

# Substituting Voas's Testability Measure for Musa's Fault Exposure Ratio

Jeffrey M. Voas  
Reliable Software Technologies  
21515 Ridgetop Circle, Suite 250  
Sterling, VA 20166  
(703) 404-9293  
jmvoas@RSTcorp.com

Keith W. Miller  
Department of Computer Science  
University of Illinois at Springfield  
Springfield, IL 62794  
(217) 786-7327  
miller@eagle.sangamon.edu

## Abstract

This paper analyzes the relationship between Voas's *software testability* measure and Musa's *fault exposure ratio*,  $K$ . It has come to our attention that industrial users of Musa's model employ his published, experimental value for  $K$ ,  $4.2 \times 10^{-7}$ , for their projects, instead of creating their own  $K$  estimate for the system whose reliability is being assessed. We provide a theoretical argument for how a slight modification to Voas's formulae for determining a predicted *minimal* fault size can provide a predicted *average* fault size. The predicted average fault size can be used as a substitute for  $4.2 \times 10^{-7}$ , and in our opinion is a more plausible choice for  $K$ .

## 1 Introduction

For software systems, the inability to guarantee failure-free operation without either exhaustive testing or performing a formal proof-of-correctness is troubling. Dijkstra [1] showed that exhaustive testing was generally infeasible by illustrating that exhaustively testing a program that accepts 2 32-bit integers would be completed in approximately 585,000 years (assuming non-stop testing was performed at a rate of 1,000,000 successful test executions per second). Since this is infeasible, it becomes apparent that if testing is to be effective at all, then non-exhaustive testing must be performed in a manner that offers some degree of confidence that the code is reliable.

Software that has not failed during non-exhaustive software testing is both boon and curse. Such software is boon for the manufacturer since no defects are known to exist; that same software may be a curse for the users, who run the risk of defects that are as yet unknown.

The problem with software that has not failed during testing is a manifestation of Dijkstra's observation [1]: non-exhaustive software testing can reveal the existence of errors but cannot reveal the absolute absence of errors. A piece of software that has not failed during testing (when testing has used a "fixed" and non-exhaustive set of inputs) does not

guarantee that a failure cannot occur for inputs not in the test set. Although successful testing provides a confidence that the software will never fail, it only provides a confidence – it does not guarantee it.

In [5], Musa et al. recognized that faults can fail to be exposed on a particular execution. Musa et al. provided a parameter termed the *fault exposure ratio*,  $K$ , that represents the proportion of program executions for which executing a fault will result in a failure. This parameter is particularly useful in determining how often you can expect to see failure (and thus reliability). Such a parameter can also be useful in determining a confidence that your code is not hiding faults, provided that the parameter is quantified with respect to the system whose reliability is being estimated.

We have learned that industrial users of Musa's model, particularly in the telecommunication's industry, employ the published, experimental value for  $K$  in [5],  $4.2 \times 10^{-7}$ , for their projects, as opposed to measuring it for the system whose reliability is being assessed. Hence, their choice for  $K$  is totally application *independent*, a constant value determined empirically on a set of software that could be dramatically different from their current application.

Voas's testability measurement estimates a *minimal* fault size [6]. In this paper, we will provide a theoretical argument for how a slight modification to Voas's formulae can provide a predicted *average* fault size. The predicted average fault size can be used as a substitute for  $K = 4.2 \times 10^{-7}$ , and in our opinion is a more plausible choice for this parameter. (To illustrate the power of larger  $K$  values, consider  $K = 1.0$ ; in this case, any fault in the code should cause failure immediately, which suggests that if you run the program several times and it doesn't fail, either your choice for  $K$  was incorrect, or your code is likely to be correct.) The rest of this paper argues that software testability techniques provide a theoretically justifiable, empirically measured, practical method for estimating an application *dependent* measure of  $K$ .

The remainder of this paper is organized as follows: Section 2 introduces the software testability technique (termed sensitivity analysis) that we believe is applicable to finding

$K$ . Section 3 describes the  $K$  parameter, and Section 4 describes how we can adapt sensitivity analysis to produce an application dependent measure of  $K$ .

## 2 Voas’s Software Testability Model

Until recently, the term “software testability” had been considered to be a measure of the ability to select inputs that satisfy certain structural testing criteria, e.g., the ability to satisfy various code-based testing coverages. In the early 90’s Voas [6, 2] redefined software testability to be:

a prediction of the probability of software failure occurring if the software were to contain a fault, given that software execution is with respect to a particular input distribution.

The motivation for this redefinition was to minimize the risk of performing too little testing and becoming excessively confident in the absence of faults.

Voas’s definition says that the testability of software is strictly related to the ability of the software to hide faults during testing when the inputs are selected according to some input selection scheme  $D$ . This definition indirectly addresses the amount of textual coverage achieved during testing. If large regions of a program are infrequently executed according to  $D$ , both of the older definition and Voas’s definition of software testability would converge, producing a lower software testability prediction.

Musa’s  $K$  parameter accounts for the possibility that code is infrequently executed. However Voas’s definition of testability does not go the next step and assert that the ability to easily execute all code regions implies that faults are not hiding; it considers other factors before such a strong assertion can be justified. Testability is more than just code coverage because executing a statement (or a path) once does not mean that a fault in that statement (or on that path) has been revealed; the matter is more complex.

Using Voas’s definition, if after testing according to input selection technique  $D$  and observing no failures (with the *a priori* knowledge that the testability of the software was high), we assert with high confidence that faults are not hiding. However if after testing according to input selection technique  $D$ , we observe no failures but find that the testability of the software is low, we gain less confidence that faults are not hiding. This view of software testability provides a way of quantifying the risk associated with critical software systems that have demonstrated successful testing.

“*Sensitivity analysis*” [6, 2] is the practical implementation of Voas’s definition. Sensitivity analysis can be slightly modified and used as an application dependent  $K$ .<sup>1</sup>

---

<sup>1</sup>Sensitivity analysis requires no oracle nor specification, however it does require that inputs are selected at random consistent with an assumed input distribution, which we denote  $D$ . Preferably  $D$  will be the *operational distribution*, but there are certain instances where a *uniform* distribution may be substituted.

Sensitivity analysis allows testers to determine when testing can stop with an acceptable confidence that no faults are hiding. This is important information when it is known that a particular component is likely to be reused in differing environments. Sensitivity analysis also predicts where remaining faults have the best chance of hiding from tests. This allows testing resources to be directed to those regions of the code that have shown the greatest potential of hiding faults.<sup>2</sup> This is in the same spirit as Musa’s  $K$ , with the important distinction that  $K$  is determined for “real” faults after they have been discovered, and sensitivity analysis is determined for “hypothesized” faults. The beauty of using hypothesized faults is that you can perform the analysis without knowing what the real faults are. This ignorance of the remaining faults is, of course, precisely the situation that exists when testing reveals no failures, the time when the software is potentially ready for release.

Sensitivity analysis makes predictions concerning future program behavior by using estimates of the effect that (1) an input distribution,  $D$ , (2) syntactic mutants, and (3) changed data values in data states have on current program behavior. A formal description of sensitivity analysis is provided in [6, 2]. It is these syntactic mutants and changed data state values that constitute the “hypothesized” faults. Sensitivity Analysis is a predictive technique that uses the results of a estimation technique, PIE (Propagation, Infection, and Execution), that is composed of three different analyses: (1) Execution analysis, (2) Infection analysis, and (3) Propagation analysis.

**Execution Analysis** is the process of estimating the likelihood that a test case exercises some statement,  $l$ .

**Infection Analysis** is the process of estimating the likelihood for whether syntactic mutants are able to create corrupted internal states with respect to the testing scheme.

**Propagation analysis** is the process of estimating the likelihood for whether infected program states will result in corrupted output, i.e., a statistical measure of *observability*.

After observations of the behavior of the program from these three analyses are completed, sensitivity analysis predicts the smallest sized fault that could be hiding in the code from *existing* faults; this is sometimes termed the *latent failure rate* or the *minimum predicted failure probability*.

### 2.1 The PIE Technique

PIE is a technique that is based on the three necessary and sufficient conditions for software to fail if the program contains a fault. These conditions form what is called the *fault/failure* model [4]:

---

<sup>2</sup>Sensitivity analysis gives a road map into the code; it identifies regions that have the greatest potential to hide faults by providing a quantifiable testability score for each region.

1. A fault must be *executed*,
2. The fault must affect the state of the program in a manner different than what the state of the program would have been had the fault not existed. This is termed as having an *infection* in the state, and
3. The erroneous program state must *propagate* to an output state.

These three conditions that are necessary for a fault to cause a software failure are simulated by the processes of PIE. PIE estimates: (1) the probability that a program location is executed according to some program input distribution, (2) the probability that an injected syntactic mutant will produce a discernible difference in the resulting program state, and (3) the probability that injected program state mutants will produce a discernible difference in the program’s output. This information is then taken from PIE and fed into sensitivity analysis to predict minimum fault sizes.

## 2.2 What PIE Produces

After PIE has been performed for the entire program, there are three sets of probability estimates for each program location  $l$ :

1. Set 1: The estimate of the probability that program location  $l$  is executed ( $\hat{\epsilon}_l$ );
2. Set 2: The estimates of the probabilities, one estimate for each syntactic mutant in  $\{p_1, p_2, \dots\}$  at program location  $l$ , that given the program location is executed, the syntactic mutant will adversely affect the program state; these are denoted as  $(\{\hat{\lambda}_{l,p_1}, \hat{\lambda}_{l,p_2}, \dots\})$ ; and
3. Set 3: The estimates of the probabilities, one estimate for each live variable in  $\{a_1, a_2, \dots\}$  at program location  $l$ , that given that the live variable in the program state following the program location is perturbed, the output will be changed; these are denoted as  $(\{\hat{\psi}_{l,a_1}, \hat{\psi}_{l,a_2}, \dots\})$ .

These three estimates are necessary for the predicted minimum fault size calculation. Note that each probability estimate has an associated confidence interval, given a particular level of confidence and the value of  $n$  used in the algorithms. The computational resources available when propagation, infection, and execution analysis is performed will determine the value of the  $ns$  that can be chosen in each algorithm.

## 3 Musa’s Fault Exposure Ratio

Sensitivity analysis examines a different behavioral characteristic than reliability: *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 is faulty?*” Our testability definition asks a different question: “*what is the probability this code will fail on its next execution if it is faulty?*”

As mentioned, Musa has a parameter termed the *fault exposure ratio*,  $K$ , that is similar to Voas’s testability value. [5] states that  $K$  represents the

“fraction of time that the “passage” [of a fault] results in a failure.”

Thus  $K$  is an estimate of how frequently faults will fail to cause program failure. For example, suppose that an actual fault will only result in a failure when a program input value of 1 is entered, and further assume that this occurs once in every 10,000 program executions. Then this fault has a fault exposure ratio of  $10^{-4}$ . As another example, suppose that a fault will only cause program failure when input values greater than 20 are entered, and assume that this occurs 90% of the time. This fault would have a fault exposure ratio of 0.9.

In [5], the authors admit that at the time of their writing, the only means for determining  $K$  was from similar programs to the program under assessment, not from the specific program under reliability assessment. So if on previous, similar projects it was noticed that  $K$  was around  $10^{-4}$ , then  $10^{-4}$  would be assigned to  $K$  for the current project. We contend that sensitivity analysis, which is performed for a specific program, is a plausible alternate to Musa’s reliance on previous programs. Musa et al. also speculated in their writings that  $K$  could be based on program structure:

“At present,  $K$  must be determined from a similar program. It may be possible in the future to relate  $K$  to program structure in some way. Another possibility is that the range of values of  $K$  over different software systems may be small, allowing an average value to be used without excessive error. This could be due to program dynamic structure averaging out in some fashion for programs of any size. The small range would require that the average “structuredness” or decision density not vary from one large program to another.”

Here we agree, provided that information about the structure of the code is combined with semantic, behavior information. We can justify this additional requirement because the rate at which faults cause program failures is a function of: (1) where the fault(s) are located, (2) what type of fault(s) exist, (3) the semantics of the code that is executed after the fault(s) is executed (which will depend on the program states), and (4) the operational or test distribution. It is prudent to consider all four factors to provide an accurate, application-dependent  $K$ . Sensitivity analysis directly considers (3) and (4). Since sensitivity analysis cannot directly measure (1) or (2), sensitivity analysis indirectly considers (1) and (2) by using hypothesized faults to

estimate the behavior of the code when it contains actual faults.

## 4 Making Sensitivity Analysis Produce $K$

Sensitivity analysis produces either a *probability prediction* of the minimum fault size (in the interval  $(0,1]$ ) or an *upper bound probability prediction* of the minimum fault size, such as  $< 0.2$ , from the information produced by PIE. (Methods for determining a probability prediction or upper bound probability prediction can be found in [3].) When an upper bound probability prediction such as  $< 0.2$  occurs, it tells the user that sensitivity analysis cannot provide an exact prediction of the minimum fault size, but sensitivity analysis estimates that the probability is less than 0.2. Minimum fault size predictions that are closer to 1.0 are desirable, indicating that faults are all likely to be uncovered during testing; values that are closer to 0.0 are problematic, indicating that there could exist many faults undetected by testing. From a reliability standpoint, small minimum fault size scores, i.e., locations with scores near 0.0, could lead to high reliability estimates despite the presence of faults, since the faults are more likely to remain masked during testing.

Sensitivity analysis produces a prediction of the minimum fault size for every location  $l$  in the program; this prediction is denoted by  $\theta_l$ . (We use “location” to be a programming language statement or expression, although it could theoretically be a different level of abstraction, such as a machine code instruction.) Sensitivity analysis produces a prediction of the minimum fault size for each module  $M$  by taking the smallest  $\theta_l$  for any location in that module; this is denoted by  $\theta_M$ . Sensitivity analysis then predicts the minimum fault size for the program itself by taking the smallest minimum fault size over all modules in the program. Thus, the poorest testability over all locations in the program will be the assigned testability for the program. The reason for this is that when sensitivity analysis is used as a stop criterion for testing, we want to predict the smallest sized fault that could exist, and then test with enough test cases to detect errors of that size. Once done, we have a confidence in defect-free code.

In comparison, the way  $K$  is currently determined in industry is not a prediction of the *minimum* fault size of the program, but essentially a prediction of the *average* fault size for the *remaining* faults in the program. It is a prediction because it is based on the faults in *other* similar projects. It just so happens that these fault sizes are expected to be relatively small since they are the faults that would remain in the code near the end of V&V, right before software release. (This was true for the programs used in getting the results published in [5].) If the sizes for the remaining faults were large, and the product released, its reliability would be poor. This is significantly different from PIE, since the artificial faults used to derive the PIE estimates are not controlled for fault size, and are in general both large faults (easily found

during testing), small faults (difficult to find during testing) and faults with sizes in between these extremes. This difference is discussed again below.

Another difference between a PIE measurement and  $K$  are the units of measurement.  $K$  produces a value with units

$$\frac{\text{failures}}{\text{fault}}$$

averaged over all known faults and potentially different projects. Sensitivity analysis (using the results from PIE) produces a value with units

$$\frac{\text{output corruptions}}{\text{hypothesized fault}}$$

at a location. Even though these units are slightly different, it is plausible to substitute an “averaged” sensitivity analysis score for  $K$ , because output corruptions/hypothesized fault is a predictor of failures/fault.

PIE selects and injects a finite set of hypothesized faults from a nearly infinite set of possible artificial faults. Because PIE simulates programmer faults, the “almost” infinite set that it selects from contains both hypothesized faults that are difficult to detect and easy to detect. To compensate for this, we take the average of the  $\theta_{MS}$  for all  $N$  modules in the program for finding  $K$ :

$$K = \frac{1}{N} \cdot \sum_{\text{for each } M} \theta_M$$

This formula is likely to provide a value for  $K$  that is slightly larger than the real value for  $K$  for the residual faults in a tested application. If the estimated  $K$  is slightly larger than the real  $K$ , then calculations based on the Musa formulae will be optimistic about the uncovering of faults during testing.

We do not know how significant this bias towards a large  $K$  will be, and we are now designing empirical research to explore this question. Even with the variation, we expect that the sensitivity analysis estimate of  $K$  is much more likely to be an useful estimate than is the constant  $4.2 \times 10^{-7}$  or a  $K$  derived from other “similar” projects.

If the bias to large  $K$ s is significant, then we could adjust the sensitivity analysis estimate as follows: instead of averaging all the hypothesized fault sizes, we could have the user indicate the number of tests done on the code so far, establishing a likely maximum fault size that is likely to have survived the testing effort. Then we limit the pool of hypothesized faults that enter into the averaging to artificial faults that reveal themselves less often (during sensitivity analysis) than that maximum likely fault size. In other words, we base our estimate on artificial faults that would likely have survived testing, labeling such faults as artificially residual faults. This new method for determining a  $K$  estimate will certainly produce  $K$ 's no larger than those based on all artificial fault measurements, and most of the time they will be smaller. Our speculation (currently under

study) is that such these smaller  $K$  estimates will be more realistic reflections of the true  $K$ .

## 5 Conclusions

Software testability is an inherent semantic characteristic of executing code that is becoming an important factor to consider during software development and assessment. Sensitivity analysis has yielded positive results in prior experimentation for complementing testing; this paper has focused on a new application area, finding an application-specific fault exposure ratio for reliability assessment. Experimentation is planned in 1996 with John Musa of AT&T to validate the approach we have proposed in this working paper. The software tool PiSCES Software Analysis Toolkit<sup>®3</sup> completely automates the sensitivity analysis necessary for the approach described here, and will be used for this project.

## Acknowledgement

This research was partially funded by US Air Force Contract F30602-95-C-0158 monitored by Rome Laboratory, Griffiss Air Force Base.

## References

- [1] O. J. DAHL, E. W. DIJKSTRA, AND C. A. R. HOARE. *Structured Programming*. Academic Press, 1972.
- [2] J. VOAS, L. MORELL, AND K. MILLER. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2):41–48, March 1991.
- [3] J. VOAS AND L. J. MORELL. Applying Sensitivity Analysis Estimates to a Minimum Failure Probability for Software Testing. In *Proc. of the 8th Pacific Northwest Software Quality Conf.*, pages 362–371, Portland, OR, October 1990. Pacific Northwest Software Quality Conference, Inc., Beaverton, OR.
- [4] LARRY JOE MORELL. A Theory of Error-based Testing. Technical Report TR-1395, University of Maryland, Department of Computer Science, April 1984.
- [5] J. D. MUSA, A. IANNINO, AND K. OKUMOTO. *Software Reliability Measurement Prediction Application*. McGraw-Hill, 1987. ISBN 0-07-044093-X.
- [6] J. VOAS. PIE: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, 18(8):717–727, August 1992.

---

<sup>3</sup>PiSCES Software Analysis Toolkit<sup>®</sup> is a registered trademark of Reliable Software Technologies Corporation.