Predicting Dependability by Testing

Dick Hamlet

Portland State University
Department of Computer Science
Center for Software Quality Research
PO Box 751
Portland, OR 97207
Internet: hamlet@cs.pdx.edu
Phone: (503) 725-3216
FAX: (503) 725-3211

May 24, 1995

Abstract

In assessing the quality of software, we would like to make engineering judgements similar to those based on statistical quality control. Ideally, we want to support statements like: "The confidence that this program's result at X is correct is p," where X is a particular vector of inputs, and p is a probability obtained from measurements of the software (perhaps involving X). For the theory to be useful, it must be feasible to predict values of p near 1 for many programs, for most values of X.
Manuel Blum's theory of self-checking/correcting programs has exactly the right character, but it applies to only a few unusual problems. Conventional software reliability theory is widely applicable, but it yields only confidence in a failure intensity, and the measurements required to support a correctness-like failure intensity (say 10^{-9}/demand) are infeasible. Jeff Voas's sensitivity theory remedies these problems of reliability theory, but his model is too simple to be very plausible. In this paper we combine these ideas: reliability, sensitivity, and self-checking, to obtain new results on "dependability," plausible predictions of software quality. The most important results are the application of reliability measurements to support Blum's theory, and use of the self-checking paradigm to eliminate reliance on a user profile and on a test oracle.

Keywords: software testing theory, (ultra)reliability, sensitivity, self-checking, dependability

1 Introduction — Need for "Software Dependability"

Software quality is much discussed in the 1990s, often in the context of improving the process by which software is developed. It is certainly true that this process is often chaotic and the resulting software of uncertain quality. Attention to codifying and monitoring development is in order. However, without a precise definition of software quality, process improvement will only eliminate the worst examples, and cannot support real engineering goals. What software engineers need is a way to directly assess software itself, to determine if it is "good enough" to fulfill its intended purpose.

1.1 Process vs. Product

A laudable interest in the software development process *per se* has the unfortunate side effect of downplaying its technical details. Attention shifts to the process itself, where the technical connection between what is done and what is produced may be lost. Organization and systematic, monitored procedures are a part of successful engineering, but the essence of engineering is application of scientific knowledge. All the organization in the world will not save a process based on an erroneous understanding of reality.

The only real measure of quality software is a "product" measure. It is the software that counts, not its pedigree. Of course, attention to the process can be helpful in producing a good product, but only if there are solid, demonstrable connections between what is done and what results from doing it. To carry out procedures for their own sake — for example, because they can be easily monitored and adjusted — is to mistake form for substance.

The substance of software is embodied in testing of programs just before they are released. Such tests make no assumptions about the process, but directly measure its result. Release testing assesses not only a particular piece of software, but is also needed to validate methods used in its development. Theoretical arguments for adopting a particular process should demonstrate that if that process is used, release tests will be more likely to succeed; and, empirical validation of the process should show that indeed they do succeed.

1.2 What Quality Measure is Needed?

The fundamental measure of quality is proper operation of a software product. Quality software does not fail. Failure can take the form of a "crash" after which the software cannot be used without some kind of drastic restart (and often a loss of information or invested time); failure can be wrong answers delivered with all the trappings of computer authority (and thus the more dangerous); and failure can be in the performance dimension — the software is too slow to be useful.

In this paper we identify software quality with the absence of failure. However, we want an engineering measure, not the binary ideal of "correct"/"not correct." Whether or not one believes that it is possible to create "zero-defect" software, to *demonstrate* correctness is impractical. Proof methods might do so in principle, but they have failed in practice. Part of the reason is that theorem provers are too inefficient and too hard to use; a deeper reason is that the formal specifications required for verification are at least as difficult to create, and as error-prone, as programs. The alternative to formal verification is testing, but tests are only samples of software's behavior, and the best we can hope for is that they establish some kind of statistical confidence in quality.

There is no standard term for good software in the sense of "unlikely to fail." "Reliable" has a related but different technical meaning in engineering (see Section 2.3). Parnas has used the term "trustworthy" [PvSK90], but he includes the severity of failure: Parnas's trustworthy software is unlikely to have *catastrophic* failures. However, catastrophe is easiest to recognize after the fact. Here we will be concerned with prediction of unspecified future events, whose severity is unknown. We will use the technical term "dependable" for the intuitive idea "unlikely to fail."

1.3 Character of "Dependability"

The quality measure we hope to define is essentially a confidence in perfection. Some desirable characteristics are:

Statistical. For engineering tradeoffs, a quantitative dependability measure must be probabilistic. The statistical character of dependability is not that a program is (say) "95% correct," but rather that assessment of the program gives (say) "95% confidence in its correctness." The difference is in the event space: it is not the input space of the program (or a time-sequence of inputs), but rather the measurement space.

Usage independent. It is a commonplace that dependability changes with software usage. In one application, a program works perfectly, while in another it fails. But the variety of usage, and the difficulty of quantifying and comparing its conditions, dictate that dependability be assessed without recourse to any knowledge of software's eventual use. This requirement is particularly important in dealing with re-usable or off-the-shelf software, whose future usage is unknown.

No use of hindsight. After software has failed, analysis that was impossible before the failure becomes obvious. For example, failures may be rated by severity, say as "catastrophic" or not. But when all failures lie in the future, these judgements are more difficult to make. Dependability must not involve factors that only Monday-morning quarterbacks can see.

Measured by successful testing. The final pre-release testing of software ought to include nothing but successful tests. When there is a failure, development can continue with debugging and retesting. Hence dependability measurement has to be based on a successful test before release.

The theory of testing for software reliability qualifies as a statistical theory. (A detailed exposition is given in Section 2.3 below.) But reliability theory is built around the idea of sampling from a user profile. Furthermore, reliability also involves a period of operation, which is suspect for software. No matter how small the wear and tear, all physical systems eventually fail; that is, their long-term reliability is zero. But software need not fail, if its designers' mistakes can be controlled. Reliability testing supports statements like: "The probability that this program can fail in normal usage is less than .001, with a confidence bound of 95%." In contrast, we want the statement: "The dependability of this program is .999" to mean that a test sample establishes 99.9% confidence that the program cannot fail.

2 Previous Work

Because our results rely on three less well known (two of them recent) developments in testing theory, these are briefly presented (along with basic terminology) in this section. Those unfamiliar with these developments may find the exposition helpful. None of this material is new; the reader familiar with it should skip to the summary (Section 2.5) and the new results in the following Section 3.

2.1 Testing Background and Terminology

Terminology

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, which is not very realistic, but which simplifies the discussion. This context is that of a "batch" 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 reality, programs may have complex input tuples, and produce similar outputs. But we can imagine coding each of these into a single value, so that the simplification is not a transgression in principle. Interactive programs that accept input a bit at a time and respond to each bit, programs that read and write permanent data, and real-time programs, do not fit this simple model. However, it is possible to treat these more complex programs as if they used testsets, at the cost of some artificiality. For example, an interactive program can be thought of as having testsets whose members (single tests) are *sequences* of inputs.

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 iff: if $x \in dom(S)$ then on input x, P produces output y such that $(x, y) \in S$. A program meets its specification (everywhere) iff it meets it on all inputs. Note that where $x \notin dom(S)$, that is, when an input does not occur as any first element in the specification, the program may do anything it likes, including fail to terminate, yet still meet the specification. Thus S defines the input domain as well as behavior on that domain.

A program P with specification S *fails* on input x iff P does not meet S at x. A program fails, if it fails on any input. When a program fails, the situation, and loosely the input responsible, is called a *failure*. The opposite of fails is *succeeds*; the opposite of a failure is a *success*.

Programmers and testers are much concerned with "bugs" (or "defects," or "errors"). The idea of "bug" in unlike the precise technical notion of "failure" because a bug intuitively is a piece of erroneous program code, while a failure is an unwanted execution result. The technical term for "bug" etc., is *fault*, intuitively the textual program element that is responsible for one or more failures. However appealing and necessary this intuitive idea may be, it has proved extremely difficult to define precisely. The difficulty is that faults have no unique characterization. In practice, software fails for some testset, and is changed so that it succeeds on that testset. The assumption is made that the change does not introduce any new failures (an assumption false in general). The "fault" is then defined by the "fix," and is characterized, 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. So "the fault" is not a precise idea. Nevertheless, the terminology is useful and solidly entrenched.

The Oracle Problem

Evidently the most important aspect of any testing situation is the determination of success or failure. But in practice, the process is error-prone. If a program fails to complete its calculation in an obvious way (for example, it is aborted with a message from the run-time system), then it will likely be seen to have failed. But for elaborate output displays, specified only by an imprecise description in natural language (a very common real situation), a human being may well fail to notice a subtle failure. In one study, 40% of the test failures went unnoticed [BS87].

An *oracle* for specification S is a binary predicate J such that $J(x, y)$ holds iff: either $x \notin dom(S)$ or $(x, y) \in S$. (That is, J is a natural extension of the characteristic function of S.) If there is an algorithm for computing J then the oracle is called *effective*. Thus, given a program and a test, an effective oracle can be used to decide mechanically if the program meets its specification at this point.

Testing theory, being concerned with the choice of tests and testing methods, usually ignores the oracle problem. It is typically assumed that an oracle exists, and the theoretician then glibly talks about success and failure, while in practice there is no oracle but imperfect human judgement.

Although it would not seem that the dependability theory we are seeking should bear on the oracle problem, it does, because confidence in a computed result may make an oracle unnecessary. This point is further discussed in Section 4.4.

2.2 Blum's Idea of Self-checking Programs

Manuel Blum has proposed [BK89, BW94] an idea that is almost exactly what we want to mean by "dependability." He argues that software users are interested in a particular execution of a particular program only — they want assurance that a single result can be trusted. Blum has found a way to sometimes exploit the low failure intensity of a "quality" program to gain this assurance. Roughly, his idea is that a program should check its output by performing redundant computations. Even if these make use of the same algorithm, if the program is "close to correct," it is very unlikely that a sequence of checks could agree yet all be wrong.

There is one serious drawback to self-checking at run time: if in fact a program discovers an inconsistency in the checks, it has experimentally determined that the assumption of "close to correct" does not hold. Then instead of reporting that the result is likely to be correct, it can only report: "Don't trust this!"

Self-checking represents a viewpoint quite different from the usual testing, because it is a *pointwise* view of quality. Testing attempts to predict future behavior of a program *uniformly*, that is, for all possible inputs; Blum is satisfied to make the prediction one point at a time (hence to be useful, the calculation must be made at run time, when the point of interest is known). All of testing's problems with user profile, test-point independence, etc., arise from the uniform viewpoint, and Blum solves them at a stroke. Testing to uniformly predict behavior suffers from the difficulty that for a high-quality program, failures are "needles in a haystack" — very unlikely, hence difficult to assess. Only impractically large samples have significance. Blum turns this problem to advantage: since failures are unlikely, for one input the calculation can be checked using the same program. The results will probably agree *unless they are wrong* — a wrong result is nearly impossible to replicate.

Blum's pointwise notion of correctness probability, and its measurement by self-checking at run time, will be our definition of dependability (Section 3).

2.3 Nelson Theory of Software Reliability

Reliability is the fundamental statistical measure of engineering quality, expressing the probability that an artifact will fail in its operating environment, within a given period of operation.

2.3.1 Random Testing

In random testing, a testset is an unbiased sample taken from a program's input space. To generate inputs "at random," pseudorandom number generation algorithms have long been used [Knu81]. Pseudorandom numbers from a uniform distribution can be used as test inputs if a program's range of input values is known. This range is ultimately fixed by hardware limitations such as word size, but it is better if the specification restricts the input domain. For example, a mathematical library routine might have adequate accuracy only for a certain range given in its specification. A uniform distribution, however, may not be appropriate.

2.3.2 Operational Profiles

Statistical predictions from sampling have no validity unless the sample is "representative," which for software means that the testset must be drawn in the same way that future invocations will occur. An *input probability density* $d(x)$ is needed, expressing the probability that input x will actually occur in use. Given continuous density function d, the *operational distribution* $F(x)$ is the cumulative probability that an input will occur in actual use:

$$F(x) = \int_{-\infty}^{x} d(z)dz.$$

To generate a testset "according to operational distribution F," start with a collection of pseudorandom reals r uniformly distributed over [0,1], and generate $F^{-1}(r)$. For a detailed presentation, see [Ham94].

The distribution function d should technically be given as a part a program's specification. In practice, the best that can be obtained is a very crude approximation to d called the *operational profile*. The program input space is broken down into a limited number of categories by function, and attempts are made to estimate the probability with which expected inputs will come from each category. Random testing is then conducted by drawing inputs from each category of the profile (using a uniform distribution within the category), in proportion to the estimated usage frequency.

2.3.3 Software Reliability Theory

If a statistical view of software failures is appropriate, statistical parameters can be measured for a program using random testing. Inputs are supplied at random according to the operational profile, and the *failure intensity* is the long-term average of the ratio of failed runs to total runs. An exhaustive test might measure failure intensity exactly. But whether or not failure intensity can be estimated with less than exhaustive testing depends on the sample size, and on unknown characteristics of programs. Too small a sample might inadvertently emphasize incorrect executions, and thus to estimate failure intensity that is falsely high. The more dangerous possibility is that failures will be unfairly avoided, and the estimate will be too optimistic. When a release test exposes no failures, a failure-intensity estimate of zero is the only one possible. If subsequent field failures show the estimate to be wrong, it demonstrates precisely the anti-statistical point of view. A more subtle criticism questions whether failure intensity is stable — is it possible to perform repeated experiments in which the measured values obey the law of large numbers?

In practice there is considerable difficulty with the operational profile:

1. Usage information may not be available. In the best cases, the profile obtained is coarse, having at most a few hundred usage probabilities for rough classes of inputs.

2. Different organizations (and different individuals within one organization) may have quite different profiles, which may change over time.

3. Testing with the wrong profile always gives overly optimistic results (because when no failures are seen, it cannot be because failures have been overemphasized!).

It is therefore of considerable importance that a fundamental testing theory avoid reliance on operational profiles.

Postulating an operational profile allows us to derive the software-reliability theory developed at TRW by Nelson and others [TLN78]. Suppose that there is a meaningful constant failure intensity Θ (in failures/demand) for a program, and hence a reliability of $e^{-\Theta M}$ over M runs [Sho83]. We wish to draw N random tests according to the operational profile, to establish an upper confidence bound α that Θ is below some level θ. These quantities are related by

$$1 - \sum_{j=0}^{F} \binom{N}{j} \theta^j (1-\theta)^{N-j} \geq \alpha \qquad (1)$$

if the N tests uncover F failures. For the important special case $F = 0$, the confidence α is a family of curves indicated in Figure 1. For any fixed value of N it is possible to trade higher confidence in a failure intensity such as h for lower confidence in a better intensity such as h'.

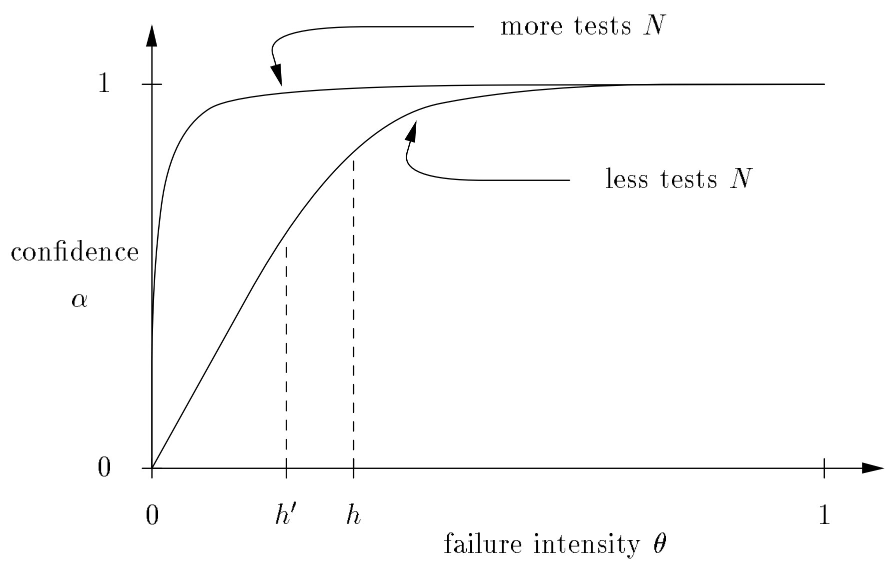

Figure 1: Confidence in failure intensity based on testing

Another way to derive the relationship between confidence, testset size, and failure intensity, is to treat the test as an experiment checking the hypothesis that the failure intensity lies below a given value. Butler and Finelli [BF91] obtain numerical values similar to those predicted by Equation (1) in this way. They define the "ultrareliable" region as failure intensities in the range 10^{-8}/demand and below, and present a very convincing case that it is impractical to gain information in this region by testing. From Equation (1), at the 90% confidence level, to predict a failure intensity of Θ requires a successful testset of size roughly $2/\Theta$, so to predict ultrareliability by testing at one test point each second around the clock would require three years. Ultrareliability is appropriate for safety-critical applications like commercial flight-control programs and medical applications; in addition, because of a large customer base, popular PC software can be expected to fail within days of release unless it achieves ultrareliability.

Thus software reliability theory provides a pessimistic view of what can be achieved by testing, compounded by deficiencies in the theory. If an inaccurate profile is used for testing the results are invalid, and they always err in the direction of predicting better reliability than actually exists.

2.4 Voas's Sensitivity Measurements

Jeff Voas has an idea that gives testing a significant twist.

2.4.1 "PIE" Model of Failure

Sensitivity [Voa92] captures the intuition that a testset is good at exposing a program's faults as execution failures. Sensitivity is a *lower* bound probability of failure if software contains faults, based on a model of the process by which faults become failures. A sensitivity near 1 indicates a program that "wears its faults on its sleeve": if it can fail, it is very likely to fail under test.

To define sensitivity as the conditional probability that a program will fail under test *if it has any faults*, Voas models the failure process of a fault in one program location. For the fault to lead to failure, its location must be executed, it must produce an error in the local state, and that error must then persist to affect the result. Voas calls his model "PIE" for Propagation, Infection, and Execution. The sensitivity of a program location can be estimated by executing the program as if it were being tested, but instead of observing the result, counting the execution, state-corruption (infection), and propagation frequencies. Sensitivity analysis thus employs a testset, but not an oracle. Voas and his co-workers have designed and constructed a tool (PiSCES) that measures sensitivity, given a program and a testset.

When the sensitivity measured by PiSCES is high at a location, it means that the testset causes that location to be executed frequently; these executions have a good chance of corrupting the local state; and, an erroneous state is unlikely to be lost or corrected. The high sensitivity does not mean that the program is likely to fail; it means that if the location has a fault then the testset is likely to expose it.

2.4.2 The "Squeeze Play"

By combining sensitivity analysis with reliability theory [VM92], it is possible to obtain a measurement of probable correctness that unlike Blum's (Section 2.2) is uniform. Suppose that all the locations of a program are observed to have high sensitivity, using a testset drawn from the operational profile. Then suppose that this same testset is used in successful random testing. (That is, the results are now observed, and no failures are seen.) The situation is then that (i) no failures were observed, but (ii) if there were faults, failures would have been observed. The conclusion is that there are no faults. This "squeeze play" plays off sensitivity against reliability to gain confidence in correctness.

Figure 2 shows the quantitative analysis of the squeeze play between reliability and sensitivity [HV93]. In Figure 2, the falling curve is the confidence from reliability testing (it is $1 - \alpha$ from Figure 1); the step function comes from observing a sensitivity h. Together the curves make it unlikely that the chance of failure is large (testing), or that it is small (sensitivity). The only other possibility is that the software is correct, for which $1 - d$ is a confidence bound, where d is slightly

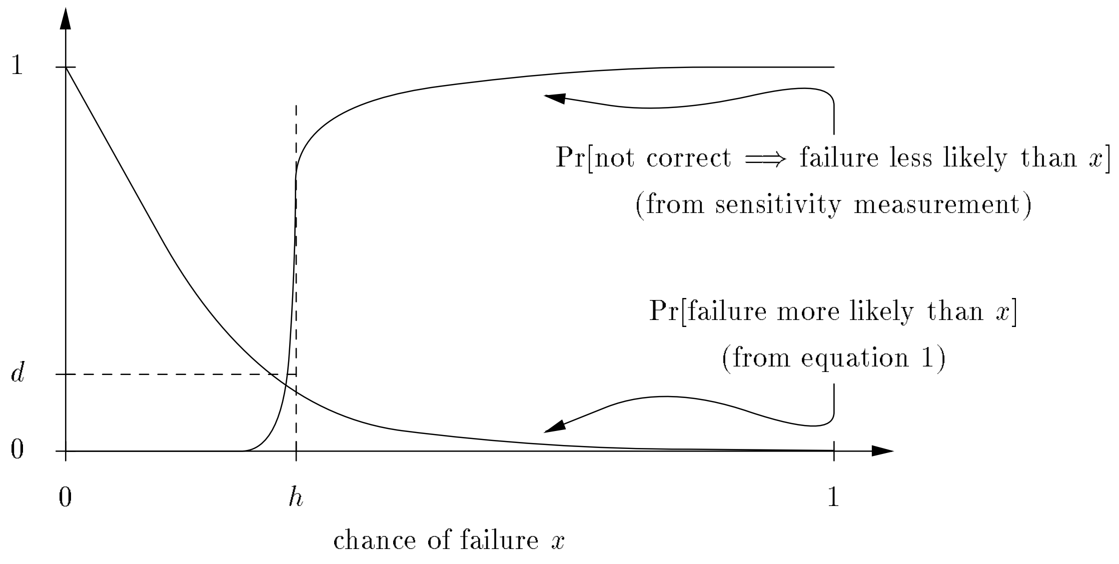

Figure 2: 'Squeeze play' between sensitivity and reliability

more than the value of the falling curve at h. Confidence that the software is correct can be made close to 1 by forcing h to the right [HV93]. For example, with a sensitivity of .001, a random testset of 20,000 points predicts the probability that the tested program is not correct to be only about 2×10^{-9}.

The squeeze play could serve as a basis for defining dependability, although we do not take this course. The pros and cons are discussed in Section 4.1.

2.5 Deficiencies and Strengths of Existing Work

We summarize the three ideas of the previous sections.

Blum's self-checking

- A pointwise measure of probable correctness.
- Prediction: "Trust this result with probability p." (Or, "Don't trust this result!")
- Cost (at runtime) is a small multiple of execution time for each checked result.
- Applies only to (mathematically) well structured problems, and requires "high" program quality.
- Requires no oracle, no user profile.

Software reliability

- Analogous to engineering reliability of physical devices.
- Prediction: "Confidence C that the failure intensity is below p."
- It is infeasible to assess reliability better than about 10^{-6}/demand.
- Universally applicable; independent of program characteristics.
- Requires an oracle, and an accurate user profile.

Voas's squeeze play

- A uniform measure of probable correctness.
- Prediction: "The probability that this program is correct is p."
- Cost is high, but probably not infeasible.
- Based on a simplistic model, but widely applicable.
- Requires no oracle; requires a user profile.

3 Results — Putting the Theories Together

The ideas of self-checking, reliability, and sensitivity complement each other. We now exploit their combination to obtain new dependability results. We choose to take Blum's formulation as fundamental, and to extend its application using reliability and sensitivity. This choice of a "pointwise" definition of dependability is not unexceptionable; for further discussion see Section 4.1.

Definition : The *dependability* of program P at input X is the probability that P is correct (as defined by its specification) at X.

For this definition to be usefully different from a "uniform" definition in which the dependability is defined for all inputs, a dependability prediction must be supported by runtime calculations, in the manner of Blum's self-checking idea [BW94]. Self-checking has been applied to only a few problems, problems whose input data has a decomposition theory, so that random variations on a computation can be performed and their results compared.

3.1 "Pointwise" Sensitivity

Voas's PIE theory of sensitivity (Section 2.4.1) becomes a "PI" theory when a program is executed for a single input X. Performing the analysis (as the PiSCES tool does) yields a sensitivity t estimating the probability that if the program can fail then it has done so (on X). Thus $1-t$ is a lower bound on the dependability: in the worst case that there is a fault, $1-t$ is the chance that failure does not result. Thus *low* sensitivity is desirable.

In order to make use of *high* sensitivity, there must be some independent support for the computed result being correct; this case (which has something in common with the squeeze play) is discussed below in Section 3.3.2.

3.2 Incorporating Reliability in Self-testing

Blum assumes that programs incorporating self-checking have been shown to behave correctly on "random" variations of inputs. Conventional reliability theory seems the way to measure this required behavior, but reliability is not directly applicable, because a reliability measurement gives a pair of probabilities (a confidence bound on a failure-intensity ceiling). We now explain how to use this information in support of a self-checking result, for a problem to which Blum's theory already applies (e.g., matrix multiplication). The extension to a broader range of problems is considered in Section 3.3 below.

The essence of Blum's idea is that if failures are sparse in the input space, then random variations of an execution are extremely unlikely to hit on a sequence of failures only, so a long sequence of agreeing variations indicates that the result is correct. For this argument to be applied, N variant executions must *all* be failures for them to agree if one (the original result) is wrong. Reliability theory can be used to calculate the probability that N random variations are all failures.

Suppose that a successful reliability test of size K has been previously conducted, using a uniform distribution (if the Blum self-checks use this distribution, as they usually do). Had N random variant executions been the beginning of a set of K executions, the self-check would have constituted a repetition of the reliability test. For all N initial points to be failures would make the failure intensity at least N/K in this repetition. In the special case of a successful test ($F = 0$), Equation (1) (Section 2.3.3) becomes

$$1 - (1 - \frac{N}{K})^K \geq \alpha. \qquad (2)$$

The meaning of the upper confidence bound α is the probability that repeating the reliability test will show a failure intensity less than N/K. Hence α is the dependability.

Rearranging Equation (2) and taking natural logs,

$$\log(1-\alpha) = K\log(\frac{K-N}{K}) = K(\log(K-N) - \log(K)).$$

Using the definition of log(), the difference of logs is

$$\int_1^{K-N} \frac{1}{t}dt - \int_1^{K} \frac{1}{t}dt = -\int_{K-N}^{K} \frac{1}{t}dt.$$

If K is large and N is near to K, the integrand can be approximated by $\frac{1}{K}$, so the integral is about N/K, and

$$\log(1-\alpha) \approx -N, \qquad \alpha \approx 1 - e^{-N}.$$

The analysis obscures the size K of the required reliability test. Some exact values are shown in Table 1. For example, the second line in Table 1 describes the situation in which a successful

$1-\alpha$	N	K	(e^{-N})
2.7×10^{-5}	10	100	4.5×10^{-5}
4.3×10^{-5}	10	1000	
4.5×10^{-5}	10	10000	
2.0×10^{-9}	20	10000	2.1×10^{-9}

Table 1: Dependability α for a reliability test of size K and N random checks

reliability test (with an oracle and a uniform input distribution) was performed on a self-checking program using 1000 test points. Then at run time, if one calculation performs 10 agreeing self checks, we predict that there is only 4.3×10^{-5} chance that the result is wrong. That is, the dependability is predicted to be 99.9957%. As the table shows, the reliability and self-checking parameters are practical, even in the ultrareliable region. Dependability predictions do not require the infeasible reliability testing that would be needed to predict ultrareliability directly [BF91].

3.3 Extending Self-checking to More Problems

The only examples that have been given of self-checking algorithms are ones in which the problem input may be distorted, and the original result reconstructed from the distorted computation. Thus the argument that a group of random self-checks must all be failures together (or all correct) rests on the equality of their results. This equality is available for only a few well structured mathematical problems.

3.3.1 Using Specification Properties

It is interesting to note that the properties required to apply self-checking are those of the *problem* (that is, the specification), not of a particular algorithm or program whose dependability is being considered. For example, it is decomposition properties of graphs that allow the idea to be used on graph isomorphism [BK89].

For mathematical problems, other ways suggest themselves to argue that a group of results must all be failures if any one is. For example, suppose that the problem requires computing a function F which is differentiable at input t, and within an ϵ-neighborhood of t, the derivative is bounded by $V(t, \epsilon)$. Then within an ϵ-neighborhood of t, F cannot vary by more than $\epsilon V(t, \epsilon)$. Suppose that a program P_F purports to compute F. If its result at t is in error by more than $\epsilon V(t, \epsilon)$, and if random results in an ϵ-neighborhood vary by less than $\epsilon V(t, \epsilon)$, then all must be wrong. Correct behavior by P_F at t requires similar bounded variation. Thus any program for such a function F can be self checked as follows:

> Following the calculation of result Y on input X, repeat the calculation at N randomly chosen points in a neighborhood of X. If any calculation differs from Y by more than the bounded variation specified for F, then Y cannot be trusted. If the variation is bounded, then there are two possibilities: (1) Y is correct to within the tolerance of the bounded variation; or, (2) Y is not correct, but this becomes increasingly unlikely as N increases.

The dependability that arises from possibility (2) depends on the program's "random" behavior; for example, if this has been assessed using reliability measurements, the dependability can be read from Table 1. The correctness tolerance (possibility (1)) may be made arbitrarily small by shrinking the neighborhood of X.

Establishing new properties that allow application of self-checking is specification-based analysis that will be much aided by the existence of a precise specification, particularly one designed for formal manipulation.

3.3.2 High Sensitivity for Self-checking

Suppose that for each member of a set of self-check inputs, the pointwise sensitivity (Section 3.1) is near 1. This means that if any fault is encountered by any of these inputs, results are likely wrong. Or, there may be no faults, and the results correct. In the former case, a dependability bound can be calculated as usual.

Using reliability (to assess "random" behavior) in conjunction with high sensitivity (to argue that the self-check points must all be failures if any are), in the manner of Section 3.2, is a technique applicable to *any* program, since both theories are general. Furthermore, the required measurements appear to be practical. The weakness in such a general theory lies with the sensitivity component. First, only unrealistic faults are modeled by PIE sensitivity, and measurements such as PiSCES performs are themselves only approximations. Second, most programs do not have sensitivities close to 1, and finally, PiSCES may be too slow to use at run time. However, even a weak case for dependability may have merit when all other assessments are infeasible.

4 Discussion

4.1 Drawbacks of the Self-checking Formulation

Our definition for dependability is based on Blum's idea of "pointwise" probability of correctness. The advantage of this formulation is that it provides a plausible interpretation for the dependability, one that sidesteps all the intuitive difficulties in treating deterministic programs as if they had stochastic properties. However, the formulation also has drawbacks.

Most important, in critical applications, it seems inappropriate to defer the prediction of dependability to run time. If the software is inadequate, instead of reporting good dependability, it will announce that a result should not be trusted. What then is the pilot of a plane under computer control, or the operator of a medical monitor, etc., supposed to do? We suggest a partial solution to this difficulty in Section 4.4 below, but it is a basic deficiency of the theory.

Our definition, despite the suggestions made in Section 3.3, is not applicable to software in general, and alternate formulations might have wider application. In particular, Voas's squeeze play offers an alternative. The squeeze play can be used to define dependability as a probability that software is correct, but for *all* inputs, based on extensive pre-release reliability and sensitivity measurements. Thus such a definition would answer the important objection raised in the previous paragraph. It would also apply to any software system with appropriate measurements (roughly, high confidence in a reliability of better than about 10^{-4}/demand, and a sensitivity better than about 0.001)[HV93]. If the combination of sensitivity and reliability is being used as indicated in Section 3.3.2, a definition based on the squeeze play might be preferred, because it gives a better result when the sensitivity is not close to 1. The drawbacks of a squeeze-play-based definition are two-fold: (1) Sensitivity analysis is too simplistic to be plausible, and (2) The squeeze play requires an operational profile in its measurements, where our definition does not. The latter is the more serious drawback, since it would preclude calculating dependability for off-the-shelf components whose application profile is unknown.

4.2 Role of Formal Methods

It is a surprising strength of the dependability ideas that they provide a strong rationale for the use of formal methods in software development.

Sound development methods must be used to create programs whose dependability will be assessed at run time. When the self-checking definition is used with inadequate software, the self-checks do not agree, no information about dependability is available, and the software simply fails. It may be little comfort that we know it has failed.

Formal methods, particularly mathematical formalisms for specification that support analysis, can be used in two ways:

Establishing new properties for checking. Because the properties that must be exploited in self-checks are specification properties, it is obviously an advantage to explore them mathematically. Section 3.3.1 gives an example of such a property.

Proving sensitivity properties. Sensitivity is partly a property of the details of code implementing a software function, but partly of that function itself. For example, Voas has shown that range-cardinality-reducing functions (e.g., Reals $\rightarrow$ Boolean) are inherently of low sensitivity [Voa91]. Formal analysis may be able to establish the necessary software properties to attain the high sensitivity required for some kinds of dependability calculations, such as indicated in Section 3.3.2.

4.3 Experimental Validation

Dependability, as calculated for a particular program according to the definition used here, includes a running "self validation." Each time a program reports that a result is undependable, there will be reason to examine it in detail, find the reason why, and fix the program. Should it be discovered that the dependability calculations are at fault, it provides an immediate counterexample to the theory.

Similarly, should a program report that a result *can* be trusted, and should it eventually be learned that the result is not correct, data accumulates to refute the theory. Since dependability is a probability, it is not impossible that a probably correct result is occasionally wrong. But it had better not happen that events calculated as having a likelihood of 10^{-9} happen daily.

An argument that a dependability theory is correct must be inductive, and thus like validation of any scientific theory, it will never be complete. What is to be expected is a series of refutations of detailed theories, each leading to improvements that answer the objection. The evolving theory will be valuable if it remains stable for periods long enough to direct the construction of useful software.

4.4 Rediscovering User Profiles

Operational profiles play *no* role in the Blum dependability definition or any of its applications suggested in this paper. When reliability measurements are used, they are based on uniform profiles, because these can be used in randomizing the self-checks. (It is an uninvestigated problem to consider if other profiles might be useful for the randomization used in self-checking, but in any case the capricious "user" does not come into it.)

However, if a user profile is known, it can be used with dependability theory. Self-checking software, capable of calculating its runtime dependability values, can be random tested with a profile. It is important to note that this testing requires no oracle, because the dependability itself will serve. For each test input drawn from the profile, the result can be taken to be correct if the self-checks succeed. Figure 1 then predicts the reliability parameters based on the number of successful tests. (Should the program report that self-checking failed, it is not necessarily because the calculated result is wrong, so the reliability obtained will be a pessimistic estimate.) This prediction will of course be subject to the same difficulties as any reliability measurement today (except for the oracle problem). In particular, the profile may be inaccurate, and assessment of ultrareliability will not be feasible. But a software user may be able to get a rough indication of how a program will perform in an unusual environment, and when the software is actually in use, will have the runtime dependability values as a check on each execution.

4.5 Dependability As An Oracle

The idea suggested in Section 4.4, to use self-checking as an effective oracle, is generally applicable to program testing. First, a conventional random test is conducted with a uniform profile (and an oracle) as indicated in Section 3.2. For subsequent tests, successful self-checking is equated with correctness. Table 1 can be used to select appropriate parameters for the random test and for self-checking. As the table shows, the bulk of testing will use self-checking instead of the actual oracle.

The most interesting application is to the testing of components intended for reuse or off-the-shelf embedding in other software. The developer of the component would conduct a uniform-profile random test, and provide a self-checking harness. An end user would then test the component using a profile appropriate to its intended use, in which the self-checking harness serves as oracle. Or, the end user might employ another, or a combination of. testing methods, for example if required to do so by a regulatory agency. (The component might fail these tests, causing this user to buy another component.) If testing is satisfactory, the end user could elect to leave the self-checking code in place (with appropriate actions when checking fails), or remove it for the production version.

5 Summary

We have taken a definition of software dependability based on Blum's idea of pointwise correctness probability, and his idea of estimating it at run time.

Software reliability theory can be used to replace Blum's somewhat vague requirement that programs "be correct on random inputs" with precise measurement of confidence in a failure intensity. The required measurements are apparently not infeasible, even in the ultrareliable region.

Formal methods have a new role to play in establishing a basis for measuring dependability of the programs they are used to develop.

Sensitivity can sometimes be used to assess dependability, widening the applicability of Blum's idea. Mathematical properties of a specification other than its input decomposition theory can be used to apply self-checking.

Finally, reliability or other testing measures can be obtained without the use of an oracle, if self-checking software is first assessed with a uniform-profile random test.

References

[BF91] R. W. Butler and G. B. Finelli. The infeasibility of experimental quantification of life-critical software reliability. In *Software for Critical Systems*, pages 66–76, New Orleans, LA, 1991.

[BK89] M. Blum and S. Kannan. Designing programs that check their work. In *21st ACM Symposium of Theory of Computing*, pages 86–96, 1989.

[BS87] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. on Soft. Eng.*, 13:1278–1296, 1987.

[BW94] M. Blum and H. Wasserman. Program result-checking: A theory of testing meets a test of theory. In *35th Annual Symposium on Foundations of Computer Science*, pages 382–391, Santa Fe, NM, 1994.

[Ham94] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, 1994.

[HV93] D. Hamlet and J. Voas. Faults on its sleeve: amplifying software reliability. In *International Symposium on Software Testing and Analysis*, pages 89–98, Boston, MA, 1993.

[Knu81] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1981.

[PvSK90] D. L. Parnas, A. van Schouwen, and S. Kwan. Evaluation of safety-critical software. *Comm. of the ACM*, 33:638–648, 1990.

[Sho83] M. L. Shooman. *Software Engineering Design, Reliability, and Management.* McGraw-Hill, New York, NY, 1983.

[TLN78] R. Thayer, M. Lipow, and E. Nelson. *Software Reliability.* North-Holland, New York, NY, 1978.

[VM92] J. M. Voas and K. W. Miller. Improving the software development process using testability research. In *Third International Symposium on Software Reliability Engineering*, pages 114–121, Research Triangle Park, NC, 1992.

[Voa91] J. Voas. Preliminary observations on program testability. In *Pacific Northwest Software Quality Conference*, pages 235–247, Portland, OR, 1991.

[Voa92] J. M. Voas. Pie: A dynamic failure-based technique. *IEEE Trans. on Soft. Eng.*, 18:717–727, 1992.