Choosing a Testing Method to Deliver Reliability*

Phyllis Frankl† Dick Hamlet Bev Littlewood Lorenzo Strigini ‡

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

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

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

ABSTRACT

Testing methods are compared in a model where program failures are detected and the software changed to eliminate them. The question considered is whether it is better to use tests that seek out failures ("debug testing") or to simulate usage and find failures along the way ("operational testing"). "Better" is measured by the delivered reliability obtained after all test failures have been eliminated. This comparison extends previous work, where the measure was the probability of detecting a failure. The theoretical treatment of the paper is probabilistic and analytical. Revealing special cases are exhibited in which each kind of testing is superior.

Keywords

Reliability, debugging, statistical testing theory

INTRODUCTION – RELIABILITY VS. DEBUGGING

There are two main goals in testing software. Firstly, it can be seen as a means of achieving reliability: here the objective is to probe the software for bugs so that these can be removed and thus improve its reliability. Alternatively, testing can be seen as a means of gaining confidence that the software is sufficiently reliable for its intended purpose, i.e., *evaluating* reliability.

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 situations likely to produce failures (e.g., methods informed by code coverage or specification coverage criteria), or concentrating on those that are considered most likely (e.g., stress testing or boundary testing methods). The choice among such testing methods is a matter of 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 bugs.
- 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 required that the testing profile is actually representative of operational use.
- Operational testing is also 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 tech-

*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".

niques 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 [14], 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 [20]. The expected number of failures during test has also been used as a measure of effectiveness [10, 5].

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 the section on Multiple Failure Regions, Debugging with Subdomains, below.)

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.

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 organisations. 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 the source failures normally exposed at the unit and subsystem levels by debug testing. Therefore, unit debug testing should be reduced in favour of additional system-level random testing.

In the "cleanroom" development methodology [6], to give an extreme example, it is considered essential that debug testing not be used at all, particularly by those doing the development. 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 reliability in the field are at best preliminary [7, 13, 18, 9, 15]. 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 careful theoretical studies have compared random testing with debug ("partition") testing [8, 12, 14, 19, 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. In most cases, we had no idea what the results would be until they were obtained from analysis of the models.

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 analysing 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.

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.

Tests and Failures

A *test* 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. (And hence 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 behaviour. A program P *meets* its specification for input x iff: 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 behaviour 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.

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*.

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 formally define. The difficulty is that "faults" have no unique characterisation. In practice, software fails for some testset, and is 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 characterised, e.g., "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 localised (e.g., 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 behaviour. 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.

So "the fault" is not a precise idea.

On the other hand, "failure" is well defined, and so is a change in failure behaviour 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.

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 of course 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. One 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.

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 that 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 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 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 modelling 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 behaviour.

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, $Q : D \rightarrow [0, 1]$, and $\sum_{t \in D} Q(t) = 1$. Operational testing[1] then proceeds by independently selecting points from the input domain with these probabilities. In many applications, a point-by-point operational profile

[1] Operational testing is sometimes called random testing, but this 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.

is far too detailed to obtain, and even a crude approximation requires considerable developer effort [17]. However, for our theoretical treatment, the profile Q is a central concept.

Informally, the operational profile can be thought of as characterising 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

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

Of course, in practice we do not know what the labellings 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:

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

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 labelled ϕ and σ. Operational testing only takes account of the operational profile in the selection of tests. Debug testing, on the other hand, seems to take account of the labeling also: 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 give such points a greater chance of being selected than they would have in operational testing (with the points that are believed to be more likely to be σ points having 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."

"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 labelled "ϕ". A perfect debug testing strategy would assign probability zero to all points labelled "σ". In practice, debug testers develop distributions based on heuristics that they hope will give high 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 test cases be drawn from the i^{th} subdomain, for some $T_i \geq 1$.

In a number of practical testing methods, the subdomains are based on analysis of the specification (*specification-based*). 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 placed in 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).

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 case 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 characterise precisely). They then check whether the requisite number of points has been selected from each subdomain and, if not, select additional test cases. In case 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 the first approach, whereas functional testing techniques are usually more amenable to the second approach. 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 first 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.

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.

The Analytical Context

Reliability in the technical sense is characterised 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 "debuggers' 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 favouring 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 fail. However, it might happen instead that many fixes from debug testing increase reliability in operation less than a few from 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 [16, 3], only debug testing has any significant chance of inducing failures and thus allowing the removal of failure regions. However, it may still happen that debug tests encounter only failure points whose probability in the operational profile is so low that fixes are worthless. 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.

Recall that a failure region is a collection of failure inputs that some change fixes exactly. In the examples that follow, we 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, although it is unrealistic: a debugger may use information about how a failed test was chosen in order to figure out how to fix the problem, and such "cues" may be beneficial or misleading depending on both the test method and the failure set of the program. We also assume that failure regions are disjoint, and all test failures are detected (that is, there is a perfect oracle). So, each test failure deterministically causes one failure region to be removed. 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.

The *failure rate* of a failure region is the probability that an element of that region will be selected when one input is selected 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 expected value of the program failure probability as a random variable Θ, after a testset of size T tests has been applied. The simplest form of comparison assumes that equal effort is spent on both testing methods, and that the effort is measured by T.

Although these 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.

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 the 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 $d > q$.

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[2] for subdomain D_i:

$$d^i = \sum_{t \in F \cap D_i} V_i(t), \tag{9}$$

Then

$$P(\Theta = 0) = 1 - \prod_{i=1}^{n} (1 - d^i)^{T_i} \tag{10}$$

and

$$E(\Theta) = q \prod_{i=1}^{n} (1 - d^i)^{T_i}. \tag{11}$$

For comparison with operational testing, equation (8) can be compared with (11) 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 [14] 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. Explicit use of failure region(s) makes our model capable of analysing 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 $0 < q < 1$.

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

- 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).

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,

$$d^k = \frac{\sum_{t \in F} Q(t)}{\sum_{t \in D_k} Q(t)}. \tag{12}$$

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 is relatively neglectful of D_k, that is, $T \sum_{t \in D_k} Q(t) \ll T_k$, or substituting $T_k = T/n$,

$$\sum_{t \in D_k} Q(t) \ll \frac{1}{n}. \tag{13}$$

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

$$\begin{aligned}
q \prod_{i=1}^{n} (1 - d^i)^{T_i} &= q(1 - d^k)^{T_k} && (14)\\
&= q\left(1 - \frac{\sum_{t \in F} Q(t)}{\sum_{t \in D_k} Q(t)}\right)^{T/n} && (15)\\
&> q\left(1 - \frac{\sum_{t \in F} Q(t)}{1/n}\right)^{T/n} && (16)\\
&\approx q\left(1 - T \sum_{t \in F} Q(t)\right) && (17)\\
&\approx q(1 - q)^T, && (18)
\end{aligned}$$

where the last term is the expected value of the failure probability for operational testing. (The approximations in (17) and (18) require that d^k and q are small, using $(1 + x)^y \approx 1 + yx$ for small x.)

To paraphrase, we have captured the situation where a subdomain includes the only failure region, and under plausible assumptions debug testing is more likely to lead to the best reliability. Intuitively, the subdomain D_k is chosen to be "failure prone," and is relatively neglected by operational testing relative to debug testing.

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 $T_k = T/n$,

$$\sum_{t \in D_k} Q(t) \gg \frac{1}{n}. \tag{19}$$

Then using (19) instead of (13) in equation (16) above reverses the inequality and gives that operational testing is superior to debug testing.

Although these two cases are intuitively obvious, and can be obtained using the failure-detection measure of [14], they demonstrate that our model is useful, and in the section on Multiple Failure Regions, Debugging with Subdomains below they will be combined to demonstrate that good failure detection does not imply the best reliability.

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

$$E(\Theta) = \sum_{i=1}^{m} q_i (1 - d_i)^T \tag{20}$$

for debug testing, and

$$E(\Theta) = \sum_{i=1}^{m} q_i (1 - q_i)^T \tag{21}$$

for operational testing, since the failure probability of the debugged program is the sum of the failure rates of the undetected failure regions.

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 "debuggers' 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 analyse 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 minimise

$$q \sum_{i=1}^{m} (1-d_i)^T, \tag{22}$$

under the condition that $\sum d_i = mq$, requires $d_i = q$. More generally, we have proved that an optimal debug method under the condition $\sum d_i = K$ would be one that made the quantities $(1-d_i)^T$ proportional to the q_i values. The planned number of tests thus affects which test method is to be preferred.

Multiple Failure Regions, Debugging with Subdomains

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

$$d_j^i = \sum_{t \in F_j \cap D_i} V_i(t). \tag{23}$$

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 the debugger's nightmare: detection and removal of many minor problems, while failing to detect the serious problems.

The two special cases described 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 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 $D_1 \subset F_1$, $D_2 \subset F_2$ 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. Consider the following different measures of the relative worth of the two testing regimes:

1. The *probability of finding a failure* with debug testing is about

   $$1-(1-d_1^1)^{T/2}(1-q)^{T/2}, \tag{24}$$

   while with operational testing it is

   $$1-(1-q)^T. \tag{25}$$

   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,

   $$E(\Theta) = q_1(1-q_1)^T + q_2(1-q_2)^T. \tag{26}$$

   Let us consider small values of T, such that the first summand is much smaller than the second one. Then, for operational testing:

   $$E(\Theta) \approx q(1-q)^T. \tag{27}$$

   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,

   $$\begin{aligned} E(\Theta) &= q_1(1-d_1^1)^{T/2} + q_2(1-q_2)^{T/2} && (28)\\ &\approx q(1-q)^{T/2}. && (29)\end{aligned}$$

   Comparing (27) and (29), operational testing results in much better delivered reliability of the software.

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

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, $d_1^1 \gg q_1$, but $d_i^i \approx 0, i \neq 1$.) 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 causing 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 yields better delivered reliability.
>
> We omit the formulas for lack of space, but the following is a typical numerical example: $m = 20$ subdomains and failure regions, $q_i = 10^{-4}, i \neq 1$, and $q_1 = 10^{-3}$, $d_1^1 = 0.05$, with a test run of 400 tests. Operational testing is more likely to detect a failure (by 0.0025 to 0.0023), yet debug testing has a better $E(\Theta)$ (by 0.64 to 0.69).

This example is quite unlike the previous one favoring operational testing, in that it appears contrived, yet does not produce a substantial difference between the methods. 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.

These cases illustrate the extra complexity of the situations that can be analysed using failure regions and the expected value of the delivered reliability.

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 analysed in two ways, with and without identifying subdomains for debug testing. This paper generalises and extends the "random vs. partition" studies that followed from the work of Duran and Ntafos [8]. We have analysed a number of special cases, showing that the theory can capture and inform our intuition about the strengths and weaknesses of the two testing schemes.

The debug tester always has the potential advantage that by adjusting the test profile and subdomain definitions, the behaviour of debug methods might improve. 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 yield much lower expected failure probability than that yielded by 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 organisation, 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 operation 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 the debug technique being used consistently turns up such problems, it may be counterproductive to use it.

It is sensible to expect that different testing methods will prove optimal for different organisations, different software projects and different stages in a project. So, research cannot offer decision makers a best testing method 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. For comparing testing methods, the direct experimental approach of measuring the costs and achieved reliability levels on parallel testing campaigns with different methods can be 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 of disjoint failure regions, 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 measured or 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. Frankl and S. 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. Frankl and E. J. Weyuker. Provable improvements on branch testing. *IEEE Trans. on Soft. Eng.*, 19(10):962–975, oct 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] B. Jeng and E. J. Weyuker. Analyzing partition testing strategies. *IEEE Trans. on Soft. Eng.*, 17:703–711, 1991.

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

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

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

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

[19] 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.

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