Software Testability: The New Verification[*]

Jeffrey M. Voas
Reliable Software Technologies Corp.
Suite 250
11150 Sunset Hills Road
Reston, VA 22090
(703) 742-8873
jmvoas@isse.gmu.edu
fax: (703) 742-9836

Keith W. Miller
Department of Computer Science
Health Sciences Building 137
Sangamon State University
Springfield, IL 62794
(217) 786-7327
miller@eagle.sangamon.edu

Abstract

Software verification encompasses a wide range of techniques and activities that are geared towards demonstrating that software is reliable. Verification techniques such as testing provide a way to assess the likelihood that software will fail during use. This paper introduces a different type of verification that shows how likely it is that an incorrect program will not fail. Our verification applies fault-injection methods to predict where actual faults are more likely to hide. This verification can be combined with software testing to assess a confidence that the code is not hiding faults.

Code that hides faults is difficult to test. In order to minimize the problem of hidden faults, we seek methods for identifying and isolating source code that is likely to hide faults. We also introduce the notion of "information loss," a characteristic that can be measured during the early phases of design to suggest where the planned software is likely to harbor faults that will be difficult to uncover during testing.

Keywords

Software testing, software testability, fault, failure, reliability, probability of failure, software design-for-testability.

[*]Supported in part by a National Research Council Nasa-Langley Resident Research Associateship and NASA Grant NAG-1-884.

1 Introduction

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 computer software begins to replace human decision makers, a fundamental concern is whether a machine will be able to perform the tasks with the same level of precision as a skilled person. If not, a catastrophe may be caused by an automated system that is less reliable than a manual system. Therefore we must have a means of assessing that critical automated systems are acceptably safe and reliable. In this paper, we concentrate on a verification technique for assessing reliability.

The *IEEE Standard Glossary of Software Engineering Terminology* (1990) defines software verification to be 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 degree of "acceptability" of the software, where acceptability is judged according to the specification. Software verification is broadly divided into two classes: dynamic software testing and formal verification (which typically involves some level of static theorem proving). *Dynamic software testing* is the process of executing the software repeatedly until a confidence is gained that either (1) the software is correct and has no more defects, which is commonly referred to as *probable correctness* [3], or (2) the software has a high enough level of acceptability. Testing can alternatively be subdivided into two main classes: white-box and 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.

Static theorem proving is the mathematical process of showing that the function computed by a program matches the function that is specified. No program executions occur in this process, and the end 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 concerning 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 paper we describe a different type of verification that can complement both dynamic testing and static theorem proving. This new type of verification, which we will call "software testability," focuses on the probability that a fault in a program will be revealed by testing. We define *software testability* as 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.

Verification, by the standard IEEE definition, is a way of assessing whether the input/output pairs are correct. Testability examines a different behavioral characteristic: *the likelihood that the code can fail if something in the code is incorrect.* Computer science researchers have spent years developing software reliability models to answer the question: "*what is the probability that this code will fail?*" Our testability asks a different question: "*what is the probably this code will fail if it is faulty?*" Musa labels a similar measurement as the *fault exposure ratio*, K, in his reliability formulae [6]. The empirical methods for estimating testability are distinct from Musa's techniques, however.

Our research has emphasized random testing, because of its attractive statistical properties. However, in full generality, software testability could be defined for different types of testing (e.g. data flow testing, mutation testing, etc.). The *IEEE Standard Glossary of Software Engineering Terminology* (1990) 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, 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."*

Note here that in order to determine "the degree" you must have a "test criteria," and hence testability is simply a measure of how hard it is to satisfy a particular testing goal. Examples of testing goals include coverage and 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 [3].

The reader should note that our definition of software testability differs from earlier definitions of testability such as the IEEE definition above. 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 desired full branch coverage during testing and found it difficult to select inputs that cover more than 50% of the branches, then the software would be classified as having poor testability. Our definition differs significantly, because we are not just trying to find sets of inputs that satisfy coverage goals; we are quantifying the probability that a particular type of testing will cause failure. 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.

Software testability analysis is related to but distinct from both software testing and formal verification. Like software testing, testability analysis requires empirical work to create estimates. Unlike testing, testability analysis does not require an oracle. Thus testing and testability are complementary: testing can reveal faults (testability cannot) but testability can suggest locations where faults can hide from testing (something testing cannot do). The next section explores how testability analysis can be used in conjunction with testing and formal methods to give a clearer view of developing software.

2 Three Pieces of a Puzzle

Software testability, software testing, and formal verification are three pieces in a puzzle: the puzzle is whether the software that we have has a high enough *true* reliability.[1] If we are lucky enough to have a piece of software that (1) has undergone enormous amounts of successful testing, (2) has undergone formal verification, and (3) has high testability, then we have three pieces that fit together to *suggest* that the puzzle is solved—high reliability is achieved.[2]

Software testing, formal verification, and software testability all offer information about the quality of a piece of software. Each technique supplies a unique perspective, different evidence that the analyst must take into account. Even with these three "clues," the analyst is still making guesses; but having all three clues is better than having only two.

As a hypothetical example of how the three analyses can work together, consider a system that has 50 modules. Each of the modules is tested with 100 random tests, and (in their current versions) all modules pass these tests. In addition, the system passed 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 5 of the modules are highly insensitive to testing; i.e., testing is unlikely to find faults in these modules if faults exist. Only one of these 5 modules has been formally verified. At this point, verification resources should concentrate on the 4 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 analysts 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 non-random tests are devised to exercise these sections more extensively.

[1]Every system has a true (or fixed) reliability which is generally unknown; hence we try to estimate that value through reliability modeling.

[2]There is no widely accepted threshold for when software changes from being reliable to being "highly" reliable, however many use the 10^{-9} failures in a 10/hour flight as that threshold, knowing that testing alone can never demonstrate this degree of precision [2].

These examples illustrate that testability information cannot replace testing and formal verification; but they also suggest that testing and formal verification should not be relied upon exclusively either. Our view of software development and verification is inclusive. The most effective software practitioners will take advantage of all available information in order to build and assess quality software. In the rest of this paper, we focus on software testability; but this discussion is always in the context of a technique that is complementary to testing and formal verification.

The remainder of this paper elaborates on three topics: software testability, designing for testability, and sensitivity analysis. In the next section we describe in more detail how testability relates to testing and formal analysis. Then we look at the mechanics of predicting testability at an early stage of development. We describe the concept of "information loss," a characteristic that can be predicted from an initial design description of the function to be programmed. The information loss characteristic provides insights that can be used to improve testability by basing software design-for-testability heuristics on it. We will describe "sensitivity analysis," a technique that quantifies behavioral information about the likelihood of faults hiding. Sensitivity analysis is dynamic and empirical, relying on repeated executions of the original and mutations of its source code and data states. Sensitivity analysis allows us to assess how much testability has been achieved by applying design-for-testability schemes. The final section shows how the predictions of sensitivity analysis can be included in estimates of software reliability.

3 Software Testability: The Big Picture

To better provide a understanding of what we mean by software testability, consider two simple analogies.

If software faults were gold, then software testing would be gold mining. Software testability would be a geologist's survey done before mining takes place. It is not the geologist's job to dig for gold. Instead, the geologist establishes the likelihood that digging at a particular spot would be rewarding. A geologist might say, "This valley may or may not have gold, but if there is gold, it will be in the top 50 feet, and it will be all over this valley." At another location, the geologist might say, "Unless you 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."

When software testing begins, such an initial survey has obvious advantages over testing blind. Testability suggests the testing intensity, whereas the geologist gives the digging depth. Testability provides the degree of difficulty which will be incurred during testing of a particular location to detect a fault. If after testing to the degree specified by testability we observe no failures, then we can be reasonably sure that our program is correct[3].

In a second analogy, we illustrate how fewer tests can yield an equivalent confidence in correctness *if* we are sure the software will not hide faults: imagine that you are writing a program for scanning black and white satellite photos, 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 at least a 10 by 20 pixel area in an image, then the program can use techniques that could not be used if the barge size were not established beforehand. For example, assume that 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. This subsampled image could be scanned 25 times more quickly than the original; with the barge size guaranteed to be large enough, any barge would still be detectable 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 a barge might exist in the image as a smaller rectangle, then the low resolution image might hide the barge inside of one of its averaged pixels. The lower bound on barge size makes the lower resolution image sufficient for locating barges. There is a direct relationship between the minimum barge size and the amount of speed-up that can be accomplished by subsampling.

Looking for a barge in the image is analogous to looking for faults in a program; instead of examining groups of pixels, we examine the output from test case executions. If a fault always will cause a larger proportion of inputs to fail during testing (this is analogous to a bigger barge), then fewer random tests will be required to reveal the fault (a coarser grid can be used to locate the barge). 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. These two analogies indicate why testability information is a useful strategy complementary to testing. In the next section we discuss how to design testability into your system.

4 Designing for Testability: An Ounce of Prevention

Our goal is to assess software quality accurately enough to demonstrate whether or not is has high quality. If we use black-box testing alone to assess software, an intractable amount of testing is required to establish a very small probability of failure. (We will use the variable θ to represent the true probability of failure.) To assess a probability of failure that is less than 10^{-9} (i.e., $\theta \leq 10^{-9}$ failures/test) with 99% confidence, it can be shown that approximately 4.6 billion successful executions (tests according to the input distribution) are needed. Even when a foolproof automated

[3]By "reasonably sure" we suggest a colloquial confidence rather than statistical confidence. We discuss more precise quantifications in a later section

oracle is available, the practical problems of such testing are clear. Furthermore, if during random black-box testing the software *does* fail, then the software must be fixed and random black-box testing must completely restart (i.e., we must ignore all previous successful executions and redo the testing). (Statistics have shown that as much as 30% of all new code contains new faults.) Clearly, we need to seek new methods that increase the effectiveness of testing.

To reduce the number of required tests to a more tractable number, one of two techniques are available:

1. Select tests that have a greater ability to reveal faults. Research into test coverages and mutation adequate test sets seeks to find more effective individual tests and sets of tests.

2. Design software that has a greater ability to fail when faults *do* exist.

Designing programs to "fail big" when they have faults means that we must create programs that (1) are likely to have larger proportions of the code exercised for each input, (2) contain program constructs that are likely to cause the state of the program to become incorrect if the constructs are themselves incorrect, (3) propagate incorrect program states into software failure. Software design-for-testability addresses the issue of developing code that "fails-big"; this topic is the focus of this section.

A problem with code-based verification techniques is that they are applied late in the software life cycle. If the code that exists at this point is flawed from incorrect or inefficient design decisions, then often little can be done during verification to undo the mistakes without enormous additional costs. This same problem exists for viewing testability only after the code is produced. But testability can be addressed much earlier in the life-cycle.

In integrated circuit design, testability has long been viewed as a required characteristic. Integrated circuit design engineers have a notion termed "observability," a notion that is closely related to software testability. *Observability* is the ability to view the value of a particular node that is embedded in a circuit. In software, when modules contain local variables, you lose the ability to see information in the local variables during functional testing.[4] In this sense, having local variables in software is analogous to lowering observability in circuits.

Discussing the observability of integrated circuits, [1] states that the principal obstacle in testing large-scale integrated circuits is the inaccessibility of the internal signals. One method used for increasing observability in integrated circuits design is to increase the pin count of a chip, allowing the extra pins to carry out additional internal signals that can be checked during testing. These output pins increase observability by increasing the range of potential bit strings from the chip.

[4]We feel that this will become an issue in functional testing of object-oriented systems.

We can apply a similar notion to increasing the pin count during software testing – increasing the amount of data state information that is checked during unit testing.

Information loss occurs when internal information computed by a program during execution is not communicated in the program's output. Information loss increases the potential for the cancellation of data state errors and therefore decreases software testability. We divide information loss into two broad classes: implicit information loss and explicit information loss.

Explicit information loss occurs when variables are not validated either during execution (by a self-test) or at execution termination as output. Explicit information loss can be observed using a technique such as static data flow analysis. Explicit information loss frequently occurs as a result of information hiding, although there are other factors that can contribute to it. *Information hiding* is a design philosophy that does not allow information to leave modules that could potentially be misused by other modules. Information hiding is widely accepted as good structured programming practice, which we advocate. However, hiding internal information is not good for testability at the system level, because the data in the local variables are then not available for revealing faults.

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. As an example of this at the operator level, consider the integer division computation **a := a div 2**; here, two different incoming values for **a**, 5 and 4, both result in **a** being assigned 2. Consider a user-defined function that takes in two integer parameters and produces one boolean parameter; here, many different integer 2-tuples are possible, while only '0' or '1' result. Now consider the computation **a := a + 1** in which there is no implicit information loss. In these two examples, the potential for implicit information loss occurring can be observed by statically analyzing the code. If a specification states that ten floating-point variables are input to an implementation, and 2 boolean variables contain the implementation's output, then we know that implicit information loss will occur in an implementation of this specification. If they are written with enough information concerning their domain and ranges, specifications can be used to estimate the degree of the implicit information loss that will occur when the specification is implemented.

4.1 The Domain/Range Ratio

Clues suggesting some degree of the implicit information loss that may occur during execution may be visible from the program's specification; we use a specification metric termed the "domain/range ratio" for suggesting a degree of implicit information loss. Recall that in the example we were also able to observe implicit information loss by code inspection. Therefore, a specification's domain/range ratio only suggests a portion of the implicit information loss that may occur; code inspection can give additional information concerning implicit information loss.

The *domain/range ratio* (DRR) of a specification is the ratio between the cardinality of the domain of the specification and the cardinality of the range of the specification. We denote a DRR by $\alpha : \beta$, where α is the cardinality of the domain, and β is the cardinality of the range. As previously stated, this ratio will not always be discernible from a specification. However, when the DRR can be estimated, it gives important information about possible testability problems in the code required to implement the specification.

DRRs roughly predict a degree of implicit information loss. Generally as the DRR increases for a specification, the potential for implicit information loss occurring within the implementation increases. When α is greater than β, previous research has suggested that faults are more likely to remain undetected (if any exist) during testing than when $\alpha = \beta$. When implicit information loss occurs, you run the risk that the lost information may have included evidence of incorrect data states. Since such evidence is not visible in the output, the probability of observing a failure during testing is somewhat reduced. The degree to which it is reduced depends on whether 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 that presents a similar conclusion concerning the relationship between faults remaining undetected and the type of function containing the fault is [5]. While performing mutation testing experiments with boolean functions, Marick [5] 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. Boolean functions typically have a great degree of implicit information loss. This result supports the idea that testability and the DRR are correlated. Additional evidence that correlation exists between implicit information loss and testability is currently being collected.

4.1.1 Correlating Implicit Information Loss and the DRR

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. A function classified as having a *yes* for implicit information loss in Table 1 is more likely to receive an altered incoming parameter and still produce identical output as if the original incoming parameter were used; a function classified as having *no* implicit information loss in Table 1 is one that if given an altered incoming parameter would produce altered output. A *yes* in Table 1 suggests data state error cancellation would occur; a *no* suggests data state error cancellation would not occur. In Table 1, all references to b assume it is a constant for simplicity. The infinities in the table are mathematical entities, but for any computer environment they will represent the cardinality of fixed length number

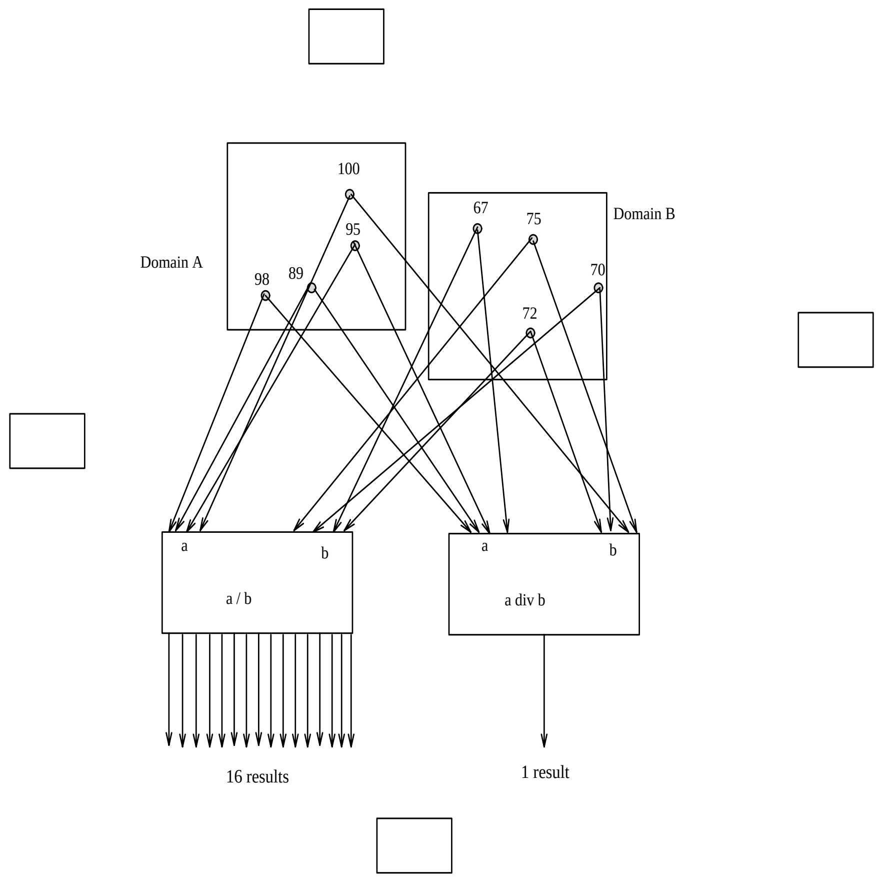

Figure 1: There are four potential values for variable *a* and four potential values for variable *b*, for a total of 16 pairs of potential inputs. Notice that for these 16 inputs, integer division always produces the same output (1), and real division produces 16 unique outputs.

representations of finite size.

Figure 1 illustrates the relationship between implicit information loss and the DRR. In Figure 1, we have 16 (a,b) input pairs that are presented to 2 functions: one performs real division, the other performs integer division. For the real division function there are 16 unique outputs, and for the integer division function there is one output. This example shows how the differences in the DRRs of these two forms of division are correlated to different amounts of information loss.

4.2 Explicit Information Loss: Harder to Find Early

Explicit information loss is not predicted by a DRR. Recall that explicit information loss is observed through static code inspection, whereas the potential for implicit information loss can be predicted from functional descriptions or code inspection. Explicit information loss may also be observable from a design document depending on its level of detail. Explicit information loss is more dependent on how the software is designed, and less dependent on the specification's (input, output) pairs.

	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	$f(a) = \sin(\text{a})$	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	$f(a, b) = \text{(a)or(b)}$	yes	$4 : 2$	a, b are boolean

Table 1: DRRs and implicit information loss of various functions; ∞_I is the cardinality of the integers, ∞_R is the cardinality of the reals.

4.3 Design Heuristics

We now present several strategies for reducing the detrimental effects that information loss has on testability.

4.3.1 Specification Decomposition: Isolating Implicit Information Loss

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 is *fixed* and cannot be modified without changing the specification itself, there are ways of decomposing a specification to reduce the potential of data state error cancellation occurring across modules. During specification decomposition, you have hands-on control of the DRR of each subfunction. With this, you gain an intuitive feeling (before a subfunction is implemented) for the degree of testing needed for a particular confidence that a module is propagating data state errors. The rule-of-thumb that guides this intuitive feeling is: "the greater the DRR, the more testing needed to overcome the potential for data state error cancellation."

During a design, a specification can be decomposed in a manner such that the program's modules are designed to either have a high DRR or a low DRR. By isolating modules that are more likely to propagate incoming data state errors through them during program testing (low DRR), testing and analysis resources can be shifted during module testing to modules that are less likely to propagate incoming data state errors across them.

4.3.2 Minimizing Variable Reuse: Reducing Implicit Information Loss

One method for decreasing implicit information loss is to minimize the reuse of variables. For instance, as we have already seen, 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 have lost the sign. Minimizing variable reuse is one attempt to decrease the amount of implicit information loss.

Minimizing variable reuse requires either creating more complex expressions or declaring more variables. If the number of variables is increased, memory requirements are also increased during execution. If complex expressions are used, we 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 testabilities. We advocate using *more* variables, and then making more variables available during testing.

4.3.3 Increasing Out-Parameters: Reducing Explicit Information Loss

Consider the analogy where modules are integrated circuits and local variables are internal signals in integrated circuits. This analogy allows us to see how explicit information loss caused by local variables parallels the notion of low observability in integrated circuits. Since explicit information loss suggests lower testabilities, we prefer, when possible, to lessen the amount of explicit information loss that occurs during testing. And if limiting the amount of explicit information loss is not possible, we at least have the benefit of knowing where the modules with greater data state error cancellation potential are before validation begins.

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 increasing the amount of output that these subspecifications return by treating local variables as out-parameters during testing. A third approach inserts self-tests (or "assertions") that are executed to check internal information during computation. Our research has suggested that assertions are particularly useful for testability analysis; not only can assertions be used to assure that a particular variable is correct or in range at some point during execution, but a failed assertion also suggests the possibility that previous computations (on which the variable definition depends) might be erroneous. In this approach, messages concerning incorrect internal computations are subsequently produced, so in terms of testability, the likelihood of fault hiding is reduced.

These three approaches both produce two important results:

1. 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.

2. The dimensionality of the range of the intended function is increased, which may increase the cardinality of the range, thus reducing information loss.

These two approaches simulate the idea previously mentioned that is used in integrated circuits—increasing the observability of internal signals [1]. In advocating these approaches, we are not repudiating the practice of information hiding during design. However, when writing software such as safety-critical software, there is a competing imperative: to enhance testability. Information that is not available during testing encourages undetected faults, and increased output discourages undetected faults. Perhaps an answer is to pattern software testing more closely on hardware testing: we should specify special output variables (pins) that are specified and implemented specifically and exclusively for testing.

A disadvantage to these approaches is that for the approaches to be beneficial, they all need additional specified information concerning the internal computations. Maybe the real message of this research is that until we make the effort to better specify what must occur, even at the intermediate computation level, testabilities and our assessed reliabilities will remain low. The unfortunate conclusion of our design-for-testability research is that *we must validate more internal information if we hope to increase software testability.* To validate more internal information, we must have some way of checking this additional internal information. This requires that more information be described in the specification and requirements phase. If we are not willing to specify these details at *some* point, we cannot expect to substantially improve reliability assessments.

5 Sensitivity Analysis

The previous section described how we can improve software testability. But a question remains: "How can we measure the increase in testability?" This section briefly presents a model for quantifying software testability that is described in more detail in [10].

Sensitivity analysis is based on separating software failure into three phases: execution of a software fault, creation of an incorrect data state, and propagation of this incorrect data state to a discernible output. This three part model of software failure is referred to as *PIE*, for Propagation, Infection, and Execution. There is a separate algorithm for each part of the *PIE* model: (E)xecution

Analyzer, (I)nfection Analyzer, and (P)ropagation Analyzer.[5]

In the rest of this section we give a brief outline of the three phases of sensitivity analysis. For more details, see [10]. To simplify explanations, we will describe each phase separately, but in a production analysis system, processing for the phases would overlap. As with the analysis of random testing, the accuracy of the sensitivity analysis depends in part on a good estimate of the input distribution that will drive the software when it is in use.

Before a fault can cause a failure, it must be executed. In this paper we will concentrate on faults that can be isolated at a single *location* in a program. A location can be defined as a single high level language statement, one machine code instruction, or some intermediate amount of computation. Our experiments thus far have defined a location as a piece of source code that can change the data state (including input and output files and the program counter). Thus an assignment statement and an **if** statement define a location, and a statement **read(a,b)** defines two locations. The probability of execution for each location is determined by repeated executions of the code with inputs selected at random from the input distribution. The execution analyzer estimates these execution probabilities.

If a location contains a fault, and if the location is executed, the data state of the execution may or may not be changed adversely by the fault. If the fault does change the data state into a data state that is incorrect for this input, we say the data state is *infected.* To estimate the probability of infection, the second phase of sensitivity analysis performs a series of syntactic mutations on each location. After each mutation, the program is re-executed with random inputs; each time the monitored location is executed, the data state is immediately compared with the data state of the original (unmutated) program at that same point in the execution. If the state differs, infection has taken place. The infection analyzer estimates the infection probability.

The third phase of the analysis estimates propagation. Again the location in question is monitored during random tests. After the location is executed, the resulting data state is changed 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 output that results from the changed data state is compared to the output that would have resulted without the change. If the outputs differ, propagation has occurred and a propagation probability can be estimated. The propagation analyzer estimates these propagation probabilities.

Sensitivity analysis is a fault simulation/injection-based method that relies on the two assumptions which are admittedly flawed: *single fault* and *simple fault.* The single fault assumption says that the program contains a single fault, not multiple faults distributed throughout the pro-

[5]The acronym is PIE because EIP doesn't spell anything. The authors regret any confusion this may cause.

gram. The simple fault assumption says that the fault exists in a single location, not distributed throughout the program. Without these assumptions, the combinatorics of simulating classes of distributed or multiple faults becomes intractable. Hence the fault classes that we simulate are artificially restricted. Despite this theoretical weakness, in practice the empirical techniques have yielded impressive experimental results (see [10]).

Each phase of sensitivity analysis produces a probability estimate based on the number of trials divided by the number of events that occurred (either execution, infection, or propagation). For a random test to reveal a fault, execution, infection, and propagation must occur to result in a failure. Thus the product of the mean of these estimates yields an estimate of the probability of failure that would result if this location had a fault. If we instead take the minimum over all three estimates and then obtain a product, we have a bound on the minimum probability of failure that would result if this location had a fault.

Sensitivity analysis is a new, empirical technique; pilot experiments in the early 1990s were done using hand coded syntactic mutations and only semi-automated data state mutations. 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. Since 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. A fully automated and commercialized sensitivity analysis tool, *PiSCES* 1.2 has been built and has been applied to systems as large as 100K SLOC. The tool can operate on larger systems, but to our knowledge has not. The value of sensitivity analysis is generally two-fold: (1) determining how much system level testing is needed to gain a confidence that faults are not hiding, and (2) for identifying those regions of the code of extreme low testability, where additional unit testing or other V&V resources should be applied. Additional benefits can be realized with slight modifications to the senstivity analysis algorithms. For example, we have elsewhere described how to modify the algorithms to increase fault-tolerance and improve software safety assessment. These and other applications are outside of the scope of this paper.

In summary, we contend that the results of experiments in sensitivity analysis are sufficient to motivate additional research and use with this technique; successful experiments have been shown in [10, 8]. We cannot guarantee that this new technique will make it possible to assess reliability to the precisions required for life-critical software. However, we do think it is premature to declare such an assessment impossible. The following sections argue that if software testability produces accurate predictions, then it will be possible to combine random black-box testing with sensitivity analysis to assess reliability more precisely than is possible with black-box testing alone.

6 The "Squeeze Play"

Both random black-box testing and sensitivity analysis gather information about possible probability of failure estimates for a program. However, the two techniques generate information in distinct ways: random testing treats the program as a single monolithic black-box but sensitivity analysis examines the source code location by location; random testing requires an oracle to determine correctness but sensitivity analysis requires no oracle because it does not judge correctness; random testing includes analysis of the possibility of no faults but sensitivity analysis focuses on the assumption that one fault exists. Thus, the two techniques give different kinds of predictions about the probability of failure.

Although the true probability of failure of a particular program (conditioned on an input distribution) is a single fixed value, this exact value is unknown to us. We therefore treat the probability of failure as a random variable Θ. We then use black-box random testing to estimate a probability density function (**pdf**) for Θ conditioned on an input distribution. We also predict a **pdf** for Θ using sensitivity analysis; this prediction is conditioned on the same input distribution as the testing **pdf**, but the **pdf** predicted using sensitivity analysis 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.[6] The assumption of this single, randomly located error is a variation on the competent programmer hypothesis.

Figures 2(A) and 2(B) show examples of two possible approximated Θ **pdf**'s. For each horizontal location θ, the height of the curve indicates the estimated probability that the true probability of failure of the program has value $< \theta$. The curve in Figure 2(A) is an example of an estimated **pdf** derived from random black-box testing; we assume that the testing has uncovered no failures. As we test more and more, we expect those 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, $\mathbf{Pr}(\theta = 1.0) = 0.0$. Details about deriving an estimated **pdf** for Θ given many random tests are given in [7].

The curve in Figure 2(B) is an example of an estimated **pdf** for Θ that might be derived from sensitivity analysis. Sensitivity analysis estimates at each location the probability of failure that would be induced in the program by a single fault at that location. This **pdf** is conditioned on the assumed input distribution, on the assumption that the program contains exactly one fault, and on the assumption that each location is equally likely to contain that fault.

We have marked interval estimates for each estimated **pdf**. If the interval marked between 0.0 and $\hat{\theta}$ includes 90% of the area under the estimated **pdf** in Figure 2(A), then according to

[6][10] formalizes the method employed to find a predicted probability of failure from sensitivity analysis.

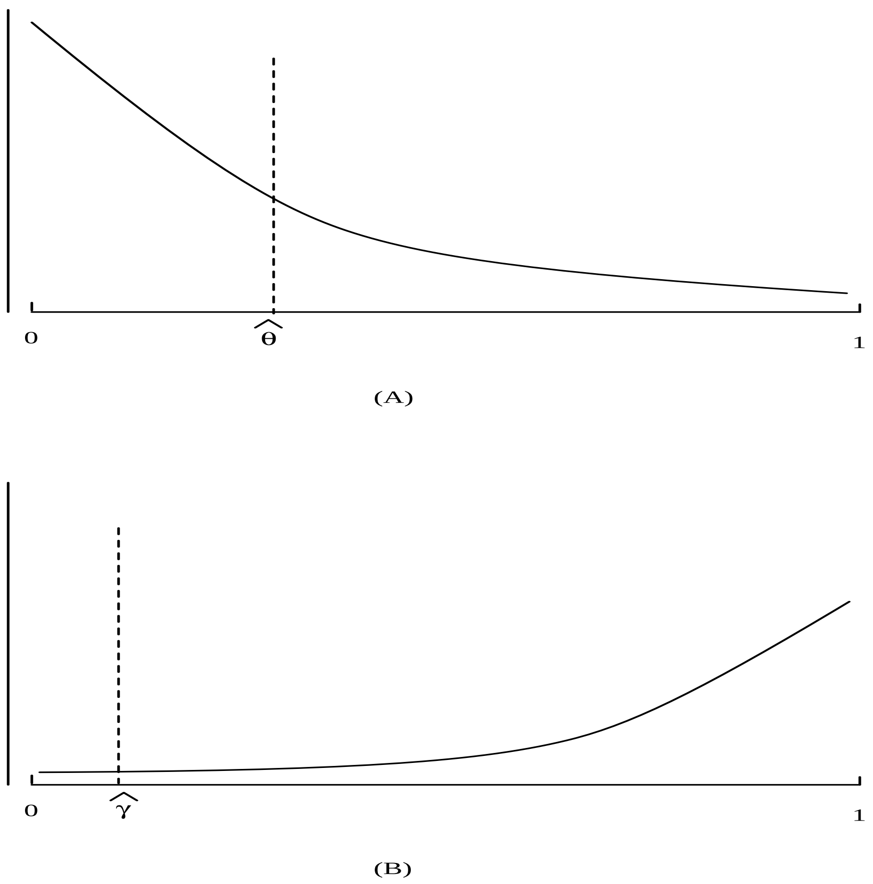

Figure 2: (A) The mean of the estimated **pdf** curve, $\hat{\theta}$, is an estimate of the probability of failure. (B) $\hat{\gamma}$ is an prediction of the minimum probability of failure using sensitivity analysis.

random testing the actual probability of failure is less than $\hat{\theta}$ with a confidence of 90%. Similarly, if the interval in Figure 2(B) includes 10% of the area under the estimated **pdf**, then according to sensitivity analysis, *if there exists a fault*, then it will imply a probability of failure that is greater than $\hat{\gamma}$ with confidence of 90%.

If there exists a fault and it induces a near-zero probability of failure, testing is unlikely to find that error. Locations that have sensitivity estimates very close to zero are troubling in an ultra-reliable application. However, a "fault" that induces a probability of failure (**pof**) 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 (i.e., $\theta = 0.0$), and ultra-reliability (or "correctness") has been achieved. Figure 2(A) suggests that if there is a fault, it is likely to induce a small probability of failure; Figure 2(B) suggests that *tiny* impact faults (meaning those that cause the program to fail with probabilities that are less that $\hat{\gamma}$) are unlikely. We now attempt to quantify the meaning of the two estimated **pdf**s taken together.

Hamlet has derived an equation to determine what he calls "probable correctness" [3]. 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 **pof**, $\hat{\theta}$ is some approximation of θ, and $0 < \hat{\theta} \leq 1$.[7] $1 - C$ 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 **pof** by the smallest fault in the program; then γ is the smallest possible non-zero **pof** for the program if all other independent faults were removed. If there are other faults, $\gamma < \theta$. γ is assumed to be unknown to us. 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 using T' test cases to find $\hat{\gamma}$, we have confidence C':

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

$\hat{\gamma}$ is obtained by subtracting a small fudge factor ϵ to the numerical estimate of γ found via sensitivity analysis using T' test cases. Given that actual *pofs* in the interval $(0,\hat{\gamma})$ are unlikely, our confidence that $\theta = 0.0$ is just:

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

where $\mathbf{Pr}(\gamma < \hat{\gamma})$ is the probability that we failed to correctly assess the minimum **pof** induced by any fault in our program from the fault classes that we simulated.[8] This probability 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. Realize that 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 that we did consider.

To better explain 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 **pof** induced by any of the faults it simulated. We will use $\hat{\gamma}$ as a reference point for establishing a confidence in this software. Trivially,

$$1 = Pr(\theta = 0) + Pr(0 < \theta < \hat{\gamma}) + Pr(\theta \geq \hat{\gamma}) \longrightarrow Pr(\theta = 0)$$

$$1 - Pr(0 < \theta < \hat{\gamma}) - Pr(\theta \geq \hat{\gamma})$$

[7]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.

[8]Assessing $\mathbf{Pr}(\gamma < \hat{\gamma})$ is out of the scope of this paper. To do so requires two additional probabilities: (1) the probability of an actual fault causing a lower impact to the actual failure probability than $\hat{\gamma}$; and (2) the probability that the order of magnitude of $\hat{\gamma}$ is too precise for the number of input values used in determining $\hat{\gamma}$, which is a statistical approximation error. The second of these problems is partially formalized in [9].

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

The goal of the "Squeeze Play" model is to push $\hat{\theta}$ towards $\hat{\gamma}$ in Figure 2 when T is fixed and confidence is high [4]. 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 that we have a program with three independent faults, with impacts to θ of: 0.0001, 0.00001, and 0.000001. In this situation, $\theta = 0.000111$, and $\gamma = 0.000001$. Suppose that $\hat{\gamma} = 0.000015$. In this situation, $\hat{\gamma} > \gamma$, but $\theta > \hat{\gamma}$. If C' is fixed close to 1.0 (meaning that the T' we will try is large enough with respect to $\hat{\gamma}$), then the likelihood that the program will fail at least once in T tests is also close to 1.0, and hence we are unlikely to be able to apply equation 1 since a failure should occur. Now suppose that we remove the faults with **pof**s of 0.0001 and 0.00001 after observing one or more failures, and when we reperform sensitivity analysis on the modified program, $\hat{\gamma}$ is still 0.000015.[9] Again we will test this code T' times in order to fix C' close to 1.0. Since $\theta < \hat{\gamma}$, $\mathbf{Pr}(\gamma < \hat{\gamma}) = 1.0$ and $\mathbf{Pr}(\theta = 0.0) = 0.0$. 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 of 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 0.0 where:

$$C'' \geq 1 - Pr(0 < \theta < \hat{\gamma}) - Pr(\theta > \hat{\gamma}) = 1 - [2e^{-2T'\epsilon^2} + (1 - \hat{\gamma})^T] \tag{4}$$

In this section we have shown one method for combining testability analysis with testing results to sharpen an estimate of the true **pof**. This use of testability measurement here is essentially a clean-up operation, a method of assessing whether or not 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 how expensive testing to different levels of confidence is given different degrees of testability.

7 Conclusions

Our research suggests that software testability clarifies a characteristic of programs that has been largely 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.

[9] In this program, we now have a single fault, so $\gamma = \theta = 0.000001$.

T	$\hat{\gamma}$	C'
7	0.1	0.5
44	0.1	0.99
458	0.01	0.99
298	0.01	0.95
460,514	10^{-5}	0.99
46,051,699	10^{-7}	0.99
693,147,181	10^{-9}	0.5
2,302,585,093	10^{-9}	0.9
2,995,732,273	10^{-9}	0.95

Table 2: Various Ts, $\hat{\gamma}$s, and C's.

We are particularly interested in designing software to increase its testability. Ideally, a design process begins with a (functional description, input distribution) pair that specifies the intended software. It may be that a theoretical upper bound exists on the testability that can be achieved for a given (functional description, input distribution) pair. If we can change the functional description to include more internal information, we should be able to increase that upper bound. Although the existence of an upper bound on testability is mentioned solely as conjecture, our research using sensitivity analysis and studying software's tendency to not reveal faults during testing suggests that such exists. We challenge software testing researchers to consider this conjecture.

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 without knowing if a particular fault exists in that software, and without reference to correctness. Since it does not rely on correctness, this characteristic, 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 [10]; research and practical application of this technique continues. Perhaps other more effective or more efficient testability measurement techniques will be discovered in the future, 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.

References

[1] Neil C. Berglund. Level-Sensitive Scan Design Tests Chips, Boards, System. *Electronics*, March 15 1979.

[2] R. Butler and G. Finelli. The infeasibility of experimental quantification of life-critical software reliability. *Proceedings of SIGSOFT '91: Software for Critical Systems* (December

4-6, 1991), New Orleans, LA., 66-76.

[3] Richard G. Hamlet. Probable Correctness Theory. *Information Processing Letters*, 25(1):17-25, April, 1987.

[4] R. Hamlet and J. Voas. Faults on Its Sleeve: Amplifying Software Reliability Assessment. In *Proc. of ACM SIGSOFT ISSTA'93.*, pages 89–98, Cambridge, MA, June 1993.

[5] Brian Marick. Two Experiments in Software Testing. Technical Report UIUCDCS-R-90-1644, University of Illinois at Urbana-Champaign, Department of Computer Science, November 1990.

[6] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability Measurement Prediction Application.* McGraw-Hill, 1987.

[7] K. Miller, L. Morell, R. Noonan, S. Park, D. Nicol, B. Murrill, and J. Voas. Estimating the Probability of Failure When Testing Reveals No Failures. *IEEE Trans. on Software Engineering*, 18(1):33–44, January 1992.

[8] J. Voas, K. Miller, and J. Payne. A Comparison of a Dynamic Software Testability Metric to Static Cyclomatic Complexity. In *Proc. of 2nd Int'l. Conf. on Software Quality Management*, Edinburgh, Scotland, July 1994.

[9] J. Voas, C. Michael, & K. Miller. Confidently Assessing a Zero Probability of Software Failure. In *Proc. of the 12th International Conf. on Computer Safety, Reliability, and Security.* October 27–29, 1993, Poznan, Poland.

[10] J. Voas. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, 18(8):717-727, August 1992.