this one can thus be viewed as advice to the debug tester, on how to choose a test profile that will yield superior reliability. If the debug tester has good intuition about which points are likely to be failure points and, moreover, about which of these failure points are likely to belong to large failure regions, such insight can be used to devise testing strategies that deliver much lower expected failure probability than operational testing. If the tester lacks such intuition or is unable to map that intuition into an appropriate input distribution, then operational testing may be indicated.

Trusting the debuggers' own judgement about their abilities would be inappropriate (see e.g., the experiments by Basili and Green [1]). But it is possible to compare the effectiveness of their testing profiles with that of operational profiles. A limited investment in such measurement would be, for any large development organization, a cost-effective step towards better quantitative decision-making.

In particular, our analysis has shown:

- There are obvious cases in which debug testing is superior (roughly, because its detection rates are greater than the failure probability). Similarly, operational testing can be obviously superior (roughly, because detection rates in many subdomains are smaller than the failure probability, so debug tests there are wasted). These examples show that the theory corresponds with intuition in limiting cases.

- Debug testers should be aware of the potential confusion between detecting failures and achieving reliability, a confusion that occurs when testing finds only unimportant failures. "Unimportant" of course refers to the weighting of the operational profile, which may well be unknown. But there is usually some intuition about the frequency with which a problem might arise in use, and if a debug technique consistently turns up low-frequency problems, it may be counterproductive to use it.

- Trust in subdomain testing depends on trusting one's beliefs about how failure regions are divided among subdomains. Previous work in this area has, in essence, considered all failure points to be equally important. We have instead distinguished between different groups of failure points based on their contribution to the overall failure probability, and have thus considered the reliability achieved by testing.

- The analysis of debug testing without subdomains suggests that, if limited resources are available, only debug methods that focus on the most important failure regions are appropriate.

- The problem of comparing testing strategies is very challenging. Our model is more general than those previously published, yet it is still quite simplistic. Despite the model's tractable nature, numerical computation shows that results are very sensitive to the details assumed for the methods compared, and suggests that the distributions may be quite unlike those usually assumed.

- The results here may be of particular relevance to those who have a responsibility for assuring ultra-high reliability in safety-critical systems. While debug testing *may* be a means of identifying failure sets that have a very small chance of being encountered, and thus improving reliability beyond what can be achieved with operational testing, this cannot be guaranteed: one could not be sure that the test regime was not in some way ill-matched to the actual failure regions present. Even when an ideal debug testing strategy yields high probability of reaching a reliability target, small deviations from the ideal may perform much worse. There is thus a need to demonstrate the reliability that has actually been achieved, and debug testing is unable to do this.

We hope that this kind of analysis will lead to more direct practical uses. Ideally, one would be able to describe sets of alternative conjectures about the failure regions of a program that: i) translate into indications for the testing regime to be used, and ii) can be checked by experiment. A tester could thus decide on a sound basis which testing regime to apply at a specific phase in the combined debugging and validation process. As a minimum, this would be based on which conjectures have proven to be verified in the previous experience of a certain development organization on a certain kind of program; at best, the observations made during the testing of a specific program could directly support the decisions about testing that program.

A necessary next step is to proceed from assuming a certain set of failure regions in the program, as we did here, to considering probability distributions of the features of the (actually unknown) failure regions of the program under test. With the latter, more realistic hypothesis, we would expect operational testing, in which failure detection is "directed" by the faults that are actually present, to be more predictable than debug testing, directed by a knowledge of where failure regions *may* be. Preliminary results [23] indicate that, under plausible assumptions, a debug tester who performs better than operational testing on average (over many programs), would still exhibit a higher variance in the achieved reliability, and thus a higher probability of unacceptably high residual failure probability in the delivered program.

Likewise, while considering the expected values of failure intensities allows some insight into the phenomena of interest, a tester will also be interested in other measures of the distribution of a program's failure probability. For instance, the probability of achieving a failure probability no greater than some stated target value (as in the example at the end of section 4) would probably be most interesting.

It is sensible to expect that different testing methods will prove optimal for different organizations, different software projects and different stages in a project. So, research cannot offer decision makers a single testing method that is best for all situations. What it can do is to offer better criteria for informing the choice of a method in a decision maker's specific situation.

No mathematical analysis, without the support of empirical knowledge, is sufficient for decision making. But for comparing testing methods, the direct experimental approach of measuring the costs and achieved reliability levels on parallel testing campaigns with different methods is prohibitively expensive. The analytic approach we have used in this paper deals with one aspect of the problem, i.e., with the effectiveness of running a certain number of test cases. Directions for future analytical research include relaxing the assumptions underlying our model, such as the assumption that failure regions are disjoint, fixes are perfect and all testers react to the same failure with the same fix, and incorporating a more realistic measure of test case cost.

Our analysis of the effectiveness of tests improves the possibilities of rational decision-making because it describes effectiveness in terms of other meaningful measures. Even for decisions that are based on intuitive judgement, it can flag – and thus avoid – illogical decisions, by showing non-obvious implications of the decision maker's premises. In addition, it can free the decision maker from total dependence on judgement, because some of the measures it involves can be more easily estimated than the reliability improvement that is really of interest.

References

[1] V. Basili and S. Green. Software process evolution at the sel. *IEEE Software*, pages 58–66, 1994.

[2] B. Beizer. The cleanroom process model: a critical examination. In *Proceedings 13th Annual Pacific Northwest Software Quality Conference*, pages 148–173, Portland, OR, 1995.

[3] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Trans. on Soft. Eng.*, pages 3–12, 1993.

[4] T.Y. Chen and Y.T. Yu. On the relationship between partition and random testing. *IEEE Trans. on Soft. Eng.*, 20(12):977–980, December 1994.

[5] T.Y. Chen and Y.T. Yu. On the expected number of failures detected by subdomain testing and random testing. *IEEE Trans. on Soft. Eng.*, 22(2):109–119, February 1996.

[6] R. H. Cobb and H. D. Mills. Engineering software under statistical quality control. *IEEE Software*, pages 44–54, November 1990.

[7] S. R. Dalal, J. R. Horgan, and J. R. Kettenring. Reliable software and communication: software quality, reliability, and safety. In *15th ICSE*, pages 425–435, Baltimore, MD, 1993.

[8] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. on Soft. Eng.*, 10:438–444, 1984.

[9] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. on Soft. Eng.*, 19(8):774–787, August 1993.

[10] P. G. Frankl and E. J. Weyuker. Provable improvements on branch testing. *IEEE Trans. on Soft. Eng.*, 19(10):962–975, October 1993.

[11] S. Gerhart, D. Craigen, and T. Ralston. Observations on industrial practice using formal methods. In *15th International Conference on Software Engineering*, pages 24–33, Baltimore, MD, 1993.

[12] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Soft. Eng.*, 16:1402–1411, 1990.

[13] J. Horgan, S. London, and M. Lyu. Achieving software quality with testing coverage. *IEEE Computer*, 27:60–69, 1994.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings 16th International Conference on Software Engineering*, May 1994.

[15] Z. Jelinski and P B Moranda. Software reliability research. In *Statistical Computer Performance Evaluation*, pages 465–484. Academic Press, New York, 1972.

[16] B. Jeng and E. J. Weyuker. Analyzing partition testing strategies. *IEEE Trans. on Soft. Eng.*, 17:703–711, 1991.

[17] L. Lauterbach and W. Randall. Experimental evaluation of six test techniques. In *COMPASS '89*, pages 36–41, Gaithersburg, MD, 1989.

[18] N. Li and Y. K. Malaiya. On input profile selection for software testing. In *Fifth International Symposium on Software Reliability Engineering*, pages 196–205, 1994.

[19] B Littlewood. Stochastic reliability growth: a model with applications to computer software and hardware design. *IEEE Trans. on Reliability*, 30(4):313–320, October 1981.

[20] B. Littlewood and L. Strigini. Validation of ultra-high dependability for software-based systems. *Communications of the ACM*, 36(11):69–80, 1993.

[21] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, pages 14–32, 1993.

[22] P. Piwowarski, M. Ohba, , and J. Caruso. Coverage measurement experience during function test. In *15th ICSE*, pages 287–301, Baltimore, MD, 1993.

[23] M. Pizza and L. Strigini. The effect of program variability on debugging efficiency – preliminary results. Technical report, Centre for Software Reliability, 1998.

[24] R. Selby, V.R. Basili, and T. Baker. Cleanroom software development: An empirical evaluation. *IEEE Trans. on Soft. Eng.*, 13(9):1027–1037, September 1987.

[25] M. Z. Tsoukalas, J. W. Duran, and S. C. Ntafos. On some reliability estimation problems in random and partition testing. *IEEE Trans. on Soft. Eng.*, 19:687–697, July 1993.

[26] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. Technical Report SERC-TR-153-P, SERC, 1994.