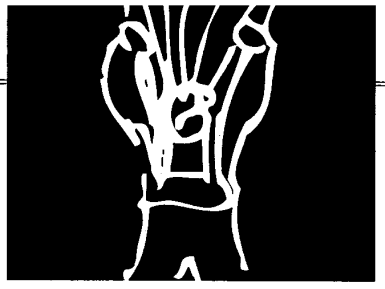


Software Testability: The New Verification

JEFFREY M. VOAS, *Reliable Software Technologies Corp.*
KEITH W. MILLER, *Sangamon State University*

◆ *Most verification is concerned with finding incorrect code. Instead, this view looks at the probability that the code will fail if it is faulty. The authors present the benefits of their approach, describe how to design for it, and show how to measure testability through sensitivity analysis.*

Software verification is often the last defense against disasters caused by faulty software development. When lives and fortunes depend on software, software quality and its verification demand increased attention. As software begins to replace human decisionmakers, a fundamental concern is whether a machine will be able to perform the tasks with the same level of precision as a skilled person. The reliability of an automated system must be high enough to avoid a catastrophe.

But how do you determine that critical automated systems are acceptably safe and reliable? In this article, we present a new view of verification and offer techniques that will help developers make this assessment. Our view, which we label *software testability*, looks at dynamic behavior, not just syntax. This differs from traditional verification and testability views, as the box on pp. 18-19 describes.

RELIABILITY PUZZLE

Every system has a true (or fixed) reliability that is generally unknown. Software testability, software testing, and formal verification are three pieces of the reliability puzzle, which developers must complete to get a picture of the software's true reliability. Each of the three puzzle pieces offers a unique bit of information about software quality. The goal is to combine all three. Testability analysis is related to but distinct from both software testing and

0740-7459/95/$04.00 © 1995 IEEE

VERIFICATION AND TESTABILITY VIEWS

Our views on both verification and testability differ from some of the more widely accepted views.

Verification. The *IEEE Standard Glossary of Software Engineering Terminology*[1] defines software verification as the "process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase."

Restated, software verification is the process that assesses the software's degree of acceptability, which is judged according to the specification. Software verification is broadly divided into two classes:

- *Dynamic software testing* is the process of executing the software repeatedly until a confidence is gained either that the software is correct and has no more defects, referred to as *probable correctness*,[2] or that the software has a high enough level of acceptability. Testing can be either white-box or black-box. White-box testing bases its selection of test cases on the code itself; black-box testing bases its selection on some description of the legal input domain. White-box testing gives you better coverage because it exercises larger regions, but white-box techniques are often helpless against classes of faults like missing code, which black-box testing can catch.
- *Formal verification* typically involves some level of static theorem-proving —the mathematical process of showing that the function computed by a program matches the function specified. No program executions occur in this process, and the result is a binary value: either the function computed by the program matches the specification or it does not. Problems arise in this rigorous process because of questions about program termination and the correctness of the rigorous process itself (who will prove the proof?). Furthermore, the process of completing such a proof can be more difficult than writing the program itself.

In this article, we describe a different type of verification that can complement both dynamic testing and static theorem-proving.

Testability. The *IEEE Standard Glossary of Software Engineering Terminology*[1] defines testability as

> *"(1) the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met,*

formal verification, which makes it a good complement to the other two pieces. Like software testing, testability analysis requires empirical work to create estimates. Unlike testing, however, testability analysis does not require an oracle — a program that performs the same functions as the software being developed. Thus, testing can reveal faults, while testability cannot, but testability can suggest places where faults can hide from testing, which testing cannot do. Testability complements formal verification by providing empirical evidence of behavior, which formal verification cannot do.

Testability information cannot replace testing and formal verification; but neither should developers rely exclusively on testing and formal verification. To be highly reliable, the software must have high testability, undergone enormous amounts of successful testing, and experienced formal verification. (In this article, we assume that "highly reliable" means 10^{-9} failures in a 10-hour period, although testing alone can never demonstrate this degree of precision.[1])

To illustrate how the three pieces fit together, consider a system that has 50 modules. Each module is tested with 100 random tests, and all modules pass the tests. In addition, the system passes 100 random tests. Ten of the modules, judged the most intricate and critical, are subjected to formal verification at various points in their development. Testability analysis reveals that five modules are highly insensitive to testing — testing is unlikely to find faults in these modules if faults exist. Only one of these five has been formally verified. At this point, verification resources should concentrate on the four modules that have low testability and have not been formally verified; they are the most vulnerable to hidden faults.

As another example, consider a system built entirely of formally verified modules. Using a development approach inspired by Cleanroom, the developers wait until after system integration to do random system testing. During this testing, some faults are discovered and the code is repaired. Regression testing and new random tests reveal no more failures, but testability analysis identifies several places in the code where testing is highly unlikely to reveal faults. These pieces of code are subjected to further formal analysis, and nonrandom tests are devised to exercise these sections more extensively.

HOW TESTABILITY WORKS

Our focus here is on one part of the puzzle, testability — how to design for it and how to measure the degree to which you have achieved it. To better illustrate what we mean by software testability, we offer two simple analogies.

The first shows how testability can enhance testing. Suppose software faults were gold. Software testing would be the actual mining process; software testability would be a geologist's survey before mining begins. The geologist does not actually dig for the gold, but

and (2) the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met."

According to this definition, to determine the degree you must have a test criteria. Consequently, testability is simply a measure of how hard it is to satisfy a particular testing goal, such as a coverage percentage or complete fault eradication. Testability requires an input distribution (commonly called a user profile), but this requirement is not unique to testability; any statistical prediction of semantic behavior during software operation must include an assumption about the distributions of inputs during operation.[2]

Our definition of software testability focuses on the probability that a piece of software will fail on its next execution during testing (with a particular assumed input distribution) if the software includes a fault. By contrast, the standard IEEE definition focuses on assessing if the I/O pairs are correct.

Computer science researchers have spent years developing software-reliability models to answer the question, What is the probability that this code is faulty? Software testability examines a different behavioral characteristic: *the likelihood that the code can fail if something in the code is incorrect.* It asks the question, What is the probability this code will fail if it is faulty?

Our testability differs from traditional views in another sense: In the past, software testability has been used informally to discuss the ease with which some input selection criteria can be satisfied during testing. For example, if a tester wanted full branch coverage during testing and found it difficult to select inputs that cover more than half the branches, the software would be classified as having poor testability.

In contrast, our testability is not concerned only with finding sets of inputs that satisfy coverage goals; it is trying to quantify the probability that a particular type of testing will cause existing faults to fail during testing. We focus our definition of testability on the semantics of the software, how it will ***behave*** when it contains a fault. This is different from asking whether it facilitates coverage or is correct.

REFERENCES

1. *IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Standard 610.12-1990, IEEE Press, New York, 1990.
2. R. Hamlet, "Probable Correctness Theory," *Information Processing Letters*, June 1987, pp. 17-25.

rather establishes the likelihood that digging at a particular spot would be rewarding. At one location, the geologist might say, "This valley may or may not have gold, but if it does, it will be in the top 50 feet and all over the valley." At another location, the geologist might say, "If you don't find gold in the first 10 feet on this plateau, there is no gold. However, on the next plateau you will have to dig 100 feet before you can be sure there is no gold." Thus, testability provides some guidance for testing, which is much better than testing blind. It can suggest the testing intensity (as the geologist suggested a digging depth) or estimate how difficult it will be to detect a fault at a particular location. If after testing to that degree of difficulty, you observe no failures, you can be reasonably sure (in an informal, not a statistical sense) that the program is correct.

The second analogy shows that testability can give you confidence of correctness in fewer tests (than you would conduct without it) *if* you are sure the software will not hide faults. Imagine you are writing a program to scan black-and-white satellite photos, and you are looking for evidence of a large barge. If you are sure that the barge will appear as a black rectangle, and that any barge will cover an image area of at least 10 by 20 pixels, you can have the program use techniques that it could not use if you had not established this barge size ahead of time.

For example, assume the original image has been subsampled so that each pixel in the new image is the average of a five-by-five square of pixels in the original image. You could scan the subsampled image 25 times more quickly than the original; with the established barge size, you could still detect any barge in the lower resolution image. (The shape of a suspected barge could be determined by more detailed examination of the original image at higher resolution.) But if the established barge size was smaller, the low-resolution image might hide the barge inside of one of its averaged pixels. Thus, there is a direct relationship between the minimum barge size and the amount of speedup that can be accomplished by subsampling.

You can view the search for a barge as a search for faults in a program, but instead of examining pixel groups, you are examining the output from test-case executions. The barge was large enough to see at low resolution, so you could use a coarser grid to locate it. If a fault will always cause a larger proportion of inputs to fail during testing, you will need fewer random tests to reveal the fault. If we can guarantee that any fault in a program will cause the program to fail for a sufficiently large proportion of tests, then we can reduce the number of tests necessary to be confident that no faults exist.

Note that in these analogies we describe random testing, which is what we have emphasized in our work because of its attractive statistical properties. However, software testability could be defined for different types of testing, such as dataflow testing and mutation testing.

DESIGNING FOR TESTABILITY

Testability's goal is to assess software accurately enough to demonstrate whether or not it has high quality. If you use black-box testing alone to assess software, an intractable amount of testing is required to establish a very small probability of failure. For example, to assess a probability of failure that is less than 10^{-9} ($\theta \leq 10^{-9}$ failures per test, where θ represents the true probability of failure) with a confidence of 99 percent, approximately 4.6 billion successful executions (tests according to the input distribution) are needed! The practical problems of such testing are obvious. Furthermore, if during random black-box testing the software *does* fail, it must be fixed, and random black-box testing must be restarted. In other words, you must ignore all previous successful executions and redo the testing. Statistics show that whenever you write code, whether you add functionality or fix old code, 30 percent of that code will have new faults. Clearly, we need to seek new methods that increase testing effectiveness.

There are two ways to reduce the number of required tests:

- Select tests that have a greater ability to reveal faults.
- Design software that has a greater ability to fail when faults do exist (design for testability).

We favor the second strategy, but it imposes several criteria on program design:

- More of the code must be exercised for each input.
- Programs must contain constructs that are likely to cause the state of the program to become incorrect if the constructs are themselves incorrect.
- Programs must be able to propagate incorrect states into software failures.

How many and to what extent these criteria are met depends on what testing scheme you use, but software design for testability can improve the chances of incorporating at least one of these features. Moreover, designing software for testability also prevents the too little, too late problem: If the code that exists at the verification stage is flawed because of incorrect or inefficient design decisions, often little can be done to undo the mistakes without enormous additional costs. In integrated-circuit design, designing for testability has long been viewed as a necessary step in the overall process. IC design engineers have a notion — observability — that is closely related to software testability. Observability is the ability to view the value of a particular node embedded in a circuit. In hardware, the principal obstacle in testing large-scale ICs is the inaccessibility of the internal signals.[2]

SOFTWARE TESTABILITY IS RELATED TO HARDWARE OBSERVABILITY.

One method of increasing observability is to increase the chip's pin count, letting the extra pins carry out additional internal signals that can be checked during testing. In software, when modules contain local variables, you lose the ability to see information in the local variables during functional testing — something that will become a major issue for object-oriented systems. To remedy this, you can apply a notion similar to increasing the pin count in a chip: you can increase the amount of data-state information that is checked during unit testing.

Ideally, a design process begins with a (functional description, input distribution) pair that specifies the intended software. We believe that a theoretical upper bound exists on the testability that can be achieved for a given pair. If we can change the functional description to include more internal information, we should be able to increase that upper bound.

Information loss. Increasing this information helps compensate for the loss of information that occurs when internal information computed by a program is not communicated in the program's output. Information loss increases the potential that data-state errors will be canceled because the lost information may have contained evidence of incorrect data states. Therefore, information loss decreases testability. Information loss falls into one of two broad classes: implicit and explicit.

Implicit information loss. Implicit information loss occurs when two or more different incoming parameters are presented to a user-defined function or a built-in operator and produce the same result.

To illustrate, consider the case in which you have an integer-division computation, $a := a$ div 2, and two incoming values for a, 5 and 4. The result for both is that a is assigned 2. In the same example, suppose a user-defined function takes in two integer parameters and produces one Boolean parameter; many integer-2 tuples are possible, but only 0 or 1 result. In contrast, consider the computation $a := a + 1$, in which there is no implicit information loss.

In both these examples, you can predict that implicit information loss will occur by statically analyzing the code. If a specification states that 10 floating-point variables are to be input to an implementation, and two Boolean variables are to contain the implementation's output, then you know that there will likely be some implicit information loss.

For more specific estimates of implicit information loss, you can look at the program's specification. If a specification is written with enough information about its domain and range, for example, it can be used to estimate the *degree* of implicit information loss that will occur. In our design-for-testability strategy, we use a specification metric, the *domain-to-range ratio*, to help developers obtain this information. We emphasize, however, that a specification's DRR suggests only *part* of the

implicit information loss that may occur, and it will not always be discernible if a specification does not have enough information about the domain and range. It is also suitable for making only *rough* predictions about the degree of loss. You must inspect the code to get the necessary additional information for a more solid estimate.

The DRR is useful because it gives important information about possible testability problems in the code required to implement the specification and can help developers focus analysis and testing resources on the parts of the code that most need them. The DRR is the ratio of the cardinality of the specification's domain, denoted by α, to the cardinality of its range, denoted by β. Generally, as the DRR increases, the potential for implicit information loss increases. When α is greater than β, faults are more likely to remain undetected (if any exist) during testing than when α equals β.[3] Because evidence of incorrect data states is not visible in the output, the probability of observing a failure during testing is somewhat reduced. How much it is reduced depends on whether or not the incorrect information is isolated to bits in the data state that are not lost and are eventually released as output. As the probability of observing a failure decreases, the probability of undetected faults increases.

Another research report presents a similar conclusion about the relationship of faults remaining undetected and the type of function containing the fault.[4] While performing mutation testing experiments with Boolean functions — which typically have a high degree of implicit information loss — Brian Marick noted that faults in Boolean functions (where the cardinality of the range is 2) were more apt to be undetected than faults in other types of functions. This finding supports the idea that testability and the DRR are correlated. We are currently collecting additional evidence of a correlation between implicit information loss and testability.

Implicit information loss is common in many of the built-in operators of modern programming languages. Operators such as div, mod, and trunc have high DRRs. Table 1 contains a set of functions with generalized degrees of implicit information loss and DRRs, where (for simplicity) b is assumed to be a constant, and infinity

TABLE 1
DRRs AND IMPLICIT INFORMATION LOSS OF VARIOUS FUNCTIONS

	Function	Implicit Information Loss	DRR	Comment
1	$f(a) = \begin{cases} 0 & \text{if } a < 0 \\ a & \text{otherwise} \end{cases}$	yes	$\infty_I : \infty_I/2$	a is integer
2	$f(a) = a+1$	no	$\infty_I : \infty_I$	a is integer
3	$f(a) = a \bmod b$	yes	$\infty_I : b$	testability decreases as b decreases, $b \neq 0$
4	$f(a) = a \text{ div } b$	yes	$\infty_I : \infty_I/b$	testability decreases as b increases, $b \neq 0$
5	$f(a) = \text{trunc}(a)$	yes	$\infty_R : \infty_I$	a is real
6	$f(a) = \text{round}(a)$	yes	$\infty_R : \infty_I$	a is real
7	$f(a) = \text{sqr}(a)$	no	$2 \cdot \infty_R : \infty_R$	a is real
8	$f(a) = \text{sqrt}(a)$	no	$\infty_R : \infty_R$	a is real, $a \geq 0$
9	$f(a) = a/b$	no	$\infty_R : \infty_R$	a is real, $b \neq 0$
10	$f(a) = a - 1$	no	$\infty_I : \infty_I$	a is integer
11	$f(a) = \text{even}(a)$	yes	$\infty_I : 2$	a is integer
12	$f(a) = \sin(a)$	yes	$\infty_I : 360$	a is integer (degrees), $a \geq 0$
13	$f(a) = \text{odd}(a)$	yes	$\infty_I : 2$	a is integer
14	$f(a) = \text{not}(a)$	no	1 : 1	a is Boolean
15	$f(a,b) = (a)\text{or}(b)$	yes	4 : 2	a, b are Boolean

represents the cardinality of fixed-length number representations. Infinities with the subscript R represent the cardinalities of real numbers; infinities with the subscript I represent the cardinalities of integers.

You can also predict information loss from a description of the function to be programmed. A function classified as having a "yes" for implicit information loss is more likely to receive an altered incoming parameter and still produce identical output as if it had received the original incoming parameter. A function classified as having a "no" for implicit information loss is likely to produce an altered output if given an altered incoming parameter. In other words, a "yes" suggests the probability that data-state errors would be canceled; a "no" suggests that they would not.

Figure 1 illustrates the relationship between implicit information loss and the DRR. Sixteen a,b input pairs are presented to two functions: one performs real division; the other, integer division. For the real-division function, there are 16 unique outputs; for the integer-division function there is one. This supports the DRR classification in Table 1: For $f(a) = a/b$ (the real-division function), there is likely to be no implicit information loss. For $f(a) = a$ div b (the integer-division function), there is likely to be implicit information loss.

Explicit information loss. Explicit information loss occurs when variables are not validated either during execution (by a self-test) or at the end of execution as output. Explicit information loss frequently occurs as a result of *information hiding*, although other factors can contribute to it. Information hiding is a design philosophy that does not allow a module to release information that other modules could potentially misuse. This technique is widely accepted as good structured-programming practice, and we advocate structured programming, but hiding internal information is not good for testability at the system level, because the data in the local variables cannot be viewed in the search for faults.

Explicit information loss is harder to find early in development because it cannot be predicted by a DRR. The ability to find it depends more on how the software is designed, and less on the specification's I/O pairs. You can observe explicit information loss through static code inspection, and possibly by reviewing the design document if it is sufficiently detailed. The document can reveal things like the number of local variables or the number of times a variable is redefined as a new value. For example, $a := a + 1$ may be redefined as $a := a$ mod 2 and again as a := function (b,c,d,e,f). This redefinition is a form of explicit information loss.

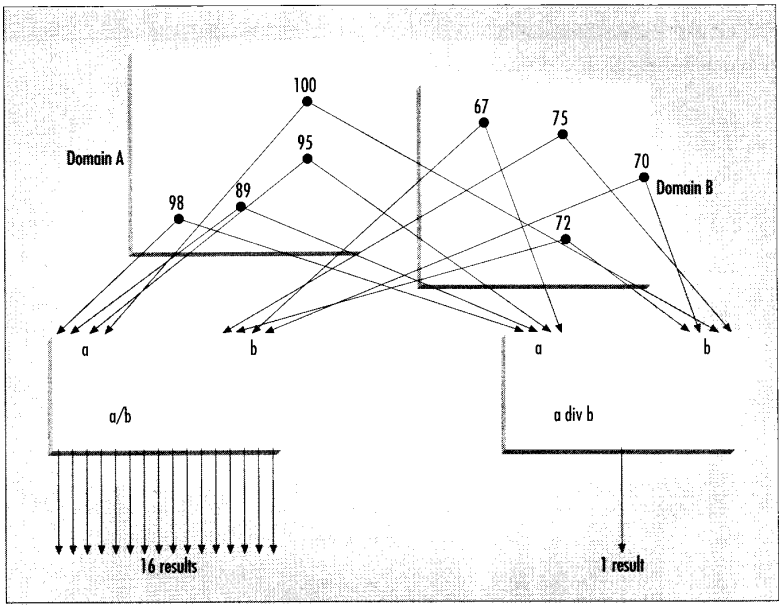


Figure 1. *How implicit information loss relates to the domain-to-range ratio. There are 16 values coming in: four values for variable a from domain A and four values for variable b from domain B. The cross product for (a,b) is 16 two-tuples. For a/b (real-number division), there are 16 results (no information loss), but for a div b (integer division), there are only two. You can predict implicit information loss by examining the type of function. Integer functions tend to incur a loss; real-number functions do not.*

Design heuristics. There are several ways to minimize the detrimental effects of both implicit and explicit information loss on testability, including decomposing the specification to isolate implicit information loss, minimizing the reuse of variables to reduce implicit information loss, and increasing the use of out parameters to reduce explicit information loss.

Specification decomposition. A major advantage of using the DRR to guide development is that it is available very early in the life cycle. Although the DRR of a specification cannot be modified without changing the specification itself, there are ways to decompose a specification to reduce the potential that data-state errors will be canceled across modules.

During specification decomposition, you have hands-on control of the

DRR of each subfunction. With this, you gain an intuitive feeling for how much testing is needed to attain a certain confidence that a module is propagating data-state errors. The rule of thumb that guides this intuition is "the greater the DRR, the more testing is needed to overcome the likelihood that data-state errors will be canceled."

You can also decompose a specification in a manner that classifies the program's modules as having a high or low DRR. By isolating modules with a low DRR — those that are more likely to propagate incoming data-state errors during program testing — you can shift testing and analysis resources to modules that are less likely to do that.

Minimization of variable reuse. As we demonstrated earlier, a computation such as $a := sqr(a)$ destroys the original value of a, and although you can take the square root after this computation and retrieve the absolute value that a had, you don't know if it is positive or negative. Minimizing variable reuse is one way to try to decrease the amount of implicit information loss.

To minimize variable reuse, you must either create more complex expressions or declare more variables. If you declare more variables, you will need more memory. If you use more complex expressions, you will reduce the testability when a single expression represents what were previously many intermediate values.

Although some literature supports programming languages based on few or no variables, programs written in such languages will almost certainly suffer from low testability. For this reason, we advocate using more variables, and thus making more variables available during testing. Clearly, adding more variables can decrease performance. However, you can gain a significant payoff in increased testability for only a minor cost in performance. Moreover, performance costs are machine-oriented, and the cost of machine resources is decreasing. Testability costs, on the other hand, are people-oriented, and human resources are becoming increasingly expensive.

Increased use of out parameters. As we described earlier, explicit information loss caused by local variables parallels the notion of low observability in ICs. Because explicit information loss suggests lower testability, we prefer, when possible, to lessen the amount of explicit information loss that occurs during testing. Even if you cannot actually reduce the loss, the reduction strategies we give here are still worth following because they tell you the location of modules with the greatest potential for data-state error cancellation before validation begins.

WE MUST VALIDATE MORE INTERNAL INFORMATION TO INCREASE TESTABILITY.

One approach to limiting the amount of explicit information loss is to insert `write` statements to print internal information. This information must then be checked for correctness during each test. A second approach is to increase the amount of output that these subspecifications return by treating local variables as out parameters during testing. A third approach inserts self-tests — called *assertions* — that are executed to check internal information during computation. When the assertion encounters an incorrect internal computation, it produces a message to that effect.

Our research suggests that assertions are particularly useful for testability analysis. Not only can you use them to ensure that a particular variable is correct or in the range at some point during execution, but also a failed assertion suggests the possibility that previous computations (on which the variable definition depends) might be incorrect. In addition, the messages about incorrect computations make it less likely that there are hidden faults.

These three strategies produce two important results, both of which essentially increase the software's observability:

- The people formalizing the specification are forced to produce detailed information about the states of the internal computations. This should increase the likelihood that the code is written correctly, and it forces the code to test itself.
- The dimensionality of the range of the intended function is increased, which may increase the cardinality of the range, thus reducing information loss.

In advocating these approaches, we are not repudiating the practice of information hiding during design. However, when writing safety-critical software in particular, there is the competing imperative of enhancing testability. Information not available during testing encourages undetected faults; increased output discourages undetected faults. An answer to this conflict may be to pattern software testing more closely on hardware testing by specifying special output variables that are specified and implemented specifically and exclusively for testing.

Analysis. All the strategies proposed to mitigate information loss require additional specified information about the internal computations. Maybe the real message of our research is that until developers make the effort to better specify what must occur, even at the intermediate computation level, testabilities and assessed reliabilities will remain low. In other words, *developers must validate more internal information if they hope to increase software testability*. To do this, there must be some way to check additional internal information, which means describing more information in the specification and requirements phase. If developers are not willing to specify these details at *some* point during this phase, they cannot expect to substantially improve reliability assessments.

SENSITIVITY ANALYSIS

Once you improve software testability, there must be some way to measure that improvement. To aid this process, we have devised a model[5] that quantifies testability on the basis of a *sensitivity analysis*. Sensitivity analysis quantifies behavioral information about the likelihood that faults are hiding. It repeatedly executes the original program and mutations of its source code and data states using two assumptions: The *single-fault* assumption says that the program contains a single fault, not multiple faults distributed throughout the program. The *simple-fault* assumption says that the fault exists in a single location, not distributed throughout the program, and that this fault is equally likely to be at any location in the program. The assumption of this single, randomly located error is a variation on the competent programmer hypothesis, which maintains that a competent programmer will write code that is reasonably close to being correct.

The specific purpose of sensitivity analysis is to provide information that suggests how small the program's smallest faults are likely to be. With this prediction, you can use statistical methods to determine how much testing will be necessary to detect faults of this size, thus obtaining a criterion that lets you determine when to stop testing.

SENSITIVITY ANALYSIS SUGGESTS HOW SMALL THE SMALLEST FAULTS ARE LIKELY TO BE.

To provide this prediction, sensitivity analysis injects simulated faults into the code and estimates their effect on software observability. Its success depends heavily on the testing scheme used to take this measurement. At the very least, the testing scheme must exercise all locations. (Other qualifications are a matter for debate.)

The strength of sensitivity analysis is that your prediction is based on observed effects from actual faults; the weakness is that the faults injected and observed are only a small set from what might be an infinite class of faults.

For a particular location in the code, sensitivity analysis estimates the probability of failure that would be induced in the program by a single fault. For a failure to occur and be observed, three things must happen: the fault must be executed, an incorrect data state must be created (and the original data state becomes "infected"), and the incorrect state must be propagated to a discernible output. Thus, sensitivity analysis separates failure into three types of events — execution, infection, and propagation — and applies analysis algorithms to estimate the probability of each event.

Sensitivity analysis is broken into three independent processes, each of which estimates the likelihood of one of the three events: execution analysis, infection analysis, and propagation analysis. (Although we describe them sequentially, in a production-analysis system they could overlap.) To estimate the likelihood of an event, each process divides the number of times the event occurred by the number of attempts to force that event. For example, if the propagation event occurs 10 out of 100 times, the propagation probability estimate is 0.1.

The result of sensitivity analysis is the estimated *probability of failure* that would result if a particular location had a fault. This estimate is obtained by multiplying the means of the three estimates from the analysis phases. If you take the minimum over all three estimates and then obtain a product, you can obtain a *bound* on the minimum probability of failure that would result if this location had a fault. Elsewhere, Voas formalizes the method to find a predicted probability of failure from sensitivity analysis.[5]

Execution analysis. Execution analysis estimates the probability of execution for each location by repeatedly executing the code with inputs selected from an input distribution or inputs from a test suite. As with the analysis of random testing, the accuracy of execution analysis depends in part on how well you have estimated the input distribution that will drive the software when it is in use. Execution analysis differs from random testing, however, in that it answers the question, "How often does this code get executed?" for a single location. Random testing answers the question, "Is this code correct?" for the entire program.

In execution analysis, a single location is analyzed with respect to the number of test cases that execute it. Our experiments thus far have defined a location as a piece of source code that can change the data state (including I/O files and the program counter). A location could be a single statement in a high-level language, one code instruction, or some intermediate amount of computation. For example, an assignment statement and an `if` statement define one location (because they involve only one variable), while a statement `read(a,b)` defines two locations (because it involves two variables).

Probability is determined by the number of times a location is executed relative to the total number of test cases run (not relative to each test case). For example, if 100 test cases are run and the location is executed in 40 of them, the execution probability is 0.4.

Infection analysis. If a location contains a fault, and if the location is executed, the fault may make the data state incorrect for that input. If so, the data state becomes infected. To estimate the probability of infection, the infection-analysis algorithm performs a series of syntactic mutations on each location. A syntactic mutation is a change from the original syntax into a new syntax that is grammatically legal and has a different meaning for at least one input value.

After each mutation, the program is reexecuted with random inputs. As part of estimating the probability of infection, each time the monitored location is executed, the infection-analysis algorithm immediately compares the data state with the data state of the original (unmutated) program at that same point in the execution. If the state differs, infection has taken place.

Propagation analysis. In this phase, the location in question is monitored during random tests. After the location is executed, the propagation analyzer changes the resulting data state by assigning a random value to one data item using a predetermined distribution. (Research is ongoing as to the best distribution to use for this random selection.) After the data state is changed, the program continues executing until an output results. The propagation-analysis algorithm compares the output from the changed data state with the output that would have resulted without the change. If the outputs differ, propagation has occurred and a propagation probability can be estimated.

Implementation. The single-fault and simple-fault assumptions underlying sensitivity analysis are admittedly flawed and artificially restrict fault classes. However, without these assumptions the combinatorics of simulating classes of distributed or multiple faults becomes intractable. Moreover, despite this theoretical weakness, empirical techniques have yielded impressive experimental results.[5]

Sensitivity analysis is a new, empirical technique. The complexity of the processing required for sensitivity analysis is quadratic in the number of code locations and therefore requires considerable bookkeeping and execution time. Pilot experiments in the early 1990s were done using hand-coded syntactic mutations and only semiautomated data-state mutations. However, because sensitivity analysis does not require an oracle, it can be completely automated for programs of any size, although processing time can be a practical limit for large programs analyzed in a single block. Reliable Software Technologies Corp. has built a fully automated and commercialized sensitivity-analysis tool, Pisces 1.5, and applied it to systems as large as 100,000 source lines of code. The tool can operate on larger systems, but to our knowledge has not.

Sensitivity analysis lets you determine how much system-level testing you need to gain a certain level of confidence that faults are not hiding. It helps identify regions of code with extremely low testability, which require additional unit testing or other verification and validation resources. Additional benefits are possible with slight modifications to the sensitivity-analysis algorithms, which we describe in detail elsewhere.[6] Such benefits include increased fault tolerance and improved safety assessment.

SENSITIVITY ANALYSIS DOES NOT REQUIRE AN ORACLE, SO IT CAN BE AUTOMATED.

We believe the results of our experiments[5,7] are sufficient to motivate additional research and use with this technique. Although we cannot guarantee that you can use it to assess reliability with the precision required for safety-critical software, we believe it is premature to dismiss this possibility.

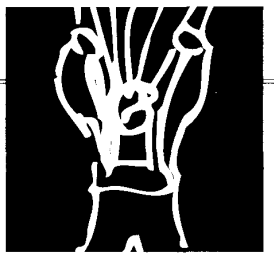
COMBINING TECHNIQUES

If software testability produces accurate predictions, then you should be able to combine random black-box testing with sensitivity analysis to assess reliability more precisely than is possible with black-box testing alone. Both random black-box testing and sensitivity analysis gather information about an estimated probability of failure. However, the two techniques generate information in distinct ways:

- Random testing treats the program as a single, monolithic black-box. Sensitivity analysis examines the source code location by location.
- Random testing requires an oracle to determine correctness. Sensitivity analysis requires no oracle because it does not judge correctness.
- Random testing involves analyzing the possibility that there are no faults. Sensitivity analysis assumes that one fault exists.

Although a program's true probability of failure, conditioned on an input distribution, is a single fixed value, the exact value is unknown. For that reason, in our approach to combine the two techniques, we treat the probability of failure as a random variable Θ and estimate a *probability density function* for Θ, conditioned on an input distribution. A pdf for some domain defines the likelihood that any element in the domain is chosen. Here we are concerned with predicting a pdf for Θ using both black-box testing and sensitivity analysis. The pdf in Figure 2a, pdf_B, transforms the number of random tests executed without discovering a failure into information about the likely reliability of the software. The pdf in Figure 2b, pdf_S, transforms the sensitivity analysis at each program location into information about the possible size of a single fault somewhere in the program. The prediction of pdf_S is conditioned on the same input distribution as pdf_B, but pdf_S is also conditioned on the assumption that the program contains exactly one fault, and that this fault is equally likely to be at any location in the program.

In Figures 2a and 2b, for each horizontal location θ, the height of the curve indicates the estimated probability that the program's true probability of failure is less than θ. In Figure 2a, we assume that the testing has uncovered no failures. As the number of tests increases, we expect probabilities of failure near 0.0 to be more likely and



those closer to 1.0 to be less likely. Of course after only one test that produces the correct output, $Pr\ \theta = 1.0 = 0.0$. Details about deriving an estimated pdf for Θ, given many random tests, are provided elsewhere.[8]

As Figure 2a shows, we mark interval estimates for each estimated pdf. If the interval between 0.0 and $\hat{\theta}$ includes 90 percent of the area under the estimated pdf, then according to random testing the actual probability of failure is less than $\hat{\theta}$ with a confidence of 90 percent. Similarly, if the interval in Figure 2b includes 10 percent of the area under the estimated pdf, then according to sensitivity analysis, if a fault exists, there is a 90 percent confidence that the probability of failure is greater than $\hat{\gamma}$.

Testing is unlikely to find a fault that induces a near-zero probability of failure. Locations that have sensitivity estimates very close to zero are troubling in an application that demands extreme reliability. However, a fault that induces a probability of failure of exactly 0 is technically not a fault at all — no failures will be observed with such a fault.

If there are no faults in a program, then the true probability of failure is 0 ($\theta = 0.0$), and we have achieved ultra-reliability (or correctness). Figure 2a suggests that if there is a fault, it is likely to induce a small probability of failure; Figure 2b suggests that *tiny* impact faults (those that cause the program to fail with probabilities less than $\hat{\gamma}$) are unlikely.

We now attempt to quantify the meaning of the two estimated pdfs taken together. Hamlet has derived an equation to determine what he calls *probable correctness*.[9] When T tests have been executed and no failures have occurred, then

$$C = Pr(\theta \leq \hat{\theta}) \geq 1 - (1 - \hat{\theta})^T \qquad (1)$$

where C is probable correctness, θ is the true probability of failure, $\hat{\theta}$ is some approximation of θ, and $0 < \hat{\theta} \leq 1$. (Hamlet calls C a measure of probable correctness, but it would be called a *confidence in correctness* if the equations were cast in a traditional hypothesis test.) $1 - C$ is the likelihood that $\theta > \hat{\theta}$, meaning that we have been fooled into thinking that $\theta \leq \hat{\theta}$, when really $\theta > \hat{\theta}$.

Let γ represent the impact caused to the true probability of failure by the smallest fault in the program; then γ is the smallest possible nonzero probability of failure for the program if we removed all other independent faults. If there are other faults, $\gamma < \theta$. γ is assumed to be unknown. Let $\hat{\gamma}$ represent the prediction of γ from sensitivity analysis according to our code; the testing distribution, D; and the fault classes that sensitivity analysis simulated. Note that one of the following situations is true, but we cannot know which it is: $\hat{\gamma} > \gamma$ or $\gamma \geq \hat{\gamma}$.

After testing T times and finding no failures, for any $\hat{\gamma}$, we have confidence C':

$$C' = Pr(\theta \leq \hat{\gamma}) \geq 1 - (1 - \hat{\gamma})^T \qquad (2)$$

Given that actual failure probabilities in the interval $(0, \hat{\gamma})$ are unlikely, our confidence that $\theta = 0.0$ is just

$$Pr(\theta = 0.0) \geq 1 - [Pr(\theta \geq \hat{\gamma}) + Pr(\gamma < \hat{\gamma})] \qquad (3)$$

where $Pr(\gamma < \hat{\gamma})$ is the probability that we failed to correctly assess the minimum probability of failure induced by any fault in our program from the fault classes that we simulated. (Assessing $Pr(\gamma < \hat{\gamma})$, which is outside the scope of this article, requires two additional probabilities: the probability of an actual fault causing a lower impact to the actual failure probability than $\hat{\gamma}$, and the probability that the order of magnitude of $\hat{\gamma}$ is too precise for the number of input values used in determining it, which is a statistical approxi-

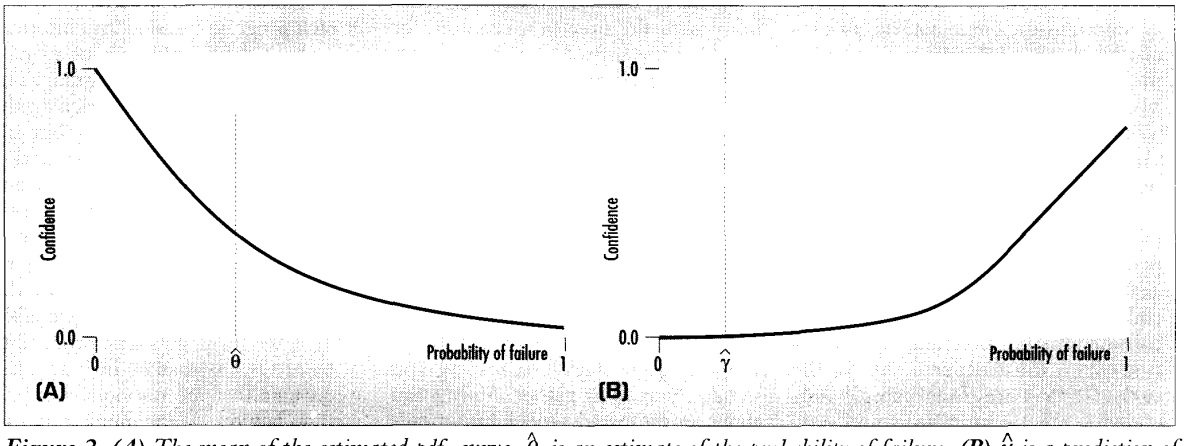


Figure 2. *(A) The mean of the estimated pdf_B curve, $\hat{\theta}$, is an estimate of the probability of failure.* ***(B)*** *$\hat{\gamma}$ is a prediction of the minimum probability of failure using sensitivity analysis.*

mation error. This problem is partially formalized elsewhere.[10]) $Pr(\gamma < \hat{\gamma})$ is a function of the fault classes that were simulated and the sample size of test cases from D that were used during sensitivity analysis. There could be a fault in our program from a fault class not simulated that has a smaller impact on γ than the fault classes we did consider.

To better understand equation 3, assume that we have tested T times and found no errors. Assume further that sensitivity analysis has selected $\hat{\gamma}$ as the smallest probability of failure induced by any of the faults it simulated. We use $\hat{\gamma}$ as a reference point for establishing a confidence in this software. Trivially,

$$\begin{aligned} 1 &= Pr(\theta = 0) + Pr(0 < \theta < \hat{\gamma}) + \\ &\quad Pr(\theta \geq \hat{\gamma}) \rightarrow Pr(\theta = 0) \\ &= 1 - Pr(0 < \theta < \hat{\gamma}) - \\ &\quad Pr(\theta \geq \hat{\gamma}) \end{aligned}$$

Thus we can estimate the probability that the program is correct by estimating $Pr(0 < \theta < \hat{\gamma})$ by sensitivity analysis and by estimating $Pr(\theta \geq \hat{\gamma})$ using Hamlet's probable correctness equation and T.

The goal of this squeeze play is to push $\hat{\theta}$ toward $\hat{\gamma}$ in Figure 2 when T is fixed and confidence is high.[11] As $\hat{\theta}$ approaches or exceeds $\hat{\gamma}$, we can be increasingly confident that the software is correct. As an example of equation 3, suppose we have a program with three independent faults, with impacts to θ of 0.0001, 0.00001, and 0.000001. In this situation, θ = 0.000111, and γ = 0.000001. Suppose $\hat{\gamma}$ = 0.000015. In this situation, $\hat{\gamma} > \gamma$, but $\hat{\theta} > \hat{\gamma}$. If C' is fixed close to 1.0 (meaning that the T we will try is large enough with respect to $\hat{\gamma}$), the likelihood that the program will fail at least once in T tests is also close to 1.0; hence, we are unlikely to be able to apply equation 3 because a failure should occur.

Now suppose that we remove the faults with failure probabilities of 0.0001 and 0.00001 after observing one or more failures, and when we perform sensitivity analysis again on the modified program, $\hat{\gamma}$ is still 0.000015. (In this program, we now have a single fault, so $\gamma = \theta$ = 0.000001.) Again we will test this code T times to fix C' close to 1.0. Because $\theta < \hat{\gamma}$, $Pr(\gamma < \hat{\gamma}) = 1.0$ and $Pr(\theta = 0.0) = 0.0$. Given that we cannot know the probability of a true fault causing a lower impact to the actual failure probability than $\hat{\gamma}$ (without knowing where all the faults are), the best that we can say about a confidence in absolute correctness (based on T successful tests and $\hat{\gamma}$) is that we have confidence C'' that the true probability of failure is zero:

$$\begin{aligned} C'' &\geq 1 - Pr(0 < \theta \leq \hat{\gamma}) - Pr(\theta > \hat{\gamma}) \\ &= 1 - [2e^{-2T'\varepsilon^2} + (1 - \hat{\gamma})^{T'}] \end{aligned}$$

where T' is the number of test cases used during sensitivity analysis and ε is a small fudge factor subtracted from $\hat{\gamma}$ to allow for imprecision during sensitivity analysis.

This is one method for combining testability analysis with testing results to sharpen an estimate of the true probability of failure. This use of testability measurement is essentially a cleanup operation — a method to assess if software has achieved a desired level of reliability. We believe that testability assessment is more useful earlier in the development of software. This idea is dramatized in Table 2, which gives you a feeling for the cost of testing to different levels of confidence, given different degrees of testability.

TABLE 2
SAMPLE VALUES FOR TESTABILITY ASSESSMENTS

T	$\hat{\gamma}$	C'
7	0.10	0.50
44	0.10	0.99
458	0.01	0.00
298	0.01	0.95
460,514	10^{-5}	0.99
46,051,699	10^{-7}	0.99
693,147,181	10^{-9}	0.50
2,302,585,093	10^{-9}	0.90
2,995,732,273	10^{-9}	0.95

Our research suggests that software testability clarifies a characteristic of programs that has largely been ignored. We think that testability offers significant insights that are useful during design, testing, and reliability assessment. In conjunction with existing testing and formal verification methods, testability holds promise for quantitative improvement in statistically verified software quality.

We are particularly interested in designing software to increase its testability. Although the existence of an upper bound on testability is solely our conjecture, our research using sensitivity analysis and studying software's tendency to not reveal faults during testing suggests that such a bound exists. We challenge software testing researchers to consider this.

A given piece of software will or will not hide a given fault from testing. We have found that it is possible to examine this code characteristic — software testability — without knowing if a particular fault exists in that software, and without reference to correctness. Because it does not rely on correctness, software testability gives a new perspective on code development.

We have briefly described one dynamic technique, software sensitivity analysis, for predicting software testability. Sensitivity analysis has yielded promising results in several experiments;[5] research and practical application of this technique continue. Perhaps other more effective or more efficient testability measurement techniques will be discovered, but whatever techniques are employed to measure testability, we are convinced that this inherent software characteristic will become an important factor to consider during software development and assessment. ◆

ACKNOWLEDGMENTS

We thank Christoph Michael and Sue Brilliant for reviewing earlier drafts of this article.

This work was supported in part by a National Research Council NASA-Langley Resident Research Associateship and NASA Grant NAG-1-884.

REFERENCES

1. R. Butler and G. Finelli, "The Infeasibility of Experimental Quantification of Life-Critical Software Reliability," *Proc. SIGSoft*, ACM Press, N.Y., 1991, pp. 66-76.
2. N. Berglund, "Level-Sensitive Scan Design Tests Chips, Boards, System," *Electronics*, Mar. 15, 1979, pp. 108-110.
3. J. Voas, "Factors That Affect Software Testability," *Proc. Pacific Northwest Software Quality Conf.*, PNSQC, Portland, 1991, pp. 235-247.
4. B. Marick, "Two Experiments in Software Testing," Tech. Report UIUCDCS-R-90-1644, CS Dept., University of Illinois at Urbana-Champaign, Nov. 1990.
5. J. Voas, "PIE: A Dynamic Failure-Based Technique," *IEEE Trans. Software Eng.*, Aug. 1992, pp. 717-727.
6. J. Voas and K. Miller, "Dynamic Testability Analysis for Assessing Fault Tolerance," *High Integrity Systems J.*, Vol. 1, No. 2, pp. 171-178.
7. J. Voas, K. Miller, and J. Payne, "A Comparison of a Dynamic Software Testability Metric to Static Cyclomatic Complexity," *Proc. Conf. Software Quality Management*, Computational Mechanics Publications, South Hampton, UK, July 1994, pp. 431-445.
8. K. Miller et al., "Estimating the Probability of Failure When Testing Reveals No Failures," *IEEE Trans. Software Eng.*, Jan. 1992, pp. 33-44.
9. R. Hamlet, "Probable Correctness Theory," *Information Processing Letters*, June 1987, pp. 17-25.
10. J. Voas, C. Michael, and K. Miller, "Confidently Assessing a Zero Probability of Software Failure," *Proc. Conf. Computer Safety, Reliability, and Security*, Springer-Verlag, London, 1993, pp. 197-206.
11. R. Hamlet and J. Voas, "Faults on Its Sleeve: Amplifying Software Reliability Assessment," *Proc. SIGSoft*, ACM Press, N.Y., June 1993, pp. 89-98.

Jeffrey M. Voas is vice president of Reliable Software Technologies, where he is currently prinicpal investigator on basic research grants with NASA, the National Institute of Standards and Technology, and the National Science Foundation. His research interests include assessing and improving software fault tolerance and measuring software dependability and safety. He is coauthor of *Software Assessment: Reliability, Safety, Testability* (John Wiley & Sons, to be published).

Voas received a BS in computer engineering from Tulane University and an MS and a PhD in computer science from the College of William and Mary. He is a member of the IEEE.

Keith W. Miller is an associate professor of computer science at Sangamon State University. His research interests include software engineering, software reliability, and computer ethics. His work has included consulting for NASA Langley Research Center, Computer Sciences Corp., and Bell Atlantic. He is also program chair for the 1995 National Educational Computing Conference.

Miller received a PhD in computer science from the University of Iowa. He is a member of ACM and the IEEE.

Address questions about this article to Voas at Reliable Software Technologies, 21515 Ridgetop Cir., Sterling, VA 20166; jmvoas@rstcorp.com.