

Dependability Certification of Software Components

Jeffrey Voas (jmvoas@rstcorp.com) & Jeffery Payne

Reliable Software Technologies

phone: 703-404-9293

fax: 703-404-9295

Sterling, VA USA

Abstract

Software components need a uniform approach for rating their quality. The need for this stems from a licensee's inaccessibility to the source code as well as other information concerning how thoroughly the component was validated.

This paper proposes a "Test Quality Rating" metric that will act as a component's dependability "score." We envision a process whereby a software publisher submits a component to an independent certification organization that would then calculate the test quality rating for that component. It is preferable for component dependability validation to be performed by an independent organization. This is because even an honest dependability overestimation error on the part of the publisher could be grounds for severe legal penalties. This score would be displayed on any marketing materials or contracts which license that component. In our paper we provides results from applying the metric to a commercial financial application written in Java in order to demonstrate the effectiveness of the metric.

1 The Problem

Software quality is often viewed suspiciously when the software is delivered in executable format (i.e., as a black box). After all, it is possible that such software: (1) received minimal testing, (2) was developed according to *ad hoc* processes, (3) has high defect densities, and (4) does not match the functionality described in the software's marketing literature. Buying software is sadly similar to buying food from a street vendor: let the buyer beware.

For the last three years, our research group has been working on a project funded by NIST's Advanced Technology Program in the area of "Component-Based Software" to create assessment technologies that grade the quality of software components. Our project has centered around building a grading technology that provides the ability to infer how dependable a component is based on the thoroughness of the testing that the component received. This thoroughness gets translated into a score which becomes the component's Test Quality Rating (TQR). The TQR is a measure of the dependability of the component.

This information enables a customer to more intelligently decide whether a candidate component is "dependable enough" for their needs. This information also allows them to

decide which of several competing components to purchase given component dependability scores and licensing costs (if alternatives exist). The information also provides publishers with an opportunity to increase consumer confidence in their components.

In addition to our dependability metric, we have developed a methodology by which the software publisher can systematically increase his or her dependability scores by performing additional test activities. This will be beneficial if as we suspect, poorer dependability scores will translate into fewer licenses [3]. This will hopefully entice software publishers to only release components that have been tested more thoroughly by providing them with a mechanism for increasing revenues for better validated software components.

Also, this methodology forces better quality component offerings since component dependability scores are based on whether the test cases employed were ones that have a greater tendency to reveal software faults. If they were, dependabilities scores increase. If they were not, the scores will be lower. Thus the uniqueness in our metric and methodology stems from that fact that we do not only consider the number of test cases used but we also consider the “fault revealing” ability of those test cases [7].

2 Our Component Dependability Metric

We will begin by presenting the theoretical underpinnings of our dependability metric. From there, we will discuss fairness issues (which must be enforced by any metric that seeks ubiquitous adoption). And finally, we will discuss our results from applying this process to a Java financial application.

2.1 Theoretical Underpinnings

Many different code and process metrics exist that could be used for stamping a quality rating onto software components. For example, code complexity metrics, reliability estimates, or metrics for the degree of code coverage achieved have all been suggested in the past.

But because it is difficult to infer the quality of a component solely from code complexity or coverage achieved, and because reliability estimates are profile-dependent, we will employ a test thoroughness metric that is based on a prediction of a component’s ability to hide faults during testing. We base our dependability metric on two different pieces of information: (1) the number of test cases that have been used (more specifically, the number of consecutive test cases that have not resulted in a component failure since the last modification of the software), and (2) a prediction of how capable the component is to hide defects from the test cases. These two differing pieces of information, when combined, form the basis of the dependability model known as the Squeeze Play [6, 8].

Squeeze Play is a white-box software reliability model for ensuring that enough testing was performed to detect the predicted smallest-sized fault that could hide in the software. If testing to that level occurs and the software does not fail, then the Squeeze Play model says, with a numerical confidence, that no faults are hiding in the software. The Squeeze Play does not assert that the software is correct, but instead provides a probability, based on repeated tests, that the software is correct.

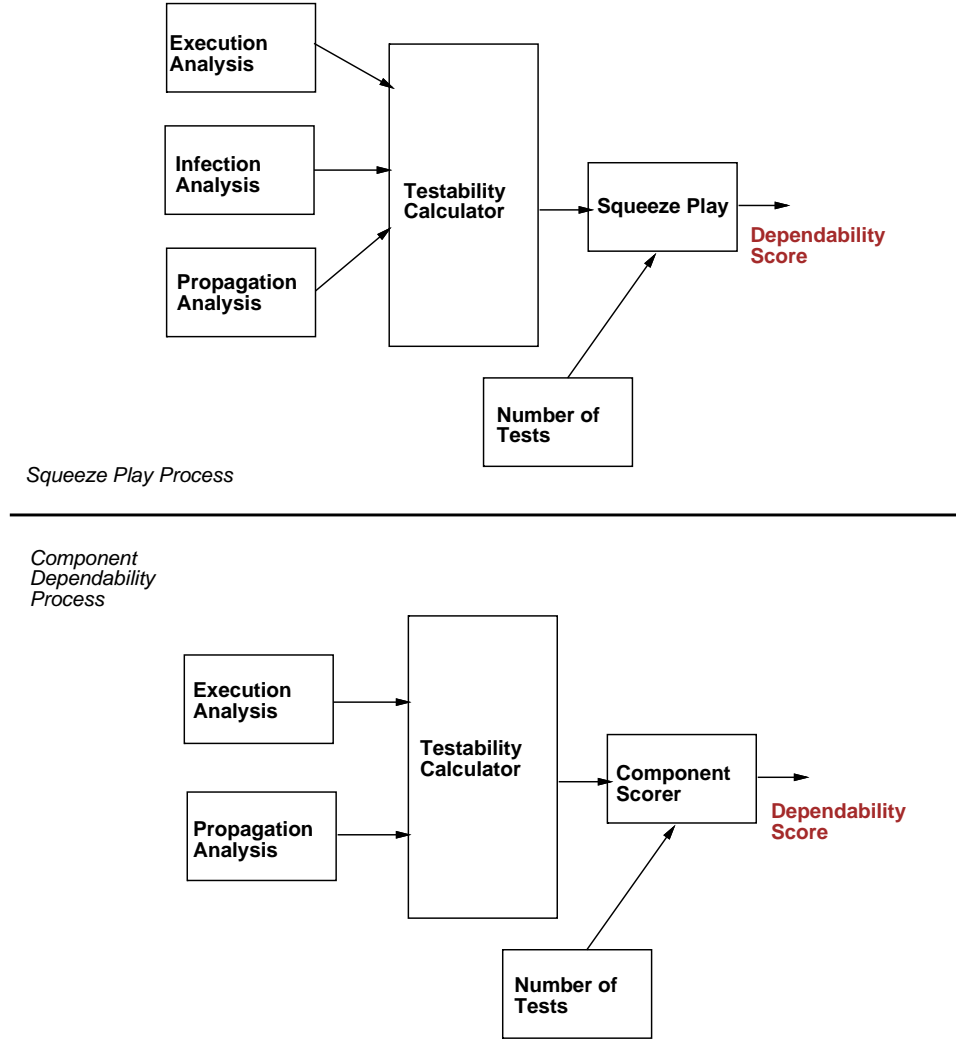


Figure 1: Comparison of the Squeeze Play model and the TQR dependability model

Squeeze Play is similar to the more common software reliability estimation models but distinct. *Software reliability* is the probability that the software will not fail in a fixed environment for a fixed period of time [9]. For example, if a program fails once in 100 test cases, its reliability is roughly 0.99, and most software reliability models will provide a score close to that. This is quite different from a confidence that the software is correct. In contrast, our metric not only considers the number of test cases where the software performed correctly but it also considers the predicted fault hiding ability of the software.

Although our metric was derived from the Squeeze Play, pragmatic considerations do not allow us to apply each of the three white-box analyses discussed in the original description of the Squeeze Play [6]. As we will discuss later, one of the three, *infection analysis*, which is a code mutation process, will be ignored. Hence our dependability metric is a quantitative measure of the quality of the testing performed as opposed to the probability of absolute correctness.

For a software publisher to calculate a component dependability score, two pieces of

information will be necessary: (1) the number of test cases used during testing, and (2) *propagation analysis* and *execution analysis* scores [11]. Software publishers should have no difficulty in determining the first piece of information, i.e., the number of successive test cases where the software produced the correct result since the last code modifications. Only when a publisher does not know what is correct versus incorrect will problems occur [4]. In order for a publisher to gather the second piece of information, propagation and execution analysis will be applied to the software component [11].

The second needed piece of information is derived after propagation analysis and execution analysis are applied to the software component. These analyses are 2 of the 3 analyses defined in the PIE software testability technique [11]. PIE is a dynamic analysis that produces estimated probabilities for:

1. The likelihood that a statement in a component is *executed*,
2. The likelihood that a mutated state will *infect* the component’s state, and
3. The likelihood that a corrupted program state will *propagate* and cause the component’s output to be mutated.

For our dependability metric, the first and third of these probabilities are multiplied together to get a prediction of how likely it is that a particular statement in a component will hide a defect during testing. This numerical prediction is the testability value for that statement in the component, and once we have these predictions for all statements in the component, then we can calculate the other key piece of information needed before a component dependability score can be computed.

How we do this follows. From the testability scores of each statement in the component, the lowest score will serve as the testability score of the component. This numerical value predicts how likely it is that the component will hide defects during testing by using a prediction of how small the smallest sized fault in the component might be. For example, a fault caught by any test case is said to be larger than a fault caught by only 1% of all test cases. This predicted smallest-sized fault value is used in the Squeeze Play model to calculate how many test executions are needed to be convinced that no faults are hiding in the component (with a confidence level) [8]. With information concerning the number of tests (on the current version with no failures) and the propagation and execution scores, a component’s dependability can be computed.

2.2 Formulae

As mentioned, a statement’s testability score is the product of the propagation and execution scores of that statement. The reason for taking the product of these scores is traced back to fact that the probability that any fault will cause the software to fail is simply the product of: (1) the likelihood of executing that fault, (2) the conditional probability of infection occurring given that the fault gets exercised, and (3) the conditional probability of the corrupted data state corrupting the software’s output after the state gets infected.

The testability score then for component α will be:

$$\hat{t}_\alpha = E_\alpha \cdot P_\alpha$$

where E_α is the execution estimate for α and P_α is the propagation estimate for α . Recall that a component's testability is the smallest testability score of any statement in the component.

The formula for the dependability score for α is:

$$D_\alpha = 1 - (1 - \hat{t}_\alpha)^N \quad (1)$$

where \hat{t}_α is the testability and N is the number of times α was executed without failing. Note that the N tests are selected at random according to the profile used with PIE was performed. Dependability scores are numerical values in the interval $[0,1]$. To determine the number of tests needed for a fixed score, Equation 1 can be rewritten as:

$$N = \lceil \frac{\ln(1 - D_\alpha)}{\ln(1 - \hat{t}_\alpha)} \rceil \quad (2)$$

We believe that Equation 1 will be attractive to both component consumers and publishers because it preserves the following properties:

1. Greater testability scores and greater amounts of testing result in a greater dependability.
2. Lower testability scores and lesser amounts of testing result in a lesser dependability.
3. A poor score can be improved by doing additional testing and/or increasing the software's testability by adding assertions.
4. Lower testability scores require greater amounts of testing to still provide a fixed dependability score.
5. Higher testability scores require lesser amounts of testing to provide a fixed dependability score.

Without guaranteeing the above trade-offs between predicted fault hiding ability and degree of testing, any test-thoroughness measure must be viewed suspiciously. But simply having a metric that preserves these properties is not enough. It is also necessary to have a tool that preserves the following properties:

1. Reproducible results,
2. Difficulty for a component publisher to forge results in an independent laboratory is not used; this does not mean that the publisher cannot improve a score legitimately, but any score improvement must be done via appropriate means, and
3. Highly automated methods for calculating the resulting metrics. This will decrease costs, decrease the likelihood of human error, and result in fewer misclassifications during certification.

The automated tool that we have developed to compute Equation 1 and the information required by Equation 1 guarantees these additional attributes.

2.3 Fairness

We will now address questions that arise concerning why infection analysis is not included in our metric as well as other issues concerning fairness.

We decided that infection analysis (which calculates the likelihood that a mutated state will *infect* the component's state) was inappropriate for our metric for four key reasons. First, infection analysis is computationally expensive. This would make the metric impractical for large software components. Second, infection analysis relies on source code mutations in its software experiments. Although such mutations have been the subject of decades of research [10], there is no general agreement on what mutations are necessary or sufficient for accurately mimicking software faults that programmers make. Third, semantically equivalent mutations require human intervention. And fourth, we believe that publisher might begin to change the language constructs they use simply to ensure that certain mutants were not created.

Semantically equivalent mutants create an interesting problem here. Semantically equivalent mutants occur when a source code mutation produces a slightly different program that does not calculate a different input / output function from the original program. In other words, the mutated program looks different, but is semantically identical to the original. Recognizing this situation is problematic but very necessary. If a certain number of test cases do not show any difference between a mutant and the original, a terrible testability score for the component will result which in turn will cause a terrible dependability to result. It may be because we haven't tried enough tests (i.e., the difference between them is obscure) or because the mutant is semantically equivalent. Short of exhaustive testing (which is almost always impractical), human analysis is the only way to ascertain if the mutant is semantically equivalent. And human intervention is problematic for the dependability analysis which must be seen as fair and must be as efficient as possible.

Further, the goal here is to provide a *fair* certification technology to each software publisher. Since it is not possible to use all potential code mutants for a component (as there are an infinite set for any component), a handful is selected that are randomly generated. Which ones are selected can bias infection scores. Also, it is possible that developers could determine how to write code simply to decrease the likelihood that certain mutants were generated during certification in order to artificially boost the testability scores. And the last thing that we want is to simply create meaningless hoops for developers to feel that they must jump through to gain better dependability scores.

Since these problems are unacceptable when seeking to provide a fair, unbiased certification process, our metric does not employ infection analysis and instead employs propagation and execution analysis. These two analyses of the PIE model are more difficult to bias than is infection analysis.

While on the subject of fairness, we should also explore the impact of the operational profile (that is used when PIE is performed) on the testability results. We recommend the use of a uniform profile when this analysis is performed, but if that does not occur, then we should consider whether different test profiles can alter testability scores. Put simply, they can. Therefore it is possible that a publisher could tweak a distribution to vary the results. However, early research by Michael [5] suggests that propagation analysis may not be as sensitive to the operational profile as we might expect. Further, even if propagation

scores were improved in this manner, the publisher would risk worsening the results from the execution analysis. A publisher would be far better off using assertions to boost propagation scores than distribution mangling, particularly since the value-added from correct assertions is profile independent [3].

In addition to the futility of mangling profiles, there is one more “check and balance” that our model uses to discourage cheating. That is a measure of how many test cases were employed during PIE relative to the testability score calculated. For example, you cannot have accuracy to the 10th decimal place with a sample size of 5. Our model seriously penalizes the practice of using few test cases when PIE is performed because of the increased likelihood of a statistical approximation error on the confidence interval’s lower bound. This is handled using Hoeffding’s inequality which is formalized in [2].

And finally, note that publishers can lie about N when computing D_α . Independent laboratories can be used to eliminate this possibility by generating the test cases at the laboratory and performing the testing in some manner similar to the following process:

1. Software publisher supplies requirements and input range,
2. Independent certification laboratory generates tests from a *uniform* distribution over that range,
3. Independent certification laboratory determines if output is correct using the requirements,
4. Publisher pays laboratory for independent testing; publisher decides how many tests to buy based on \hat{t}_α , and
5. If the software fails on any of these tests, then the publisher needs to revise the software and re-perform propagation and execution analysis as well as to resubmit the software for testing.

If dependability scores are calculated at an independent laboratory, we recommend that the laboratory use a uniform profile over an input range specified by the publisher.

3 Results from a Java Financial Application

We performed experimentation on a real world financial application written in Java. The amount of time it took to complete the analysis was approximately 2 person-weeks. That included the time that the tool ran as well as the time it took to manually sift through the results.

The candidate system was a complex, data-driven Java-application capable of performing several thousand different computations while processing over 1000 input variables. Accuracy is critical, as this application allows users to plan their financial futures. The size of the application was around 21,000 source lines of code with 127 classes.

The goal of experimenting with this Java application was to determine how accurate the statement testability and component scores were. If statement testability scores are accurate, then component testability scores will also be accurate, since a component testability score

Seeded Error Experiment	Statement Testability	Number of Test Cases Predicted to be Needed	Accurate?
1	0.015	705	No
2	0.96	2	No
3	0.96	2	Yes
4	0.5	5	Yes
5	0.27	14	Yes
6	0.55	6	Yes
7	1.0	1	Yes
8	0.55	6	Yes
9	0.355	7	Yes
10	0.50	7	Yes
11	1.0	1	Yes
12	0.54	5	Yes
13	0.50	5	Yes

Table 1: Results from the 13 Experiments

is likely to translate into a liberal estimate of the amount of testing needed for most faults in the component. This follows since the component testability is the lowest statement testability.

The experimentation began by seeding faults into the components because no known faults existed in the application. Then the faulty application received propagation analysis and execution analysis. Dependabilities were then calculated for the statements and components. The goal was to see if the number of test cases it took to reveal seeded errors in a statement was of the order of magnitude predicted by the statement’s testability score. The results are shown in Table 1.

Thirteen different errors were seeded. We will now walk through the results.

Seeded Error 1 The first statement had a testability score of 0.015, which means that a test suite with 705 test cases should detect any error in that statement. After building six different test suites of size 705, two test suites revealed the error.

Seeded Error 2 The second statement had a testability score of 0.96, which means that a test suite with 2 test cases should detect any error. After building 700 different test suites of size 2, only 77 test suites revealed the error.

Seeded Error 3 The third statement had a testability score of 0.96, which means that a test suite with 2 test cases should detect any error. After building 700 different test suites of size 2, every test suite revealed the error.

Seeded Error 4 The fourth statement had a testability score of 0.5, which means that a test suite with 5 test cases should detect any error. After building 100 different test suites of size 5, every test suite revealed the error.

Seeded Error 5 The fifth statement had a testability score of 0.27, which means that a test suite with 14 test cases should detect any error. After building 54 different test suites of size 14, every test suite revealed the error.

Seeded Error 6 The sixth statement had a testability score of 0.55, which means that a test suite with 6 test cases should detect any error. After building 116 different test suites of size 6, 115 test suite revealed the error.

Seeded Error 7 The seventh statement had a testability score of 1.0, which means that a test suite with 1 test case should detect any error. Seven hundred of 700 test suites caught this error.

Seeded Error 8 The eighth statement had a testability score of 0.55, which means that a test suite with 6 test cases should detect any error. After building 232 different test suites of size 6, 231 test suite revealed the error.

Seeded Error 9 The ninth statement had a testability score of 0.35, which means that a test suite with 9 test cases should detect any error. After building 174 different test suites of size 9, 173 test suite revealed the error.

Seeded Error 10 The tenth statement had a testability score of 0.50, which means that a test suite with 7 test cases should detect any error. After building 200 different test suites of size 7, 199 test suite revealed the error.

Seeded Error 11 The eleventh statement had a testability score of 1.0, which means that a test suite with 1 test cases should detect any error. After building 300 different test suites of size 1, 204 test suite revealed the error.

Seeded Error 12 The twelfth statement had a testability score of 0.54, which means that a test suite with 5 test cases should detect any error. After building 116 different test suites of size 5, 115 test suite revealed the error.

Seeded Error 13 The twelfth statement had a testability score of 0.50, which means that a test suite with 5 test cases should detect any error. After building 100 different test suites of size 5, 99 test suite revealed the error.

Had we used the component testability score to calculate the number of test cases necessary to find faults seeded into that component (instead of the statement's score for the statement that received the fault seeding), our estimate for the number of test cases needed would have always been more than enough to reveal the seeded errors. But that would not have been true had the two locations where the seeded errors were not caught as quickly as suggested by the statement testability scores been the locations in the components with the minimum testability scores. This possibility needed further investigation.

3.1 Coverage and Boundary-value Tests

The next step was to determine why some of the statement testabilities were too liberal. Our suspicion was that the seeded errors rarely infected the component's state. If this were true,

then that suggests that infection analysis may be necessary for more accurate testability predictions even given the practical limitations already described.

After testing the seeded errors' infection frequencies, our suspicion was validated: in those cases where a statement testability was too high, it was because the seeded error had a low infection frequency. And since we do not perform infection analysis, we do not have access to that information.

But this is a catch-22. To have accurately calculated every statement's testability, we would have needed to know what the seeded errors were going to be *a priori* in order to employ the precise mutants during infection analysis. And worse, in practice, we'd need to know what the real faults were in order to make the appropriate mutants. This is implausible and showed a limitation in our model.

As it turned out, both of the seeded errors that didn't result in accurate scores were in conditional statements. This suggested the need for ways to improve the accuracy of the testability scores through advances in execution analysis or related coverage techniques. Here, we examined the use of two higher order coverages: CDC/MCC coverage and *boundary-value* coverage. Condition / Decision Coverage (CDC) [1] measures whether each individual condition within a decision has been exercised in both the TRUE and FALSE directions. Multiple Condition Coverage (MCC) [1] is similar to CDC but it additionally requires that every possible combination of TRUE/FALSE values in each condition is exercised (taking into account the short-circuiting nature of Boolean expressions in Java).

Consider the following sample code:

```
if ((a == b) && (c == d)) {
    System.out.println("Entered the loop");
} else {
    System.out.println("Did not enter the loop");
}
```

Now suppose an error is placed in the line with the `if` statement, as follows:

```
if ((a == b) && (c != d)) {
    System.out.println("Entered the loop");
} else {
    System.out.println("Did not enter the loop");
}
```

Assume that the line with the `if` statement will execute during every test case. Also assume that during propagation analysis the value of the entire `if` expression is reversed (TRUE becomes FALSE or FALSE becomes TRUE) thus causing the output to be different for every test case. Here, the testability score will always be 1.0 because the code always executes and the corrupted state always propagates.

Unfortunately, if the only test cases used causes `(a == b)` to be FALSE, the fault will not be discovered despite the fact that this statement has high testability. We predicted that in situations like this, CDC/MCC coverage-adequate test data could improve the fault revealing ability of the test suite used.

During experimentation using random data for the above example, `a`, `b`, `c`, and `d` were set to 0 or 1 with equal probability. CDC/MCC coverage results showed that all condition

combinations had been hit. Using this test data, the first seeded error was observed by 516 out of 1000 test cases. Therefore, test data that was CDC/MCC adequate helped to reveal the low infection fault.

In addition to the potential benefit from adding CDC/MCC-adequate test cases to our dependability model, we also analyzed whether boundary-value test cases for relational conditions might be valuable. Consider the following sample code:

```
if ((a > 10000) && (b > 5000)) {
    System.out.println("Boundary exceeded.");
} else {
    System.out.println("Boundary not exceeded.");
}
```

Now suppose an error is placed in the line with the `if` statement as follows:

```
if ((a > 9990) && (b > 5000)) {
    System.out.println("Boundary exceeded.");
} else {
    System.out.println("Boundary not exceeded.");
}
```

In this example, none out of 500 test cases revealed the fault in the code despite having 100% CDC/MCC coverage. The problem is that only ten integers (9991 to 10000) will cause the seeded error to surface. There is no guarantee that any of the 10 integers will be chosen even if every conditional combination is exercised. This seeded error will be revealed if variable `a` is given a value that is marginally greater than the boundary value of 9990 (e.g., 9991). In our experimentation employing boundary value test data, all test suites revealed the seeded error, thus showing that boundary-value test data is yet another way to augment our dependability model to account for the missing infection analysis.

In summary, CDC/MCC coverage and boundary-value adequate testing is not a replacement for infection analysis; they cannot expose all low infection faults. Still, they do provide insight for creating additional test data that may reveal low-infection faults. Our plan is to continue experimentation with our dependability model and if we continue to find that CDC/MCC coverage and boundary-value adequate test suites frequently account for infection analysis, we plan to add those tests into the number of required test cases according to Equation 2. But at this point, we cannot make that assertion.

4 Conclusions

In summary, our NIST-sponsored research project has resulted in:

1. A test thoroughness metric that can be used to rate the dependability of software components,
2. An automated certification tool for Java components, and

3. A methodology by which a publisher can legitimately improve testability scores. Publishers can employ assertions to boost propagation estimates. And they can employ an intelligent test case generation tool that we have prototyped to boost low execution scores [3].

The metric allows a component purchaser to know how much validation was done relative to the component's predicted *defect hiding ability*. The metric is not a traditional *reliability estimate* or *probability of failure*. It simply provides information concerning how much testing was performed relative to the predicted fault hiding ability of the component. It is, however, based on the fault/failure model, which defines the processes necessary for a software program to fail as the result of an existing fault [12]. While our results were not perfect, because of the need to add specialized test cases to account for the missing infection analysis, we showed that our approach is a reasonable means for determining how much testing is needed for a software component. If that level of testing is achieved by a publisher, then that publisher should be rewarded.

The only unknown at this point is whether component consumers will actually bias their spending decisions on the thoroughness of the testing that a software system received. That is, will they be amenable to paying more for components that have been more thoroughly tested? Or might consumers opt to not license any component that did not have a component dependability rating associated with it? These answers will only be gathered over time and after the consequences of software failure start outweighing (in importance) the frequency of software failure.

Acknowledgments

The authors thank Keith Miller for reviewing an earlier draft. This work has been supported by National Institute of Standards and Technology Advanced Technology Program Cooperative Agreement Number 70NANB5H1160. THE OPINIONS AND VIEWPOINTS PRESENTED ARE THE AUTHOR'S PERSONAL ONES AND THEY DO NOT NECESSARILY REFLECT THOSE OF THESE AGENCIES.

References

- [1] G. MYERS. *The Art of Software Testing*. Wiley Interscience, 1979.
- [2] W. HOEFFDING. Probability inequalities for sums of bounded random variables. *American Statistical Association Journal*, pages 13–30, March 1963.
- [3] J. VOAS AND L. KASSAB. Using Assertions to Make Untestable Software More Testable. *Software Quality Professional*, November 1998.
- [4] P.E. AMMANN, S.S. BRILLIANT AND J.C. KNIGHT. The Effect of Imperfect Error Detection on Reliability Assessment via Life Testing. *IEEE Transactions on Software Engineering*, 20(2):142–148, February 1994.

- [5] C. C. MICHAEL. On the regularity of error propagation in software. Technical report, Reliable Software Technologies Corporation, Sterling, Virginia, 1996. Research Division Technical Report RSTR-96-003-04.
- [6] J. VOAS AND K. MILLER. Improving the Software Development Process Using Testability Research. In *Proc. of the 3rd International Symposium on Software Reliability Engineering*, pages 114–121, Research Triangle Park, NC, October 1992. IEEE Computer Society.
- [7] J. VOAS AND K. MILLER. The Revealing Power of a Test Case. *J. of Software Testing, Verification, and Reliability*, 2(1):25–42, May 1992.
- [8] J. VOAS AND K. MILLER. Software Testability: The New Verification. *IEEE Software*, 12(3):17–28, May 1995.
- [9] J. D. MUSA, A. IANNINO, AND K. OKUMOTO. *Software Reliability Measurement Prediction Application*. McGraw-Hill, 1987. ISBN 0-07-044093-X.
- [10] R. A. DEMILLO, R. J. LIPTON, AND F. G. SAYWARD. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [11] J. VOAS. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Eng.*, 18(8):717–727, August 1992.
- [12] M. FRIEDMAN AND J. VOAS. *Software Assessment: Reliability, Safety, Testability*. John Wiley and Sons, New York, 1995. ISBN 0471-01009-X.