Research Issues in Testing for Software Systems Assurance

white paper

Dick Hamlet
Portland State University
Center for Software Quality Research
Box 751
Portland, OR 97207
(503) 725-3216
hamlet@cs.pdx.edu

Abstract

Software will be improved by a broad-spectrum attack on the intrinsic complexity of the problems software tries to solve. However, most of the methods that have been (and will be) proposed for improving the development process are not subject to practical validation. The arguments for using these methods may be plausible, but their experimental validation is elusive. Software testing, on the other hand, has the potential to serve as a direct evaluation of software quality. Testing's role in high-integrity software is not to *create* quality — it cannot be "tested in" — but rather to *measure* it after the fact. Testing can therefore be the touchstone that applies to any and all methods designed to engender quality. Furthermore, testing can be applied to software whose development history is obscure, "off the shelf" components whose developers are unwilling or unable to account for the process they used.

Unfortunately, the theory of software testing is not adequate for the role of arbiter of software quality. Any useful theory must be statistical in nature, but existing software reliability theory is not generally accepted. This white paper outlines the deficiencies of reliability theory, and proposes instead a theory of software "dependability" that addresses those deficiencies. Basic research in dependability testing is essential, because without a plausible, widely accepted theory, all software evaluation is subjective.

Software is a unique human artifact because it is not constrained by wear, harsh environments, etc., as are physical systems. Software defects are designed in, and they answer to no higher law than human engineering. This situation is the source of difficulties in software theory, but it also holds immense promise: software *can* be the best thing people make. We just have to understand it.

Keywords: software testing theory, reliability, testability, dependability

1. The Problem: Measuring Software Quality

When a new software-development method is proposed, its evaluation is usually imagined to involve an experiment in which the method is used in one development, and not used by a control group, then the results compared. It being acknowledged that this ideal experiment is too expensive in practice, a case study may be subjectively evaluated by those who used the method. But all such studies are very expensive and hard to control, and they are easily invalidated by the special nature of particular projects, particular organizations, etc. If anyone chooses to disagree, it is easy to find factors that were not controlled, factors that invalidate the results. In short, it will never prove possible to "evaluate" changes in a complex development process with an acceptable degree of objectivity. The argument for any method will continue to be no more than that the method is plausible, that its users felt it was helpful, and in the end, that we wish and hope that it works.

If there is to be scientific progress in software development, it must rest on measurements of the software product, not on the process by which the product was made. The measurements must be accepted, and development innovations must be shown to lead to improvements in the measurements. This demonstration will of necessity be mostly theoretical. Those who propose methods must argue successfully that their method *should* lead to improvements *in the measurement.* These arguments may of course be flawed, but they are the first step in justifying an expensive and hard to control experiment or case study.

For the past 15 years, testing has been thought of as an activity for exposing software problems. A good test has been seen as one in which the software fails so that a defect causing the failure can be corrected. But because no testing method (or combination of methods) is effective at exposing all possible failures, a very unsatisfactory situation arises. Eventually, all the defects a testing method can find will have been found, so the method's usefulness (for exposing failure) is at an end. But the "tested" software is still of unknown quality. Some defects have been removed; but what defects remain? The analogy to fishing a lake is apt: when you catch no fish, it doesn't necessarily mean there are none in the lake. A deep lake 100 miles in diameter (roughly the size of one of the Great Lakes) might contain 10^{11} m^3 of water, and trolling it for a day, a fisherman might probe about 10^4 m^3 (assuming the lure is attractive in a 0.2 m^2 crossection), or about fraction 10^{-7} of the space. A program with a pair of integer inputs running on a 32-bit machine has about 2^{64} possible input values, and testing it for a day at one test/sec is about 3×10^4 tests, or fraction 10^{-15} of the possibilities. It is as ideal to imagine that each instant of fishing time covers a different part of the lake, as to suppose that every test will be truly different. Just as fish do not always bite when the bait comes near, so bugs do not always reveal themselves when tests encounter faulty code. All in all, it seems that a fisherman predicting no fish in Lake Michigan after getting skunked is far more likely to be right than a software tester predicting no bugs in a trivial program that tested without failure.

Despite the difficulties of "fishing" for software failures, a statistical testing theory is the only candidate for plausibly measuring software quality. Testing samples software behavior; an acceptable theory must calculate the statistical confidence in quality to be expected from a test that does not fail.

2. Statistical Testing — Theory and Practice

In software reliability theory [Thayer+78, Hamlet94], random tests are treated as samples from the program's behavior space, with inferences (and associated confidence bounds) from the sample. Reliability theory is attractive because probabilistic methods are an appropriate way to circumvent the basic unsolvable problems that underlie program behavior. However, existing software reliability theory is unacceptable [Hamlet92], for four reasons:

(1) *Systematic (as opposed to random) nature of software defects*. Reliability engineering has successfully dealt with chaotic processes like mechanical wear, and random defects introduced by imperfectly controlled manufacturing. But software faults are in principle different. Software failures are perfectly deterministic, perfectly repeatable, properties of a unique object. To attribute to programs statistical parameters like mean time to failure seems in principle wrong.

(2) *Problem of the operational profile*. To accurately sample any phenomenon, the samples must be drawn so that they accurately represent the population, or predictions about that population are meaningless. For program testing, the population is described by an "operational profile" giving the probability that a given input will occur in use. Such profiles are very difficult to obtain, even when the input space is represented by only a coarse functional division. For nonnumeric input spaces, it may be difficult to make random input selections. Worst of all, using the wrong profile for failure-free testing always *over*estimates the software's quality. In the analogy to fishing, nothing will be learned by fishing in the wrong place without luck.

(3) *Test oracle problem*. The essence of reliability is quantification of success. But for software, *recognizing* success is a non-trivial problem. A *test oracle* decides whether or not a particular program result is correct according to the specification. Specifications often exist only in natural language, and require human interpretation to decide if a result is correct, so the oracle is not automatic. The problem is compounded by random selection of inputs: there will be no "easy" cases where the answers are obvious. (Fishermen do not seem to have any analogous problem!)

(4) *Impracticality of reliability testing*. Careful analysis [Butler&Finelli91, Littlewood&Strigini93] shows that demonstrations of software reliability by statistical testing, even if valid in principle, are too time-consuming to be carried out in practice, even with vastly improved methods and very powerful hardware. There are simply too many possibilities (too big a "lake") for a significant sample to be taken (not enough time to "fish"). Difficulties (2) and (3) also contribute to impracticality, in a less fundamental way.

Each of these four difficulties should be addressed by research:

(1´) *Suggesting experiments*. Ultimately, the objection that software is not stochastic can be answered only by experimental work showing that real software appears to approximate a stochastic system, for all that it is not probabilistic in principle. However, an underlying theory must drive experimentation, suggesting revealing experiments to confirm or refute its assumptions and results.

(2´) *Coverage testing*. Unit testing methods rely on "coverage" to uncover failures. (For example, the most obvious technique insists that test data force the execution of every program statement.) A basis for the powerful (but technically fallacious) belief that coverage testing without failure ensures reliability has never been established [Hamlet&Taylor90], but such a basis would validate much of practical testing today. A precise relationship between coverage and reliability would be a "profile-independent" theory, which would remove a large practical obstacle to testing for reliability. Testing of reusable components, and certifying off-the-shelf software, require a theory that is independent of usage, and these two applications are of increasing importance.

(3´) *Mechanical oracles*. Two general solutions to the oracle problem have been proposed. In "dual programming" [Panzl78], at least two versions of a program are independently developed from the same specification. Comparing their outputs gives a "semi-oracle" — when results disagree at least one of the programs is incorrect, but agreement may be misleading because of a "common-mode failure" of the versions [Knight&Leveson86]. Other semi-oracles can be realized by adding redundant information to the code, for example, comments that describe the program "mode" [Howden89]. A better solution requires a specification formal enough to mechanically implement a check on program results, which constitutes a true oracle [Antoy&Hamlet92].

(4´) *Probable correctness*. Only by reexamining the sampling problem from a new perspective, can the intractability of the testing problem be overcome. What is needed is a notion of "probable correctness," essentially an estimate of our confidence in the success of tests [Hamlet87]. In the "testability" approach of Jeff Voas, information about potential program faults is combined with random testing to reduce the problem size [Voas&Miller92, Hamlet&Voas93]. (In the analogy, Voas finds the "holes" which regular fishing might miss, where special effort is called for; by identifying the disproportionate effort there, the overall effort is vastly reduced.) In a second new viewpoint, Manuel Blum suggests that program users really want to know if a program can be trusted only for one particular input. The program can perform a small number of extra calculations to check itself, without an oracle. These calculations are arranged to be very unlikely to agree unless the original result is correct [Blum&Kannan89]. (In the analogy, it is undesirable to catch a "fish," that is, to experience a program failure. Blum can show that "fishing right here" is safe, without caring about the rest of the lake.)

The theory of software reliability that will emerge from these investigations is not yet determined. But it is clear that the statistical approach is promising, and that although the problems are formidable, they can be systematically attacked. The theoretical difficulties in understanding software arise from its unique properties as a "non-physical" engineering artifact. Software does not wear out, does not fail under temperature extremes, does not have flaws in its materials, etc. All of its defects are literally designed in by human beings. Software therefore presents a unique opportunity as well as a difficult challenge. It *can* be constructed with a perfection impossible to attain in other artifacts. In practice, this potential for high quality has too often been traded for low cost. It is all too possible to get a great deal of functionality cheaply using unreliable software. The necessary research seeks the understanding needed to go the other way: to guarantee high quality when it is required.

References

[Antoy&Hamlet92]

S. Antoy and D. Hamlet, Self-checking against formal specifications, *Proc. Int. Conf. on Computing and Information*, Toronto, May, 1992, 355-360.

[Blum&Kannan89]

M. Blum and S. Kannan, Designing programs that check their work, *Proc. 21st ACM Symposium on Theory of Computing*, 1989, 86-96.

[Butler&Finelli91]

R. Butler and G. Finelli, The infeasibility of experimental quantification of life-critical software reliability, *Proc. Software for Critical Systems*, New Orleans, LA, December, 1991, 66-76.

[Hamlet87]

R. Hamlet, Probable correctness theory, *Inf. Proc. Let.* 25 (April, 1987), 17-25.

[Hamlet92]

D. Hamlet, Are we testing for true reliability?, *IEEE Software*

[Hamlet94]

D. Hamlet, Random testing, in *Encyclopedia of Software Engineering,* J. Marciniak, ed., Wiley, 1994, 970-978.

[Hamlet&Taylor90]

D. Hamlet and R. Taylor, Partition testing does not inspire confidence, *IEEE Trans. Software Eng.* SE-16 (December, 1990), 1402-1411.

[Hamlet&Voas93]

D. Hamlet and J. Voas, Faults on its sleeve: amplifying software reliability testing, *Proc. ISSTA '93,* Boston, June, 1993, 89-98.

[Howden89]

W. E. Howden, Validating programs without specifications, *Proc. TAV-3,* Key West, December, 1989, 2-9.

[Knight&Leveson86]

J. Knight and N. Leveson, An experimental evaluation of the assumption of independence in multiversion programming, *IEEE Trans. Software Eng.* SE-12 (Jan., 1986), 96-109.

[Littlewood&Strigini93]

B. Littlewood and L. Strigini, Validation of ultrahigh dependability for software-based systems, *CACM* 36 (Nov., 1993), 69-80.

[Panzl78]

D. J. Panzl, Automatic software test drivers, *IEEE Computer* 11 (April, 1978), 44-50.

[Thayer+78]

R. Thayer, M. Lipow, and E. Nelson, *Software Reliability,* North-Holland, 1978.

[Voas&Miller92]

J. M. Voas and K. W. Miller, Improving the software development process using testability research, *Proc. Third International Symposium on Software Reliability Engineering,* Research Triangle Park, NC, October, 1992, 114-121.