

Improving the Software Development Process Using Testability Research

Jeffrey M. Voas
Reliable Software Technologies Corp.
Penthouse Suite
1001 North Highland Blvd.
Arlington, VA 22201

Keith W. Miller
Department of Computer Science
101 Jones Hall
College of William & Mary
Williamsburg, VA 23185

Abstract

Software testability is the tendency of code to reveal existing faults during random testing. This paper proposes to take software testability predictions into account throughout the development process. These predictions can be made from formal specifications, design documents, and the code itself. The insight provided by software testability is valuable during design, coding, testing, and quality assurance. We further believe that software testability analysis can play a crucial role in quantifying the likelihood that faults are not hiding after testing does not result in any failures for the current version.

1 Introduction

Software development processes typically focus on avoiding errors, detecting and correcting software faults that do occur, and predicting reliability after development. In this paper we emphasize the third focus, the analysis of software to determine its reliability.

We contend that *software testability*, the tendency of code to reveal its faults during random testing, is a significant factor in software quality. As such, we propose to take testability into account throughout the development process. The insight provided by software testability is valuable during design, coding, testing, and quality assurance.

In the first section below we give an overview of software testability and of a model used to analyze code for testability. In the following four sections we describe how testability can be taken into account during design, coding, testing, and quality assurance. The final section summarizes and describes areas for future research.

2 Preliminaries

Software testability analysis is a process for measuring the “value” provided by a particular software testing scheme, where the value of a scheme can be assessed in different ways. For instance, software testability has sometimes been assessed via the ease with which inputs can be selected to satisfy some structural testing criteria, e.g., *branch coverage*. With this view of testability, if it turned out to be an extremely difficult task to find inputs that satisfied a particular structural coverage criteria of a program, then the testability of the program would be reduced. A different view of software testability defines it to be a prediction of the probability that existing faults will be revealed during testing given an arbitrary input selection criteria C [17]. Here, software testability is not purely regarded as an assessment of the difficulty to select inputs to cover software structure, but instead as a way of predicting whether a program would reveal existing faults during testing when C is the method for generating inputs.

To compare these two view points, we must first understand the underlying assumption on which the first definition operates. It implicitly assumes that the more software structure exercised during testing, the greater the likelihood that existing faults will be revealed. With this definition, a straight-line program without any conditional expressions or loops would be assigned a higher software testability than any program with a more complex flow of control. However when testability is based on the probability of fault detection, a straight-line program without any conditional expressions or loops could potentially be assigned a lower testability than a program with more complex flow of control. This would not occur with the coverage-based definition. This is because there are conditions other than coverage that can determine

whether or not software will fail during testing. The advantage of our definition is that it incorporates other factors than coverage that play an important role in whether faults will hide during testing. These factors will be described later.

In either definition, software testability analysis is a function of a (program, input selection criteria) pair. The means by which inputs are selected is a parameter of the testing strategy: inputs can be selected randomly, they can be based upon the structure of the program, or they may be based on the tester’s human intuition. Testability analysis is impacted heavily by the choice of input selection criteria. Testability analysis is more than an assertion about a program, but rather is an assertion about the ability of an input selection criteria (in combination with the program) to satisfy a particular testing goal. Programs may have varying testabilities when presented with varying means of generating inputs.

From this point on, we will only discuss the latter definition of software testability, which is based on the probability of tests uncovering faults. Furthermore, we will concentrate on black-box random testing as the type of testing used to establish the testability of a program.

In order for software to be assessed as having a “greater” testability by this definition, it must be likely that a failure occurs during testing whenever a fault exists. To understand this likelihood, it is necessary to understand the sequence of events that leads up to a software failure. (By software failure, we mean an incorrect output that was caused by a flaw in the program, not an incorrect output caused by a problem with the environment in which the program is executing.) Software failure only occurs when the following three necessary and sufficient conditions occur in the following sequence:

1. A input must cause a fault to be *executed*.
2. Once the fault is executed, the succeeding data state must contain a *data state error*.
3. Once the data state error is created, the data state error must *propagate* to an output state.

This model is termed the fault/failure model, and it’s origins in the literature can be traced to [9, 12]. The fault/failure model relates program inputs, faults, data state errors, and failures. Since faults trigger data state errors that trigger failures, any formal testability analysis model that uses the second definition for software testability should take into account these three conditions. ([2] is an example of a mutation-based testing methodology that considers the first two

conditions.) It is the second and third conditions that the second definition of testability takes into account that the first definition does not. This is the essential difference.

A semantic-based definition of testability predicts the probability that tests will uncover faults if any faults exist. The software is said to have high testability for a set of tests if the tests are likely to uncover any faults that exist; the software has low testability for those tests if the tests are unlikely to uncover any faults that exist. Since it is a probability, testability is bounded in a closed interval [0,1].

In order to make a prediction about the probability that existing faults will be revealed during testing, formal testability analysis should be able to predict whether a fault will be executed, whether it will *infect* the succeeding data state creating a data state error, and whether the data state error will propagate its incorrectness into an output variable. When an existing data state error does not propagate into any output variable, we say that the data state error was *cancelled*. When all of the data state errors that are created during an execution are cancelled, the existence of the fault that triggered the data state errors remains hidden, resulting in a lower software testability. These conditions provide a formal means for predicting the testability of software that is tightly coupled to the fault/failure model of computation.

PIE [17, 14, 15] is a formal model that can be used to assess software testability that is based on the fault/failure model. *PIE* is based on three subprocesses, each of which is responsible for estimating one condition of the fault/failure model: *Execution Analysis (EA)* estimates the probability that a location¹ is executed according to a particular input distribution; *Infection Analysis (IA)* estimates the probability that a syntactic mutant affects a data state; and *Propagation Analysis (PA)* estimates the probability that a data state that has been changed affects the program output after execution is resumed on the changed data state.

PIE makes predictions concerning future program behavior by estimating the effect that (1) an input distribution, (2) syntactic mutants, and (3) changed data values in data states have on current program behavior. More specifically, the technique first observes the behavior of the program when (1) the program is executed with a particular input distribution, (2) a location of the program is injected with syntactic mutants,

¹A location in *PIE* analysis is based on what Korel [6] terms a single instruction: an assignment, input statement, output statement, and the <condition> part of an **if** or **while** statement.

and (3) a data state (that is created dynamically by a program location for some input) has one of its data values altered and execution is resumed. After observing the behavior of the program under these scenarios, the technique then predicts future program behavior if faults were to exist. These three scenarios simulate the three necessary and sufficient conditions for software failure to occur: (1) a fault must be executed, (2) a data state error must be created, and (3) the data state error must propagate to the output. Therefore the technique is based firmly on the conditions necessary for software failure.

The process for predicting the probability that a location is executed follows: the program is instrumented to record when a particular location is executed via a print command that is added into the source code and then compiled. The instrumented program is then run some number of times with inputs selected at random according to the input distribution of the program. The proportion of inputs that cause the print command to be invoked in the instrumented program out of the total number of inputs on which the instrumented program is executed is an estimate of this probability. This probability estimate along with others for the software can then be used to predict the software's testability.

The process for predicting the probability that a fault in a location will affect the data state of the program will now be provided. This process is repeated several times for each location: a syntactic mutation is made to the location in question. The program with this mutated location is then run some number of times with inputs selected at random according to the program's input distribution. For all the times the mutated location is executed, we record the proportion of times that the program with the mutated location produces a different data state than the original location; this proportion is our estimate of the probability that a fault at this location infects. For example, suppose that a program is executed 10 times, and during the 10 executions the original location is executed 1000 times, and 345 data states produced by the mutated program are different than what the original "unmutated" location produces, then our probability estimate is 0.345 with an associated confidence interval. In general, many different syntactic mutants are made for a single location, each yielding a probability estimate in this manner. These probability estimates for this location along with those for other locations in the software can then be used to predict the software's testability.

The process for predicting the probability that a

data state error will cause program failure given that a location creates a data state error follows. This process is repeated several times (over a set of program inputs) for each location: The program is executed with an input selected at random from the input distribution. Program execution is halted just after executing the location, a randomly generated data value is injected into some variable, and program execution is resumed. If the location is in a loop, we customarily inject another randomly selected value into the same variable on each successive iteration. Specific details on how this process is performed are found in [14]. This process simulates the creation of a data state error during execution. We term this process "perturbing" a data state, since the value of a variable at some point during execution represents a portion of a data state. The tool then observes any subsequent propagation of the perturbed data state to successor output states after execution is resumed. This process is repeated a fixed number of times, with each perturbed data state affecting the same variable at the same point in execution. For instance, assume that after performing this process on some variable 10 times the output is affected 3 of those times. Then the resulting probability estimate would be 0.3 with some confidence interval [7]. This process is performed using different variables as the recipients of the perturbed data states. Probability estimates found using the perturbed data states can be used to predict which regions of a program are likely and which regions are unlikely to propagate data state errors caused by genuine software faults. These probability estimates for this location along with those for other locations in the software can then be used to predict the software's testability.

PISCES is a tool developed in C++ that implements the *PIE* technique for software written in C. The building of *PISCES* has occurred in stages over the past several years. The first commercial version of *PISCES* is hoped to be completed by September '92. This version will incorporate all the nuances of the theoretical model. The funding available to us will determine the scheduling of this project. The *PISCES* program and design were written by Jeffery Payne of RST Corp.

Another testability model that can sometimes be quantified via the code or specification is termed the domain/range ratio (DRR). This model differs from *PIE* in that it is static instead of dynamic. Also, another difference is that *PIE* is a function of the probability density function over the domain of the program, whereas the DRR metric is independent of the prob-

ability density function. The *domain/range ratio* of a specification is the ratio between the cardinality of the domain of the specification to 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 visible from a specification. (An in-depth definition of the DRR metric can be found in [16].) After all, there are specifications whose ranges are not known until programs are written to implement the specifications. If a program does not correctly implement a specification, then the program's DRR may not match the specification's DRR. This is demonstrated in [16].

DRRs roughly predict a degree of software's testability. Generally as the DRR increases for a specification, the potential for fewer data state errors affecting software's output occurring within the implementation increases. When α is greater than β , research using *PISCES* has suggested that faults are more likely to remain undetected (if any exist) during testing than when $\alpha = \beta$.

3 Testability and Design

Although software testability is most obviously relevant during testing, by paying attention to testability early in the development process, the testing phase can potentially be improved significantly. Already at the design phase, testability can be enhanced.

During design, more general specifications are elaborated and decomposed. Decomposition eventually results in functional descriptions of separate code modules. As these module descriptions are defined, the developer can adjust the decomposition to improve the eventual testability when the modules are implemented.

The key to predicting testability already at the design phase is DRR, the domain/range ratio described above. When the inputs and outputs of a module design are specified, the designer should be able to give a fairly accurate assessment of the DRR of that module. A module design should already include a precise definition of all the legal inputs and outputs that should result, and these legal definitions form the basis of a DRR estimate. However, not all *legal* inputs (outputs) are *possible* inputs (outputs) when the module is integrated into the entire system. If the designer can give a more precise description of the possible inputs (outputs), these can form the basis of a better DRR estimate.

Once a DRR is estimated for each module design, the designer can identify modules whose high DRR indicates that the module will tend to hide faults from random testing. In most applications such modules are inevitable: when data are distilled, a high DRR results. However, the designer can take care to isolate high DRR functionality in as few modules as possible, and to make high DRR modules as simple as possible. Since random testing is an ineffective method for assuring the quality of high DRR modules, implementors and quality assurance personnel will have to use other methods to assess these modules. These other methods (such as path testing strategies[18], proofs of correctness[4], and when possible exhaustive testing) are particularly difficult for large, complex modules. By isolating high DRR operations in small, straightforward modules the designer can facilitate efficient analysis later in the development process.

Some operations outlined with a high DRR in a specification can be designed to have a higher DRR in the implementation. This is accomplished by having a module return *more* of its internal data state to its users. This advice flies in the face of the common wisdom that a module should as much as possible hides its internal workings from other modules[11]. We agree that such hiding can enhance portability and reduce interface errors; however, there is a competing interest here: increasing testability. In order to increase the testability of a module, it should reveal as much of its internal state as is practical, since information in these states may reveal a fault that will otherwise be missed during testing. Therefore the designer should, especially for modules that will otherwise have a high DRR, try to design an interface that includes enough state information to increase testability to an acceptable level.

4 Testability, Coding, and Unit Test

When designs are implemented, the DRR again provides direction for development that enhances software testability. At the design stage, the process focuses on the DRR of modules; at the coding stage, the focus shifts to individual code locations. Single operations can induce a high DRR; for example, $a \bmod b$, where $a \gg b$, is a high DRR operation. The programmer should take special care when programming these locations. This care should include increased attention during code inspections, small proofs of correctness for the block of code in which the locations arise, and increased white box testing at boundaries and special values of the operation itself. As before,

when random black-box testing is unlikely to uncover faults, the programmer must use alternative methods to assure quality.

Some locations with a high DRR are obvious from the operation. However, more subtle high DRR code can arise from the interaction of several different locations, perhaps separated by many intervening locations. Furthermore, a location or locations that would not necessarily have a high DRR under all input distributions may have a high DRR under particular input distributions. For these reasons, visual inspections are inadequate to identify all potential high DRR code locations during coding and unit testing. The *PISCES* software tool, described above, gives automated “advice” on the testability of code locations. Given an input distribution, *PISCES* runs a variety of experiments that yield a testability estimate for each relevant location in a module. Because *PISCES* testability analysis is completely automated, machine resources can be used in place of human time in trying to find locations with low testability. Because *PISCES* execution times are quadratic in the number of locations, this analysis can be accomplished with much more thoroughness at the module level than during system test ($a^2 + b^2 \leq (a + b)^2$).

5 Testability, System Test, and Reliability Assessment

During system test, the attention may shift radically from the most abstract view to an intensely concrete focus, depending on the outcome of system tests. As long as system tests uncover no software faults, the quality assurance effort concentrates on assessing the overall quality of the delivered product. However, when a system test does not deliver the required behavior, the development staff must locate and repair the underlying fault. Testability analysis can add information that is useful both for assessing the overall quality and for locating software bugs.

Debugging software is easiest when a fault causes software to fail often during testing; each failure furnishes new information about the fault. This information (hopefully) helps locate the fault so that it can be repaired. The most difficult faults are those that only rarely cause the software to fail. These faults provide very few clues as to their nature and location. When a software system has been analyzed for testability using *PISCES*, each location has a testability estimate; according to that estimate, if a fault exists at that location, it is likely to cause a failure rate

close to that testability estimate. When the debugging process begins to converge to a deliverable product, it may exhibit a very low but non-zero failure rate. When seeking the location of a fault that could cause this “low impact,” the developer can use the *PISCES* testability scores to identify likely candidates among the code locations being tested. In several preliminary experiments (described in [13]), testability scores were highly correlated with faults at selected locations.

The importance of testability during reliability assessment concerns the confidence of testers that they have found all the faults that exist. In the past, quantifying that confidence had to rely exclusively on random testing. The more testing, the higher the confidence that the latest version was fault-free. However, as an increasing number of tests revealed no failures, the predicted reliability goes up proportional to $1/T$ [8]. Especially when software requires high reliability (such as flight software, medical devices, and other life-critical applications), random testing soon becomes intractable as the exclusive source of information about software quality.

However, testability analysis may allow developers to obtain much higher confidence in a program using the same amount of testing. The argument is as follows: in traditional random testing, probability determines that large-impact errors are likely to be discovered early in testing, and smaller and smaller impact errors are the only type to survive undetected as the testing continues. It is the potential “tiny” faults that prohibit us from gaining higher confidence at a more rapid rate as testing continues.

But testability analysis offers a new source of information about the likelihood of such tiny faults existing. If we can write programs with high testability, then we can empirically demonstrate that tiny faults are unlikely to exist. This quantifiable confidence can add to our confidence that testing has uncovered all existing faults (which are unlikely to be high-impact). In essence, we put a “squeeze play” on errors: we design and implement code that is unlikely to hide small faults, and then we test to gain confidence that larger faults are unlikely to have survived testing.

This technique is still experimental; we have not yet determined that industrial programs can be written with sufficiently high testability to make the squeeze play effective. However, we think that if testability is a concern throughout the development process, highly testable code *can* be produced, specifically for the purpose of passing strict requirements for high reliability. Such code would have to be designed for relatively simple functions and straightforward code. Interestingly,

(for somewhat different reasons) others have suggested that this kind of code may be the wave of the future for critical software [10].

5.1 Applying PIE to Probability of Failure Estimation

Both random black-box testing and *PIE* gather information about possible probability of failure values for a program. However, the two techniques generate information in distinct ways: random testing treats the program as a single monolithic black-box but *PIE* examines the source code location by location; random testing requires an oracle to determine correctness but *PIE* requires no oracle because it does not judge correctness; random testing includes analysis of the possibility of no faults but *PIE* focuses on the assumption that one fault exists. Thus, the two techniques give independent 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 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 estimate a **pdf** for Θ using the result of *PIE*; this estimate is conditioned on the same input distribution as the testing **pdf**, but the **pdf** estimated using the results of *PIE* 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. The assumption of this single, randomly located error is a variation on the competent programmer hypothesis [1].

Figures 1(A) and 1(B) show examples of two possible estimated 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 θ . The curve in Figure 1(A) is an example of an estimated **pdf** derived from random black-box testing; we assume that the testing has uncovered no failures. Details about deriving an estimated **pdf** for Θ given many random tests are given in [8].

The curve in Figure 1(B) is an example of an estimated **pdf** for Θ that might be derived from *PIE*'s results. *PIE* can be used to estimate at each location the probability of failure that would be induced in the program by a single fault at that location. All these estimates are gathered into a histogram, one entry for each location estimate. The histogram is then smoothed and normalized to produce an estimated

pdf. 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 by $\hat{\theta}$ includes 90% of the area under the estimated **pdf** in Figure 1(A), then according to random testing the actual probability of failure is somewhere to the left of $\hat{\theta}$ with a confidence of 90%. Similarly, if the interval in Figure 1(B) includes 10% of the area under the estimated **pdf**, then according to *PIE if there exists a fault*, then it will induce a probability of failure that is somewhere to the *right* of $\hat{\gamma}$ with confidence of 90%.

The probability of failure of 0 is a special case that complicates the interpretation of the **pdf** estimated by the results of *PIE*. If there exists a fault and it induces a near-zero probability of failure, testing is unlikely to find that error. Locations that have *PIE* estimates very close to zero are troubling in an ultra-reliable application. However, a fault that induces a **pdf** of 0 is not technically 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., $\theta = 0$), and ultra-reliability has been achieved. We do not expect this to be the case in realistic software, but our analysis cannot rule out its possibility.

Figure 1(A) suggests that if there is a fault, it is likely to induce a small probability of failure; Figure 1(B) suggests that such small impact faults 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” [5]. When T tests have been executed and no failures have occurred, then:

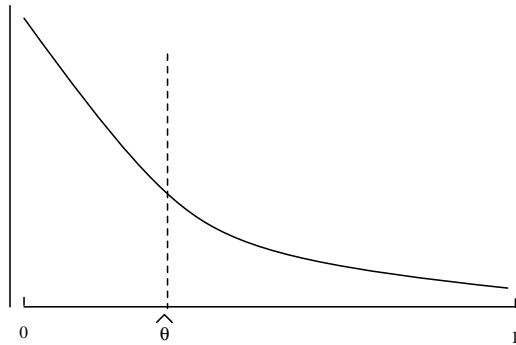
$$C = \text{Prob}(\theta \leq \gamma) = 1 - (1 - \gamma)^T \quad (1)$$

where C is probable correctness, θ is the true **pdf**, and $0 < \gamma \leq 1$.²

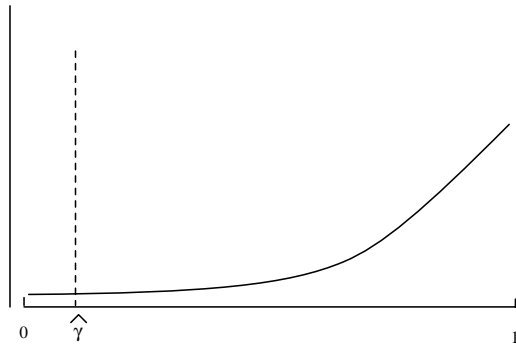
Hamlet’s equation is related to the **pdf** estimated by testing in Figure 1(A) as follows: for any given γ , $C = \int_0^\gamma t(\theta)d\theta$, where $t(\theta)$ is the value of the testing **pdf** at θ . This equation requires a large number of tests, T , to establish a reasonably high C for a γ close to 0.

It is possible via *PIE* to predict a minimum probability of failure that would be induced by a fault at a location in the program. In Figure 1(B) we have labeled a particular value $\hat{\gamma}$; using the **pdf** estimated by

²Hamlet calls C a measure of probable correctness, but it would be called a confidence if the equations were cast in a traditional hypothesis test.



(A)



(B)

Figure 1: (A) The mean of the estimated **pdf** curve, $\hat{\theta}$, is an estimate of the probability of failure. (B) $\hat{\gamma}$ is an estimate of the minimum probability of failure using *PIE*'s results.

PIE's results, we calculate $\alpha = \int_{\hat{\gamma}}^1 s(\theta)d\theta$, where $s(\theta)$ gives the value of the *PIE pdf* at θ . α is the probability according to *PIE* that the true **pdf** is greater than $\hat{\gamma}$. We will refer to α as our confidence that $\hat{\gamma}$ is the true minimum failure rate for the program. If *PIE*'s results have predicted $\hat{\gamma}$ as the minimum **pdf** and if we have confidence α that it is the minimum, then we can make the following conjecture:

$$\text{if } \text{Prob}(\theta \leq \hat{\gamma}) = 1 - (1 - \hat{\gamma})^T$$

and if $((\theta = 0) \text{ or } (\theta > \hat{\gamma}))$ with confidence α ,

$$\text{then with confidence } \alpha, \text{Prob}(\theta = 0) = 1 - (1 - \hat{\gamma})^T. \quad (2)$$

This prediction of the $\text{Prob}(\theta = 0)$ is higher than is possible from T random black-box tests without the results of *PIE*.

6 Summary and Future Research

The significance of testability is only recently becoming recognized in the software engineering community [3]. In this paper we have illustrated how testability with respect to random black-box testing has importance throughout the software development life-cycle. Automated testability analysis, such as *PISCES*, exploits relatively inexpensive CPU power to help guide design, coding, and testing. Also, static analysis of the DRR gives insight early in the specification and design stages. In all these applications, testability gives a new perspective on the relationship between software quality and our ability to measure that quality.

Future research will focus on expanding the capabilities of the *PISCES* tool, empirically exploring different syntactic and semantic mutations for testability analysis, and comparing testability using different testing strategies. We expect that semantic-based statistical analysis of this sort will become increasingly

important as computer power becomes increasingly affordable and software quality in life-critical software becomes an increasing concern.

Acknowledgements

This work has been funded by a National Research Council NASA-Langley Resident Research Associateship and NASA Grant NAG-1-884. Since collaborating on this paper at NASA-Langley Research Center, Voas has accepted the position of Vice President of Advanced Research at Reliable Software Technologies Corporation in Arlington, VA.

References

- [1] RICHARD A. DEMILLO, RICHARD J. LIPTON, AND FREDERICK G. SAYWARD. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [2] RICHARD A. DEMILLO AND A. J. OFFUTT. Constraint-Based Automatic Test Data Generation. *IEEE Trans. on Software Engineering*, 17(9):900–910, September 1991.
- [3] R. S. FREEDMAN. Testability of Software Components. *IEEE Transactions on Software Engineering*, SE-17(6):553–564, June 1991.
- [4] D. GRIES. *The Science of Programming*. Springer-Verlag, 1981.
- [5] RICHARD G. HAMLET. Probable Correctness Theory. *Information Processing Letters*, pages 17–25, April 1987.
- [6] BODGAN KOREL. PELAS-Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, SE-14(9), September 1988.
- [7] AVERILL M. LAW AND W. DAVID KELTON. *Simulation Modeling and Analysis*. McGraw-Hill Book Company, 1982.
- [8] 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.
- [9] LARRY JOE MORELL. A Theory of Error-based Testing. Technical Report TR-1395, University of Maryland, Department of Computer Science, April 1984.
- [10] J. D. MUSA. Reduced Operation Software. *Software Engineering Notes*, July 1991.
- [11] DAVID L. PARNAS. Designing software for ease of extension and contraction. *IEEE Trans. on Software Engineering*, SE-5:128–138, March 1979.
- [12] D. RICHARDSON AND M. THOMAS. The RELAY Model of Error Detection and its Application. *Proceedings of the ACM SIGSOFT/IEEE 2nd Workshop on Software Testing, Analysis, and Verification*, July 1988. Banff, Canada.
- [13] J. VOAS AND K. MILLER. Applying A Dynamic Testability Technique To Debugging Certain Classes of Software Faults, *Software Quality J.*, To appear.
- [14] J. VOAS. *A Dynamic Failure Model for Performing Propagation and Infection Analysis on Computer Programs*. PhD thesis, College of William and Mary in Virginia, March 1990.
- [15] J. VOAS. A Dynamic Failure Model for Estimating the Impact that a Program Location has on the Program. In *Lecture Notes in Computer Science: Proc. of the 3rd European Software Engineering Conf.*, volume 550, pages 308–331, Milan, Italy, October 1991. Springer-Verlag.
- [16] J. VOAS. Factors That Affect Program Testabilities. In *Proc. of the 9th Pacific Northwest Software Quality Conf.*, pages 235–247, Portland, OR, October 1991. Pacific Northwest Software Quality Conference, Inc., Beaverton, OR.
- [17] J. VOAS. *PIE: A Dynamic Failure-Based Technique*. *IEEE Trans. on Software Engineering*, 18(8), August 1992.
- [18] ELAINE J. WEYUKER. An Empirical Study of the Complexity of Data Flow Testing. *Proc. of the Second Workshop on Software Testing, Validation, and Analysis*, pages 188–195, July 1988. Banff, Alberta.