5.1 Ideal Test Cases

At present, there does not exist a test method that allows us to conclude, from a successful test, that a program does not contain any error. If such a method exists, what properties it should have? This chapter presents a theoretical result in this regard, due to Goodenough and Gerhart [GOGE77].

To facilitate discussion, let

- \( F \) be a program,
- \( D \) be the input domain of \( F \),
- \( F(d) \) denote the result of executing \( F \) with input \( d \in D \),
- \( \text{OUT}(d, F(d)) \) specify the output requirement for \( F \) (i.e., \( \text{OUT}(d, F(d)) \) is true if and only if \( F \) terminates cleanly for \( d \) and \( F(d) \) is an acceptable result),
- \( \text{OK}(d) \) be an abbreviation for \( \text{OUT}(d, F(d)) \), and
- \( T \) be a subset of \( D \).

Elements of \( T \) are called test cases. \( T \) constitutes an ideal set of test cases if \( (\forall t)_{T} \text{OK}(t) \supset (\forall d)_{D} \text{OK}(d) \). If we can find an ideal set of test cases, we can conclude from a success test that the program contains no error.

A test using \( T \) is said to be successful if the program executes correctly with every element of \( T \). Formally,

\[
\text{SUCCESSFUL}(T) \equiv (\forall t)_{T} \text{OK}(t).
\]

Typically, a set of test cases is a subset of \( D \) selected to satisfy some test-case selection criterion, \( C \), which is usually a predicate over \( 2^{D} \), the power set (i.e., the set of all subsets) of \( D \).

A test-case selection criterion, \( C \), is said to be reliable if and only if programs succeed or fail consistently when executing a set of test cases satisfying \( C \). Formally,

\[
\text{RELIABLE}(C) \equiv (\forall T_1)_{2^{D}} (\forall T_2)_{2^{D}} ((C(T_1) \land C(T_2)) \supset (\text{SUCCESSFUL}(T_1) \equiv \text{SUCCESSFUL}(T_2)))
\]

A test-case selection criterion, \( C \), is said to be valid for a particular program if and only if there exists a set of test cases satisfying \( C \) which will cause the program to fail the test if the program is incorrect. To be more precise,
VALID(C) ≡ (∃d)D(¬OK(d)) ⊃ (∃T)2D(C(T) ∧ ¬SUCCESSFUL(T))

Note that validity does not imply that every set of test cases selected with C will cause the program to fail the test if the program is incorrect.

To clarify these concepts, let us consider a program with a single integer input. The program is intended to double the input, but instead, it squares the input. To express it in the formalism introduced above, F(d) = d × d for all d ∈ D and OK(d) ≡ (F(d)=d + d).

Note that OK(0) and OK(2), but ¬OK(d) for all d not in {0, 2}. Listed below are a few possible test-case selection criteria:

\[ C_1(T) ≡ (T=\{0\}) ∨ (T=\{2\}) \]  (reliable but not valid)
\[ C_2(T) ≡ (T=\{t\}) ∧ (t ∈ \{0,1,2,3,4\}) \]  (not reliable but valid)
\[ C_3(T) ≡ (T=\{t\}) ∧ (t ∈ \{1,3,4\}) \]  (reliable and valid)
\[ C_4(T) ≡ (T=\{t_1, t_2, t_3\}) ∧ (D ⊇ T) \]  (reliable and valid)

Goodenough and Gerhart [GOGE77] have shown that a successful test constitutes a direct proof of program correctness, if it is done with a set of test cases selected by a test criterion which is both reliable and valid. Formally,

**Theorem 5.1.1:**

\[ (∃C)(VALID(C) ∧ RELIABLE(C) ∧ (∃T)2D(C(T) ∧ SUCCESSFUL(T))) ⊃ SUCCESSFUL(D) \]

In general, it is difficult to prove the validity and reliability of a test-case selection criterion. In some cases, however, it may become trivial. For example, the proof of validity becomes trivial if C, the test-case selection criterion, does not excludes any member of D, the input domain, from being selected as a test case. If C does not allow any member of D to be selected, then C can be valid only if the program is correct. Thus, in that case, it is required to prove the program's correctness in order to prove the validity of C. The proof of reliability becomes trivial if C requires selection of all elements in D because in that case there will only be one set of test cases. The proof also becomes trivial if C does not allow any element from D to be selected.

**Theorem 5.1.2:** (Howden's Result) There exists no computable procedure H which, given an arbitrary program F with domain D, can be used to generate a nonempty finite set D ⊇ T such that

\[ (\forall t)T(OK(t)) ⊃ (∀d)D(OK(d)). \]

**Proof:** If such a procedure existed, it would be possible to use it to determine the equivalence of arbitrary primitive recursive functions. It is known that no such procedure exists. The argument
runs as follows. Let $F_1$ and $F_2$ be any two programs in a language for constructing programs which compute primitive recursive functions. Let $f$ be the function computed by $F_2$. Since $F_1$ and $F_2$ are primitive recursive functions, they terminate and $H$ can be used to determine the correctness of $F_1$ for calculating $f$. This is equivalent to determining the equivalence of $F_1$ and $F_2$. ♦

### 5.2 Debugging vs. Operational Testing

In [FHSL98] in Frankl et al. examined the relationship between debug testing and operational testing.

There are two main goals in testing a program. On the one hand, testing can be seen as a means of achieving reliability: here the objective is to probe the software for faults 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 remove faults, and improve the quality of 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 faults 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 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 faults 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 in finding faults, but the faults found may be of the sort that rarely causes failures in production runs. In that event, removal of those faults would not improve the reliability of the software appreciably. Operational testing, on the other hand, is designed to detect failures that are most likely to occur in production runs. The faults so uncovered would be more important in that their removal will improve the reliability of the software more significantly.

1 Notice that, throughout the paper, we treat the "importance" of a fault 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

---

Program Analysis and Testing 5-3 © J. C. Huang 2004
The effectiveness of a debug testing method depends on how well the underlying assumptions of the method about faults match the reality, whereas the effectiveness of an operational test depends on how well the operational profile used matches that in reality.

Operational testing is attractive because it produces information for reliability assessment as well as reliability improvement.

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.

In [FHLS98] Frankl et al. discussed testing effectiveness based on the reliability of a program after it is tested. This measure was used to compare debug testing to operational testing, exploring circumstances under which each technique is likely to yield superior reliability.

There is a deeply rooted belief among program testers and debuggers that the process of probing software for faults 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.

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 [COMI90, SEBB87], 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 [BEIZ95] has attacked Cleanroom as "lead[ing] to false confidence."

greatly from one fault 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.
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 [HOLL94, PIOC93, FRWE93]. 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 [GECR93]. 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 [DUNT84, HATA90, JEWE91, TSDN93, CHYU94, CHYU96]. 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 [FHLS98] a new analytical approach was taken 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.

**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 \( \Sigma_{t \in D} Q(t) = 1 \). 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 is far too detailed to obtain, and even a crude approximation requires considerable developer effort [MUSA93]. 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

---

\(^1\) 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.
indicate that such a point, when selected, results in a failure), or \( \sigma \) (for success). Define the indicator variable

\[
\delta(t) = 1 \text{ if } t \text{ has label } \phi \\
= 0 \text{ if } t \text{ has label } \sigma
\]

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

\[
\theta = \sum_{t \in D} Q(t) \delta(t) = \sum_{\text{failureset}} Q(t)
\]

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 \( \theta \) will have to be statistical, and come from the results of a test set 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 \( \theta \).

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 labeled \( \phi \) and \( \sigma \). 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 \( \phi \) 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 \( \sigma \) 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, \( x_j \), with probability of selection in operation \( p_j \). The operational tester says "I don't know anything about the chance that \( x_j \) will have label \( \phi \), so I will select it with probability \( p_j \); that way, if it has a label \( \phi \), 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 \( x_j \) 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 labeled "\( \phi \). 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.)

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.

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 \( \phi \) 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 predetermined 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

---

1 "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).
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 $i$.

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 test set is selected by sampling according to the operational profile, then direct estimates of the failure probability may be obtained. If a test set 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 [LIST93, BUF93], 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 [LIST93, BUF93].

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.
[DUNT84]. We have analyzed 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.

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.

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 ultrahigh 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.