Are We Testing for True Reliability?

DICK HAMLET, *Portland State University*

◆ *Conventional reliability theory works fine — if its assumptions hold. But for software, they fail. A new theory of trustworthiness is needed.*

Software engineers are less eager to accept reliability modeling than engineers in other disciplines. Instead, they propose clever methods for developing "defect-free" programs or for testing to eliminate all defects. These methods, although valuable and necessary, are essentially unrelated to reliability.

Reliability is the statistical study of failures, which occur because of some defect in the program. The failure is evident, but you don't know what mistake is responsible or what you can do to make the failure disappear. Reliabililty models are supposed to tell you what confidence you can have in the program's correctness.

But conventional reliability theory — which is taken from the reliability engineering of physical objects — is not satisfactory. It works only when a similar set of operational assumptions hold. It is not tailored for software's quirks. Thus, it rarely provides developers with confidence that they can rely on their software.

TWO KINDS OF MODELS

To understand what a reliability model demands, you must first understand that there are two kinds. *Reliability-growth* models are applied during debugging. They model repeated testing, failure, and correction. Managers can use them to predict when the mean time to failure is large enough to release the software.

Reliability models, in contrast, are applied after debugging, when the program has been tested, and no failures have been observed. The reliability model predicts the MTTF you can expect.

At first glance, the reliability model appears to be just a limiting case of the reliability-growth model in which zero failures are observed. However, these models

differ qualitatively. Because reliability-growth models are used during debugging, as you observe failures, the observations give you direct, nonstatistical feedback on the model's performance: The MTTF you calculate from the model should be roughly the MTTF you actually observe, which for most of the debugging period will be short, say 100 runs. Furthermore, since the reliability-growth model is designed to tell simply when the software's operational quality is at an acceptable level, prediction accuracy can be less than perfect.

The predictions of a reliability model, on the other hand, are purely statistical. Because you do not observe failures, there is nothing to check the predicted MTTF against. Moreover, the MTTF is much larger than any observation data you can obtain, say 100,000 runs. For safety-critical software, the required MTTF may be orders of magnitude higher. For these systems, the calculated MTTF must be precise, because it fulfills a contractual obligation or is related to an inflexible requirement like protecting human lives.

Thus, much more is demanded of a reliability model than a reliability-growth model. And for that reason, you should be careful in accepting the results of a reliability model at face value. These are the models I am concerned with in this article. I want to show that if testing has shown zero failures for enough samples that a model predicts an acceptable MTTF, you can't always trust that model.

WHY CONVENTIONAL RELIABILITY THEORY FAILS

Software reliability is usually modeled by analogy to physical reliability — the engineering of reliability for objects. Physical reliability is concerned with collections (often of apparently identical mechanical parts), whose members differ slightly because of random (as opposed to systematic) fluctuations — such things as manufacturing, operating environment, and durability. The collection is tested, and the destruction times of members are noted; physical reliability theory is concerned with calculating the MTTF.

In the software analogy, a single program is given different inputs to form the collection. A failure rate θ is postulated for each program P as the probability that P will fail on an arbitrary input. Inputs are assumed to be drawn from an operational profile that may weight some more heavily than others, reflecting the actual usage expected of P. Then the statistical theory relates θ to the number of tests N conducted without failure. The MTTF is $1/\theta$, as derived in Martin Shooman's text.[1] If the theory holds, the calculated MTTF will predict observed behavior when inputs come from the operational profile.

TWO DEVELOPMENT MYTHS: DEFECT-FREE SOFTWARE AND TESTING AWAY FAILURES

Some developers seek to avoid testing altogether, claiming that they can develop defect-free software using formal methods. The argument is that software fails only because discrete mistakes were made in developing it. That is, each application has a perfect program — a program that cannot fail. These developers believe that they can create defect-free software.

Other developers acknowledge that perfect programs cannot always be created, but they still hope to remove all defects through testing.

Both these approaches have serious flaws.

Defect-free software. On the surface, the idea of defect-free software seems logical; there is no physical medium in software, as there is in other types of development, to wear out, become flawed during manufacturing, and so on. Of course, the computer executing a program is a physical object, and a physical medium transmits the program text, but these factors are trivial sources of failure compared with program-design mistakes.

Developers, therefore, are tempted to attack failure at its source. But attempts to construct perfect software are doomed to fail for several reasons:

♦ *Intuitive problems.* The problems the software is supposed to solve are imperfectly understood by those who need the solutions. Users communicate this imperfect understanding, imperfectly, to professionals who devise the solutions, who in turn imperfectly communicate the solution's characteristics.

Inevitably, the wrong problem is solved — a difficulty not addressed by improving the development method, particularly by using formal methods, which users do not see or understand.

♦ *Complex problems.* The solution to a complex problem isn't always simple, and the problem may be arbitrarily complex. Computer systems are designed to be general purpose, so more and more complex problem solutions will always be attempted, eventually outstripping the control of any method.

The Strategic Defense Initiative shows that some developers will always be willing to attempt problems acknowledged to be beyond the state of the art.

♦ *Slipshod maintenance.* Once a computer system exists, there is an irresistible pressure to modify it, and unhappily to do so with less care than was used to create it. The fragile nature of languages and of digital computation itself feed the maintenance problem. Programs and their behavior lack any kind of continuity. A very small change to a program can have a very large effect on its behavior; behavior can be arbitrarily different on input that is arbitrarily similar. There are programs for every problem without this discontinuous behavior, but no one knows how to stick to that set of programs, particularly during modification.

But that these problems make defect-free software impossible is no reason to abandon attempts to improve software development or use formal methods. On the contrary, the future will see more formal methods because they are cost-effective. However, more formal development can-

The chance that a single test will fail is θ, or $1 - \theta$ that it will succeed. Thus the probability that N independent tests will all succeed is $(1 - \theta)^N$. The largest value of θ such that $(1-\theta)^N > \alpha$ defines the $1 - \alpha$ upper confidence bound on θ — that is, the probability that this value exceeds the correct value for P.

Solving for θ, testing with N independent points from the operational profile gives $1 - \alpha$ confidence that the failure rate is below θ, if $\theta = 1 - \alpha^{1/N}$. For example, one million test points gives 99 percent confidence that the failure rate is below 4.6×10^{-6}, an MTTF of about 220,000 runs.

Unfortunately, this conventional reliability theory is flawed in a number of ways.

Random variables differ. Some software, like a telephone switch, is intended to operate continually, so starting it and waiting for failure is analogous to starting a destructive test of a physical object. The random variable is execution time, so you can form a collection of objects using a single program by giving it different inputs, weighted according to its operational profile.

However, most programs operate either in a batch mode, in which they are given input then compute and terminate, or in an interactive mode, in which they may be in operation for a long time but spend most of it awaiting human input.

For batch and interactive programs, run count replaces time as the random variable. That is, the MTTF is measured in executions, or number of tests tried. For a batch program, a run is a complete execution; for an interactive program, it is a single I/O interchange.

Program runs aren't always independent. To satisfy the analogy to mechanical parts, program runs must be independent of one another. Otherwise, in the mechanical analogy, the parts being tested would influence one another (the failure of one would depend on the others), fundamentally contravening the statistical theory.

When programs accumulate state information over their runs, they are explicitly compromising run independence. For example, an interactive program can, as a side-effect of responding to an ill-formed command, go into a state in which most subsequent commands fail. Those subsequent runs are statistically meaningless. Dave Parnas cleverly notes that it would be a good idea for safety-critical programs to reinitialize themselves whenever possible to improve the accuracy of reliability theory by avoiding state build-up.[2]

Physical systems fail because defects appear during their use. Design defects are not unknown in established fields such as civil engineering. For example, in the walkway collapse at the Kansas City

not guarantee reliability, which must be independently assessed at the end of any human activity, to catch inevitable mistakes.

Testing away failures. Again, on the surface, testing away all defects seems possible. Each program's code is finite and hence must have only a fixed, finite number of defects. But even clever testing methods are doomed to fail.

Practical systematic testing is a game of coverage. The tester tries to make sure that all the elements of the program or specification have been tried. In partition testing, you divide, or partition, the input space into classes that characterize the elements to be covered and create test sets by selecting points from each class.

The hope is that the input classes will be homogeneous — any point selected from each class will be representative — and that cleverly defined classes will uncover all failures. Partitioning may be based on program structure, as in clear-box methods (named for electronic components in boxes that allow the interior to be examined), or on the specification, as in black-box methods, which do not use the code.

In structural testing, program elements are exercised. Certainly if some part has not been tried at all, you can have no confidence in its quality. The best known structural test method is statement testing, in which tests force the execution of each statement. Statement testing is part of a control-flow hierarchy of methods, including branch testing (tests must force each branch to be taken both ways), a variety of dataflow methods (tests must cover paths defined by certain definition/use relationships among variables), and full-path testing (tests must force execution of all paths). Path testing has many variants, in which loops need not be executed for each of the infinite ways generally possible. These variants are usually viewed as the most stringent of the practical structural methods.

All the finite, path-based, structural methods are straightforward and relatively inexpensive to implement, and a variety of research and commercial tools have been created for them.

In contrast to control-flow methods, mutation testing is probably the least known structural method. Small individual changes are introduced in the program under test (the most interesting ones occur in expressions), and test sets must distinguish the mutant program from the original. Mutation testing is often surprisingly difficult because it is hard to think through the extensive bookkeeping required.

The only implementation of mutation testing is the brute-force creation and execution of a vast collection of mutants, which is expensive and slow. So-called "weak mutation" — in which the mutation's effect is detected in the state immediately following — is better in this regard, but still not accepted.

In black-box testing, the specification supplies the organizing information for systematic testing. What is often called functional testing isolates a collection of actions (functions) a program should per-

Hyatt-Regency hotel, construction could not handle the weight of a crowd, and the structure simply waited for its first crowd, then collapsed. However, reliability theory is not applied to such mechanical situations — the objects are one-of-a-kind. Only computer programs have the wealth of design defects that might support a statistical approach.

For a statistical theory, variations must be independent. When one test system fails, it must not imply anything about the failure of others. Programs fail only because of design defects lying in wait for the input that excites them. Test systems differ only in the input each is given. Hence, these inputs must not be correlated. That is, the failure or success of one input cannot force the same behavior for another. Correlated inputs are analogous to cheating on the testing effort by copying one data point several times. If that point is a failure, the reliability will appear worse than it really is; if not, the reliability will appear to be better. In program-release testing, the result is always overly optimistic. Because a release test by definition does not fail, if it contains correlated points, the calculated reliability is too high.

There may be no operational profile. For a telephone switch, you can determine the operational profile empirically from past data, or model the load analytically and derive data from the model. Such a distribution is accurate and appropriate. However, most programs' profiles are not known. What is more in doubt is the validity of any operational profile for a piece of system software. The software may handle an immense number of possibilities, whose frequency and sequence depend on human vagaries. Each execution is more a unique special case than a sample drawn from some distribution. Again, an inappropriate distribution can only work against calculated reliability, because nothing is gained by testing in regions weighted too heavily, but improperly neglected regions give a false security that the software does not in fact support. Thus conventional theory will give overly optimistic results.

There is no appropriate failure rate. A common assumption in software development is that the instantaneous rate of failure — the hazard rate — is constant. For programs that *are* more likely to fail the longer they run, a model might use a time- or run-increasing hazard rate, as in mechanical systems that wear out. (The analogy would be not that the program wears out, but that it accumulates state that increases its likelihood of improper behavior.) Such a model, however extreme for software, gives nearly the same results as for a constant hazard.

form, and requires test data to exercise each function. You can further refine the test sets by including parameters that modify functional behavior, or by requiring that sequences of functions be tested.

Functional testing directly tries what is expected of the software. You can plan tests as soon as you have the specification, which is a plus. Functional testing also protects developers against the embarrassment of a program that doesn't work at all because of an oversight or misunderstanding about some feature.

Why partition testing is misleading. Partition testing has great intuitive appeal, but it is not a panacea. As an absolute method, it is flawed because partition tests can be misleading. For most programs and most testing methods, an infinite number of test sets will satisfy the method. The tester therefore will find it difficult to apply partition testing systematically.

For example, suppose test set T_M is selected, it satisfies a systematic method, and for T_M the tested program does not fail. However, if among the infinite number of other test sets that also satisfy that method, some (or perhaps many) would cause the program to fail, then T_M is misleading.

Unfortunately, misleading test sets are the rule rather than the exception. The following examples for structural and functional methods are common.

♦ *Structural.* A program has several nested conditionals, such that to reach one of its statements S_e, the input must be a singular matrix of rank greater than three, in which one row has elements in strictly increasing order. If S_e contains a complex formula involving the matrix elements, the tests devised for (say) branch testing are not likely to reach it with nontrivial matrix elements — it is too difficult to find data just to reach S_e. Hence if S_e's formula is correct for trivial elements only, most branch-adequate test sets will be misleading.

♦ *Functional.* A program can process read and write commands on random-access files named as a command parameter. A functional test might cover each command, some parameter possibilities for each, and some command sequences. But how likely is the test set to include the sequence write-write, in which two writes address the same record? When this case fails, the program can be thrown into a state in which everything goes wrong. In fact, an acceptance test of an early Digital Equipment Corp. PDP-10 operating system did include such a write-write sequence because record numbers were generated at random. The test failed but was ignored. Testers discovered later that by exciting this bug, a user program could gain unrestricted access to any disk block by absolute address! But no "reasonable" tester would select a set with this sequence, so functional tests, of this system at least, will almost always be misleading.

False comparisons. Misleading test sets confound most of the ways that have been devised to compare the effectiveness of testing methods. For example, the most common comparison uses an inclusion relation: If a test set for method 1 necessarily satisfies method 2, then 1 includes (or subsumes) 2. In the

Furthermore, a growing hazard rate does not really capture failure arising from state accumulation. First, state is not always harmful — it can be used to catch and even correct program problems, and a single program can alternately exhibit beneficial and harmful aspects of state accumulation as it runs. Second, tinkering with the hazard rate does not address the issues of sample independence, nor does it alter the qualitative result that reliability does not depend on program size.

Defects per line should be roughly constant. Conventional theory fails to explain a primary observation about software systems: the MTTF (in runs) in large systems is roughly inversely proportional to program size. That is, the number of defects per line is roughly constant. This software "law," which is observed in practice, is plausible when defects are human mistakes that arise because of the complexity of large programs. But conventional theory predicts that the MTTF does not depend on program size.

TOWARD TRUE RELIABILITY

Conventional reliability is deficient in two significant ways: It relies on an operational distribution that may not exist, and its assumptions about sample independence do not hold. To correct these deficiencies and find a better theory, we must probe the correct sample space and examine the chance of program failure under arbitrary circumstances. Unless we can find a true reliability theory — a theory of what Parnas calls trustworthiness, unlikely to fail catastrophically[2] — developers are building on sand. When they make generalizations like "random system testing should replace unit partition testing," or "inspections are better than testing," they may be setting themselves up for disaster.

Sampling basis. For any reliability theory, you must have a proper sampling basis to infer the probability of failure. Program input is not an appropriate sampling choice because the statistical procedures work only when samples directly probe the sources of failure. For programs, random testing over the input space is only tenuously connected to the design flaws that reside in the code. Input-space sampling also fails to predict the direct relationship between defects and program size because program characteristics are invisible.

The programming analogy to mechanical reliability would be better if the sample space were closer to the source of defects, the program text. If inspections were perfect at detecting failure, it would make more sense to inspect a sample of code, and infer the quality of uninspected code, than to sample executions and infer the quality on unexecuted inputs. Inspec-

best-known inclusion relation among methods, branch testing strictly subsumes statement testing. Intuitively, when method 1 subsumes 2, there is no point in using 2, because method 1 is always at least as good and may be better.

Misleading test sets flaw this inclusion relation, however. The "better" method's test set can always be misleading, while the "worse" method's (different) test set is not. Furthermore, these test sets can be natural for the methods, so the worse method's test set may actually be the right choice for practical testing. To illustrate, consider the Pascal procedure:

```
function misled(x: real): real;
  begin
    misled := x;
    if x > 0 then
      misled := 1.0/sqr(x)
  end
```

Suppose you are trying to compute the reciprocal of the square root function, but if the input is not positive, you wish to return it unchanged. Under the inclusion relation, branch testing is the better method, and statement testing, the worse. A natural statement test set for an electrical engineer would be something like {2}, for which the expected result is 0.707.... Any such test will uncover a failure that results from mistaking the square function "sqr" for the root function "sqrt."

However, if you are trying to attain branch coverage, you are in danger of trying a misleading test set like {–1,1}. The emphasis on branches focuses attention on the predicate rather than on the computation. Thus, the better method isn't better at all. The more elaborate the demands of a systematic testing method, the greater the danger of trivializing the test set to satisfy it, and hence the more likely that it will mislead.

Obviously then, misleading test sets are the bane of any testing method. Bill Howden attempted to capture their absence by calling a testing method reliable (*not* in the statistical sense of the word) for a program if there are no misleading test sets for that program that satisfy the method.1 Unfortunately, methods are seldom reliable, and reliability is not in general a property of any algorithmic testing method. Furthermore, the restrictions you must place on programs and methods to guarantee reliability are too stringent to make them practical.[2]

Why partition testing is unreliable. Ross Taylor and I, repeating the 1984 experiments of Joe Duran and Simeon Ntafos, found that random testing and partition testing are more similar than you might think.[3]

For that reason, you can express an estimate of the quality that can be tested into software with partition testing as an MTTF. A typical unit partition test might contain 100 points. If all succeed, and you assume the test is random, there is 80 percent confidence in an MTTF of about 62 executions. This very modest estimate hardly justifies the usual claim that software has been tested and works.

REFERENCES

1. W. Howden, "Reliability of the Path-Analysis Testing Strategy," *IEEE Trans. Software Eng.*, Sept. 1976, pp. 208-215.
2. R. Hamlet, "Reliability Theory of Program Testing," *Acta Informatica*, 1981, pp. 31-43.
3. D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Trans. Software Eng.*, Dec. 1990, pp. 1402-1411.

tions are hard to quantify and control, however, so a different kind of testing is needed, one that samples the space of actual defects.

Sampling the state space. Both Parnas and I have proposed using the program's state space as the sampling domain for trustworthiness,[2,3] but this is not a good practical approach. The argument for using it is that failure results from textual flaws, or faults, when particular circumstances arise at the control point that contains the fault — which describes the state space exactly. It is a tuple of internal-variable values and a value of the location counter. The sampling distribution should be uniform because failure is no more likely for one state location than another. Such a theory correctly predicts that failure probability is proportional to program size.[3]

Unfortunately, the state space of most programs is far larger than the input domain, so a theory that requires sampling this much space is not attractive.

Larry Morell and Jeff Voas suggest an alternative.[4] They argue that points of the state space are themselves correlated. Starting with an input, each program normally goes through a sequence of states (its computation) to reach an output. The computation steps are defined by the operational semantics of the program statements as mappings from state to state.

Hence, when some input leads to program failure, the entire computation has "failed," so sampling states from that sequence overemphasizes failure. States from a single correct sequence are correlated in a similar way. To add to the confusion, the same state may appear in both failed and correct computations.

You don't have to sample variable values in a state if the final result does not depend on them. Although the problem to determine such dependencies is generally unsolvable, dataflow techniques give a good approximation. Thus, I assume that a computation's data states contain only variable values on which the result depends. Each computation starts with an input, and may have an initial "correct" state subsequence, then the first bad state occurs, followed by a "failed" portion of the sequence. Thus, sampling the input space doesn't work. Inputs that are apparently independent can lead to the same state and thus are not independent samples, relative to failure in that state.

Data fan-in/-out. The crux of appropriate state-space sampling is the idea of state collapse — when different computations contain common sequences of states. State collapse can occur through fan-in — when two paths in a program join or when the values taken on by internal variables are restricted.[5] An example is an assignment statement with a constant right side. This statement produces a kind of ultimate fan-in because after its execution, the assigned variable has lost whatever range of values it might have had; it now can have only a single value.

Capturing the development process lets us expose likely sources of defects and test for them with appropriate partitions.

Fan-out occurs when the possible values in states expand. For example, an input statement is the ultimate fan-out: whatever restricted values the input variable might have had before the statement, after it, any value is possible.

Many program statements do not fan in or out. For example, an assignment statement using an arithmetic operator like + has as many state value possibilities after execution as it did before.

When programs have a good deal of fan-in, their possible data states collapse to a set that can be smaller than the input domain. Many inputs lead to exactly the same computation, and intuitively it is the computations that should be sampled in testing. When programs have a good deal of fan-out, there is a combinatorial explosion of the state space because coverage of early states does not imply coverage of later states with a wider value range. When programs neither fan out or in, each input leads to a different computation, and appropriate state-space sampling is the same as input sampling. However, the appropriate distribution is uniform, not the operational profile.

Thus, in some cases, far fewer test points are needed to establish trustworthiness than are needed to satisfy conventional reliability. In other cases, the two require about the same number of points, but with different test distributions; And in still other cases, trustworthiness can require vastly more points than reliability, with the full exploration of each state as an upper bound.

Interestingly, computation diversity and fan-out occur when programs read input throughout a computation rather than just at the beginning. Intuitively, such programs are interactive, and make essential use of saved state. The ultimate pathological case is that of real-time programs, in which inputs and state expansion appear at any point in the computation because an interrupt occurs.

HOW SHOULD WE TEST?

Although testing certainly has its limitations, it is unwise to discard it as a useful part of software development. The development process is beginning to be studied and controlled. Capturing the process provides the opportunity to expose likely sources of defects, and to test for them with appropriate partitions. Conventional random testing also has its merits.

Partition testing. The box on pp. 22-25 describes the weaknesses of partition testing to dispel the idea that a successful test means the software is reliable. But I would never recommend that you abandon partition testing altogether — and particularly not in favor of system-reliability testing.

Partition testing is the developer's best tool to probe the software for specific defects. Of particular importance are defects that lead to failures with catastrophic consequences. However infrequent a catastrophic failure may be, it is worth expending effort to preclude it, and a partition devised by considering the failure possibilities (for example, using a safety fault tree[6]) is just the way to attack the problem. Indeed, you should use partition testing whenever you suspect a particular source of defects, with a partition emphasizing the defect-prone input. Testing in that partition gives little confidence in overall reliability, but it is the only means of gaining confidence that the particular problem will not arise.

For example, a module that undergoes a specification change late in development, or one that fails an inspection, is an obvious potential defect source. Not only should you heavily exercise its structural and functional unit-test partitions, but when it is integrated into a system, the whole should be tested with a partition that singles out module execution.

Partition testing has many advantages. A functional partition test can be designed beginning in the requirements stage of development. Test data for structural partitions can be automatically generated, and even if hand generated, the tester has a systematic goal and automatic support.

If you have a complex piece of software, whose usage patterns you do not know, and you have a vast nonnumeric input domain, partition testing is probably your only choice.

Random testing. Preliminary results (based on a somewhat doubtful model that uses failures "tagged" by their origin[7]) indicate that uniform-distribution, state-space testing should be much better than partition testing at establishing confidence in apparently defect-free software.

However, random testing is not practical because it requires many orders of magnitude more test points than current practice. Even if you have an oracle — some means of mechanically deciding if program results are correct — random testing is barely feasible. Without an oracle, it is not feasible at all in most cases.

But random testing is the theoretical model that can answer fundamental questions that have too long been ignored. If, in fact, testing for a reasonable reliability is impractical by any means, then testing is merely a defect-detection method that may not stack up well against others like inspections, and we should be changing our quality-assurance methods accordingly. On the other hand, if we can find a small state space to sample, we can make even trustworthiness practical.

Measuring the state-space coverage a test gives is not impractical. Well-known program-instrumentation techniques can record test penetration and statistically analyze the results. If the state-space theory is correct, such measurements can pinpoint states that have been poorly tested, and leave the difficult problem of how to reach them to the tester.

Voas has given the problem a novel twist with practical promise. He directly probes the state space by perturbing a state, then monitoring whether the perturbation affects program results. If not, that state (or the corresponding statement of the program) is not very "sensitive," since even if the data state were incorrect (as arranged by the perturbation) the results are correct. The lesson for programmers is that faults in insensitive statements will be hard to detect by testing. Perhaps the cleverest part of Voas's idea is that he need not consider the input space, so he doesn't have to reach the data states he perturbs or consider the operational profile.

The main points I have tried to make in this article are

♦ In testing for true reliability, clever partition methods may be no better than random testing, and if they are not, then no practical testing technique exists for guaranteeing software quality.

♦ The analogy to mechanical reliability is a poor one for software.

♦ More research is needed on trustworthiness; it may be that the state explosion is not so important as it seems. Correlated states are grouped into program computations, which may be the appropriate entities to sample.

♦ It may be possible to statically characterize programs for which the combinatorics of testing is not forbidding, giving precision to the desirable quality of "testability." Testability is also dynamically described by Voas's idea of sensitivity. Experiments are needed to determine if these theoretically appealing ideas are in fact related to testing difficulty.

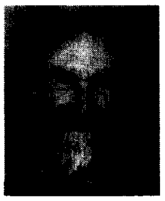
If testing for quality is the goal, then we must find a solution to the oracle problem. Until random tests of a million points become practical, testing is only a poor competitor for other heuristic defect-detection methods. ♦

ACKNOWLEDGMENTS

This work was supported by National Science Foundation grant CCR-9110111.

REFERENCES

1. M. Shooman, *Software Engineering Design, Reliability, and Management*, McGraw-Hill, New York, 1983.
2. D. Parnas, A. van Schouwen, and S. Kwan, "Evaluation of Safety-Critical Software," *Comm. ACM*, Sept. 1990, pp. 638-648.
3. R. Hamlet, "Probable Correctness Theory," *Information Processing Letters*, June 1987, pp. 17-25.
4. L. Morell and J. Voas, "Inadequacies of Date State Space Sampling as a Measure of Trustworthiness," Software Eng. Notes, Apr. 1991, pp. 73-74.
5. J. Voas, "Preliminary Observations on Program Testability," *Proc. Pacific Northwest Quality Conf.*, PNQC, Portland, 1991, pp. 235-247.
6. N. Leveson and P. Harvey, "Analyzing Software Safety," *IEEE Trans. Software Eng.*, Sept. 1983, pp. 569-579.
7. D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Trans. Software Eng.*, Dec. 1990, pp. 1402-1411.



Dick Hamlet is a professor of computer science at Portland State University, where he is investigating the theoretical foundations of testing. He is the author of two textbooks and about 40 refereed conference and journal articles.

Hamlet received a PhD in computer science from the University of Washington.

Address questions about this article to Hamlet at Portland State University, CS Dept., Center for Software Quality Research, PO Box 751, Portland, OR 97207; Internet hamlet@cs.pdx.edu.