Object-Oriented Software Testability

Jeffrey M. Voas

Reliable Software Technologies Corporation

*Loudoun Tech Center, Suite 250, 21515 Ridgetop Circle, Sterling,
Virginia 20166, USA, Telephone: 703.404.9293, Fax: 703.404.9295,
email: jmvoas@RSTcorp.com*

Abstract

This paper studies whether object-oriented systems are more likely to hide faults during system level testing. When this project began, we suspected that although object-oriented designs have features particularly supportive of software reuse, this paradigm is detrimental to system level testing for fault detection. After comparing the testability of both OO and procedural systems, this project studied ways to engender reusability, while maintaining an acceptable level of testability. By studying whether this family of programming languages and design paradigm are harmful to testability, and evaluating methods to increase testability, we provide information that should be considered before OO languages are endorsed as tools for developing safety-critical software.

Keywords

Software testing, testability assessment, information hiding, encapsulation, assertions

1 INTRODUCTION

This paper is a condensed version of a technical report that was provided to The National Institute of Standards and Technology (NIST) in December, 1994. The purpose of this NIST-sponsored project was to study whether object-oriented systems are more likely to hide faults during system level testing. When this project began, we suspected that although object-oriented designs have features particularly supportive of software reuse, this paradigm is detrimental to system level testing for fault detection. After comparing the testability of both OO and procedural systems, this project studied ways to engender reusability, while maintaining an acceptable level of testability. By studying whether this family of programming languages and design paradigm are harmful to testability, and eval-

uating methods to increase testability, we provide information that should be considered before OO languages are endorsed as tools for developing safety-critical software.

This project studied the impact of (1) inheritance, (2) polymorphism, (3) function overloading, (4) encapsulation, (5) information hiding, and (6) software assertions on object-oriented systems. Recommendations for how to maintain sufficiently high testability while maintaining object-oriented design (OOD) were developed. Our major recommendations were:

1. Formal methods, although capable of demonstrating dependability of objects, do not demonstrate that the manner in which the objects are linked together produces a dependable composition.
2. Assertions appear to have the greatest ability to preserve OOD while increasing testability and hence decreasing testing costs; also, we have provided a simple lemma of why software assertions cannot lower software testability.
3. Information hiding and encapsulation are detrimental to state error propagation, which is very necessary if faults are to be found via software testing.
4. Abstract data types have little impact on software testability as far as we can determine.
5. Inheritance is not necessarily detrimental to software testability; however, when it is combined with information hiding, it may become a “lethal” combination. Unit testing costs increase as the depth of inheritance increases due to the number of drivers and stubs that are necessary. As a counter argument, subclasses tend to be simpler in deep, complex inheritance trees and hence increase the ability to assess high reliability of the subclasses. This presents another related problem: *the reliable composition of reusable subclasses*.
6. Polymorphism is difficult to test, i.e., find test cases to exercise different binding scenarios. However from our previous intuition and this effort’s experimentation, polymorphism, when faulty, causes the faults to be of larger sizes, and that means increased testability, and hence it is not problematic for testing.

We will now explain the testability model and experiments that we performed in building these recommendations.

2 BACKGROUND

Previous research has conjectured that procedural languages are less likely to hide faults during system level testing than are OO languages. On the other hand, object oriented languages show advantages for developing software that is highly maintainable, readable, structured, and reusable. Thus for safety critical software we have a conflict:

We would like to use OO languages to ease the process of reusing components but might then pay a price in terms of reliability (which for safety-critical systems we cannot afford).

In the object-oriented paradigm, *objects* are atomic units of encapsulation, *classes* manage collections of objects, and *inheritance* structures collections of classes. Objects partition the state of computation into encapsulated units. Each object has an interface of operations that control access to an encapsulated state. The operations determine the object's behavior, while the state serves as a memory of past executions that might influence future actions. Note that procedural and object-oriented languages differ sharply in their patterns of resource sharing. Procedure-oriented languages encourage autonomy at the procedure level, while not encouraging interaction through nonlocal variables. Object-oriented languages discourage procedure autonomy, and instead organize procedures into collections of operations that share an object's state through nonlocal variables. Thus encapsulation at the level of objects derives its power by abandoning encapsulation at the level of procedures. It is our conjecture that encapsulation at the object level is very detrimental to overall system testability.

When we talk about a potential problem with the OOD philosophy and software testability, we are talking about testability assessments at the system testing level, not the unit testing level. Small objects and classes can be tested quite thoroughly in isolation, but it is at the higher levels of object composition where we suspect that there is a reduction in testability, e.g., due to capabilities afforded the developer such as **private** and **protected** data objects. There are cost benefit and reuse claims for why OO languages should be used, and we are not in disagreement with these claims. For example, given the enormous costs of software, maintenance and reuse are characteristics that OO languages support nicely. So from an economic perspective, those are advantages. But from a testing perspective, OO languages may hide catastrophic faults for longer time intervals, and if true, this makes testing such systems more difficult, and hence the benefit of these languages must be brought into question.

2.1 Our Basic Testability Approach

For clarity, we should define what we mean by the term “software testability”; our definition differs slightly from the way others commonly use the term. 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. For example, if the goal is to select a set of inputs that execute every statement in the code at least once, and it is virtually impossible to find a set to do so, then the testability ascribed would be lower than if it were easy to create this set. (For a thorough reference on the differing perspectives on software testability of OO software, see (Binder, 1994).)

In an effort to minimize the risk of performing too little testing and becoming excessively confident in the absence of faults, Voas (Voas, 1992) (Voas and Miller, 1995) (Voas et al., 1991) redefined the term software testability. Voas defines “software testability” to be a *prediction of the probability of software failure occurring due to the existence of a fault*, given that the software testing is performed with respect to a particular input distribution. Other well-known researchers have since accepted this new definition in their

research initiatives (Howden and Huang, 1993). This definition says that the testability of the 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 these definitions of software testability would converge, producing a lower software testability prediction. 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 claim can be justified.

In Voas's definition, if we observe no failures while testing (given input selection technique D), we can boast high confidence that faults are not hiding *only* if we have a priori knowledge that the testability of the software was high. However, if after testing according to input selection technique D , we observe no failures and are told 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. The other factors considered before this definition of testability converges with the coverage-based definition of testability is provided by a technique termed "sensitivity analysis" (Voas, 1992) (Voas and Miller, 1995) (Voas et al., 1991).

3 EXPERIMENTAL RESULTS

RST Corporation has developed an automated sensitivity analysis tool in their *PiSCES Software Analysis Toolkit*^(TM). We examined and compared the testability of C (procedural) and C++ (object-oriented) code using this tool. For the experiment, we developed an automatic-teller machine (ATM) program in C and C++. We also developed a simple SHAPES drawing package in both languages as a second experiment.

Due to the size of this NIST-sponsored effort, the amount of experimentation was limited, and not statistically conclusive. We wish however to place the results in the public domain to encourage discussion and independent experimentation.

3.1 The ATM Experiment

We first summarize the results obtained by performing propagation analysis on both object-oriented and procedural versions of an Automated Teller Machine (ATM) simulation. The ATM system was coded in both C and C++ from a simple ASCII specification. In this experiment, we focused solely on the impact of encapsulation and information hiding. 102 test cases were developed such that all locations in the program were covered.

The Automated Teller Machine (ATM) program simulates a single ATM connected to a bank. The machine accepts ATM cards and verifies the validity of the user by accepting a PIN number and matching it with the users PIN number maintained at the bank. If the user enters three unsuccessful PIN numbers, the machine eats the card and informs

Paradigm	Minimum Inequality	Minimum Point Estimate
Object-Oriented	0.0098	0.011
Procedural	0.011	0.075

Table 1 The lowest location upper bound and point estimate propagation scores for any location in the respective versions.

Paradigm	# of Inequalities	Percentage of Inequalities
Object-Oriented	17	16.2%
Procedural	16	14.3%

Table 2 The total number of locations for which propagation never occurred and the percentage of these with respect to the total number of locations.

the user to contact the bank. If valid, the user has access to one checking and one savings account. Possible transactions includes: withdrawals, transfers, and balance checks. A transaction is invalid if either the user tries to withdraw greater than \$200 per access or attempts a transfer/withdraw that overdraws an account. Each valid transaction generates a separate receipt. All receipts are printed when the user has completed all desired transactions.

Procedural and Object-oriented versions of ATM were developed from a generic specification including information contained in the previous paragraph. The procedural version maintains a set of data structures and a set of procedures that operate on data to perform the aforementioned tasks. the object-oriented version consists of a set of classes that combine to create functioning ATM and bank objects.

Inputs to the program take the form:

```
<atm card>
<pin number>
<transactions>*
<quit>
```

Tables 1- 3 summarize the results. The average and minimum scores in Tables 1- 3 are based on propagation estimates at the location level. In Table 1, we are showing both the minimum inequality and minimum point estimate.* In this table, it is interesting to note that the minimum point estimate is almost 7 times larger for the procedural than for the object-oriented. Although this may seem large, it should be noted that these values are of

*An inequality occurs whenever propagation never occurred; instead of providing a 0.0 propagation estimate, we place an upper bound on the point estimate, making it into an inequality.

Paradigm	Average Propagation Score
Object-Oriented	0.49
Procedural	0.57

Table 3 The average propagation point estimate score (this does not include upper bounds).

the same order-of-magnitude. Table 2 shows the number of locations (although not from identical code) that demonstrated no propagation; this was only slightly greater in the OO system than in the procedural system. But then again, it was too close to draw any statistical significance. Table 3 shows that on average, it will take 20% more test cases to test the OO code than the procedural code. Once again, this is not a large difference.

In summary, this example showed that for the same 102 test cases, the OO design with encapsulation and information hiding faired slightly worse in enforcing the propagation condition. This experiment suggests that information hiding and encapsulation are detrimental to testability, specifically propagation, which has been our suspicion for several years.

3.2 The SHAPES Experiment

SHAPES is a simple draw package that allows various shapes to be drawn, moved, and stacked on top of one another. Currently SHAPES supports three basic shapes: lines, rectangles, and a simple face (head, eyes, nose, mouth) made out of the first two shapes. The program defines a drawing space in which shapes can be manipulated by changing their *x*, *y* coordinate positions.

In this example, our OO design enforced inheritance and polymorphism, and function overloading in C++, whereas the ATM experiment enforced encapsulation and information hiding. Here, we were interested in looking at whether inheritance, polymorphism, or function overloading have an observable negative impact on testability. To draw comparisons we used a C procedural version that did not include these features.

Input values to SHAPES consist of information for creating shapes and manipulating them. A SHAPES input file is in the following format:

```
<number of shapes>
<shape type><shape position>*
<shape manipulation><parameters>*
```

All shape diagrams are output to the screen after creation or a refresh. Tables 4- 6 summarize the results of this experiment.

What this experiment suggests is that inheritance, polymorphism, and function overloading in C++ do not have a negative impact on propagation as opposed to the pro-

Paradigm	Minimum Inequality	Minimum Point Estimate
Object-Oriented	0.031	0.32
Procedural	0.033	0.20

Table 4 The lowest location upper bound and point estimate propagation scores for any location in the respective versions.

Paradigm	# of Inequalities	Percentage of Inequalities
Object-Oriented	3	4%
Procedural	37	26%

Table 5 The total number of locations for which propagation never occurred and the percentage of these with respect to the total number of locations.

cedural version. In fact, in all categories for which we compared results, the OO version outperformed the procedural version. We are not conjecturing that these characteristics are *beneficial* to testability, but that we were surprised by the results. We have begun a close examination of the C version to determine why this occurred.

Before this project began, we expected that inheritance, in isolation from information hiding, would not be problematic. It is when information is hidden from lower methods that propagation can be thwarted. As for polymorphism and function overloading, we also expected no direct impact on testability, however in combination with other features, this could be different. This experimentation has suggested that encapsulation and information hiding are detrimental to effective fault detection with system level testing. Although this experiment can only be viewed as one data point, it does agree with the original hypothesis that was put forth several years ago (Voas, 1991). Additional research is required, but this is preliminary evidence from an actual OO system where the hypothesis was substantiated. This task has failed to conclude whether polymorphism or inheritance are detrimental to testability, given that the OO version produced better results than its procedural counterpart. Once again, this evidence must be viewed as a single data point and is not conclusive.

Paradigm	Average Propagation Score
Object-Oriented	0.729
Procedural	0.576

Table 6 The average propagation point estimate score (this does not include upper bounds).

3.3 Assertions

Given that encapsulation and information hiding are probably detrimental to OOD, we then applied one specific design-for-testability (DFT) heuristic to the aforementioned C++ code, *assertions*.

An *assertion* is a test on the entire state of an executing program or a test on a portion of the program state.[†] Although empirical assertions are validation mechanisms, their use in hardware testing has earned them the label of “design-for-test” mechanisms, and thus we will also consider them here as an approach to improving the design of OO software.[‡] (“Observability” is the term used in hardware design representing the degree to which the inner logic of a chip can be tested.)

Typically, software testing checks the correctness of values only after they are output. In contrast, assertions check intermediate values, values that are not typically defined as output by the requirements. The benefit of checking internal states is that testers know as soon as possible whether the program has entered into an erroneous state.

Assertions can be derived at any time during the software life-cycle, however we will only use them once code is available. When developing software using formal methods, assertions are employed before code is available. Here, we are looking at applying assertions during the software assessment phase, but we wish to derive these assertions during design using measures for where they should be placed by quantifying metrics such as the DRR. Typically, assertions are used during testing (to improve testability) and removed during deployment (for efficiency and speed). The removal of assertions however can be problematic, and it is preferable that this is done in an automated fashion to lessen the probability of human error. Removing assertions after testing is analogous to compiling without the debug flag when run-time errors are no longer being experienced. (Assertions remaining in production software can be useful in detecting and diagnosing problems.) Also of extreme importance are the mechanisms used to derive the assertions from the requirements or specification; incorrect assertions can lead to a false sense of a good design process.

Assertions that are placed at each statement in a program can automatically monitor the internal computations of a program execution. However, the advantages of universal assertions come at a cost. A program with such extensive intrusive instrumentation will execute more slowly. Also, some assertions may be redundant. And the task of instrumenting the code with “correct” assertions at each location is of high risk; there is no guarantee that the assertions will be correct.

We advocate a middle ground between no assertions at all (the most common practice) and the theoretical ideal of assertions at every location. Our experiments showed that OO systems can have regions of quite high testabilities, and hence assertions are not war-

[†]The type of assertion that we are interested in is *empirical* assertions that are invoked when the code is executed; we are not talking about formal logic pre and postconditions that are used in a “cleanroom” like development process.

[‡]By “improving the design of OO software,” we mean “forcing a more explicit and better specified design.”

ranted. A plausible compromise is to inject assertions only at locations where traditional testing is unlikely to uncover software faults (Voas and Miller, 1994). For instance, we can statically detect information hiding within a design, and we can easily place assertions on encapsulated variables. By doing so, we explicitly test the validity of computations at the system and unit level that are normally tested implicitly.

We will assume that all assertions are logical and evaluate to TRUE when the internal state is satisfactory (meaning it passes the particular test in the assertion), and FALSE otherwise. When an assertion evaluates to FALSE, we consider the execution of the program to have resulted in failure, even if the specified output is correct. You can think of this as “artificially” modifying what is defined as failure:

failure is said to occur if the output is incorrect *or* an assertion fails.

In essence, this not only redefines failure, but it modifies what is defined as output.

Lemma of the Impact of Assertions on Testability

Here, we wish to give a simple lemma that the impact of an assertion on error propagation must either be: (1) negligible, or (2) positive. Error propagation is a direct factor in assessing software testability: greater error propagation implies greater testability. As you will see, this lemma is intuitive and obvious, and hence we will not belabor the point.

Lemma 1 *The impact of an assertion on error propagation must either be: (1) negligible, or (2) positive.*

Proof. Assume that for some program P , all memory that the program has access to is contained in a ten-element array: $\mathbf{a}[0]$, $\mathbf{a}[1]$, ..., $\mathbf{a}[9]$. Assume further that some percentage x of that array, $0 < x \leq 100$, is output from P , and all information that is output from P is checked by an oracle, O . And assume that each member of $\mathbf{a}$ is only ever defined with a value once. For any element in $\mathbf{a}$, there is a probability (≥ 0) that either a fault (design error) or corrupt input will cause the element to also become corrupted; we denote these probabilities: $P_{\mathbf{a}[0]}$, $P_{\mathbf{a}[1]}$, ..., $P_{\mathbf{a}[9]}$. For example, if some element of $\mathbf{a}$ is defined in unreachable code, this probability is 0.0.

If $x = 100$, then all members of $\mathbf{a}$ are currently being checked correctness by O , and if $x < 100$, then not all members of $\mathbf{a}$ are being checked. If $x = 100$, adding an assertion to check an element $\mathbf{a}[y]$ that is already being checked will not increase the likelihood of error propagation. But if $x \neq 100$, and we assert on a member of $\mathbf{a}$ that is not being checked by O , then unless this data member is dead, the likelihood of error propagation must increase.

This is true because of the basic probabilistic laws: given two events A and B ,

$$\Pr(A) \vee \Pr(B) \geq \Pr(A)$$

$$\Pr(A) \vee \Pr(B) \geq \Pr(B)$$

In our notation, suppose that $\mathbf{a}[0]$ through $\mathbf{a}[8]$ are being tested by O ; then adding an assertion to $\mathbf{a}[9]$ cannot decrease the likelihood of error propagation, because:

$$P_{\mathbf{a}[0]} \vee P_{\mathbf{a}[1]} \vee P_{\mathbf{a}[2]} \vee P_{\mathbf{a}[3]} \vee P_{\mathbf{a}[4]} \vee P_{\mathbf{a}[5]} \vee P_{\mathbf{a}[6]} \vee P_{\mathbf{a}[7]} \vee P_{\mathbf{a}[8]} \geq$$

$$P_{\mathbf{a}[0]} \vee P_{\mathbf{a}[1]} \vee P_{\mathbf{a}[2]} \vee P_{\mathbf{a}[3]} \vee P_{\mathbf{a}[4]} \vee P_{\mathbf{a}[5]} \vee P_{\mathbf{a}[6]} \vee P_{\mathbf{a}[7]} \vee P_{\mathbf{a}[8]} \vee P_{\mathbf{a}[9]}.$$

□

We have just shown that assertions cannot decrease software testability assessments; they can only improve testability scores or have no effect whatsoever. To better understand why this occurs, consider the two main implications that the assertion, **ASSERT**($a \ b \ c$, $\mathbf{x}$, $\mathbf{a}^*\mathbf{b} - \mathbf{c}$), has when triggered:

1. If the most recent expression that assigned $\mathbf{x}$ has not assigned it a value of $\mathbf{a}^*\mathbf{b} - \mathbf{c}$ (for the most recently assigned values of a , b , and c), then the assertion will trigger and return a message that it failed; this *may* be because the expression being used to calculate $\mathbf{x}$ is incorrect, or
2. If the expression that assigns $\mathbf{x}$ does not assign it a value that is equal to $\mathbf{a}^*\mathbf{b} - \mathbf{c}$, then this may mean that some combination of the values referenced in that expression contain incorrect values. For debugging purposes, analysis both of the expression and the calculations for the referenced variables should be performed. This provides a way of partially checking the “goodness” of the state coming into that expression, i.e., a way of testing for whether an incorrect data state has propagated to the point in the program state where the assertion is called.

3.4 Verifying the Value-Added by Assertions

In the C++ ATM code, we have identified eight locations in the source code that are of particularly low testability. Although the procedural code did have low testability locations, we are interested in showing improved OO code testability, and so we did not inject assertions to that code. (We expect similar gains in testability had we applied assertions to the procedural code.)

For each of the low testability locations in the C++ version, an assertion was manually placed immediately following the location, and the testability analysis was rerun. Here, we did not assume that the code was correct, but we believe that the assertions were correct. Realize that after an assertion is injected, it then is a location that contributes to the output space, and hence the functional definition of what constitutes failure for the system is also modified. These assertions forced each propagation point estimate to increase to 1.0, which is a remarkable increase in the testability of the code with respect to the 102 test cases (See Table 7).

Code	Original Propagation	Propagation After Assertion
<code>rec->type = ret_val;</code>	0.00	1.0
<code>rec->transaction = WITHDRAW;</code>	0.095	1.0
<code>rec->type = ret_val;</code>	0.00	1.0
<code>rec->transaction = DEPOSIT;</code>	0.00	1.0
<code>ret_val = CHECKING;</code>	0.00	1.0
<code>rec->type = ret_val;</code>	0.00	1.0
<code>RecordNumber = 0;</code>	.156	1.0
<code>RECORDMAX = 30;</code>	0.00	1.0

Table 7 The “before and after” propagation point estimate location scores for OO-ATM.

4 CONCLUSIONS

It has been conjectured that there may be a theoretical upper bound on the testability achievable given a particular mathematical function that we wish to compute (Voas, 1991). It is unknown whether such exists, nor whether the upper bound is computable. However what it does suggest is that various program designs, implementations, and languages all contribute to the resulting testability given a fixed input distribution. Thus as a side- effect of this research, we hope to better determine whether the existence of such a theoretical upper bound is a realistic conjecture.

Assertions represent a valuable design tool for OO systems. We have produced a small lemma showing that an assertion can *never* decrease the propagation condition, i.e., an assertion can only improve or make no change to the propagation condition, which is very important for thwarting the negative impact of information hiding. Our conclusion that assertions are beneficial to testing parallels the recommendations of Osterweil and Clarke in (Osterweil and Clarke, 1992), where they classified assertions as “among the most significant ideas by testing and analysis researchers.” This conclusion is also confirmed in (Yin and Bieman, 1994) and (Mueller and Hoshizaki, 1994). Propagation estimates are a means for logically deciding where assertions are needed in code regions that seem unlikely to reveal faults during testing. Current schemes for the placement of assertions are often either *ad hoc* or brute-force, placing assertions in random places or everywhere.

Intrusive assertions have costs: (1) a decrease in performance during testing, and (2) the cost of deriving assertions from the specification. Also, if the assertions are removed before the code is deployed, there will be a slight, additional cost. But for critical systems, if a value-added benefit can be demonstrated relative to cost for a scheme, the scheme cannot be automatically dismissed. Given the results of Table 7, we recommended applying them.

5 REFERENCES

- Yin, H. and Bieman, J.M. (1994) Improving software testability with assertion insertion. In *Proc. of International Test Conference*, October.
- Binder, R. V. (1994) Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, September.
- Osterweil, L. and Clarke, L. (1992) A Proposed Testing and Analysis Research Initiative. *IEEE Software*, pp. 89–96, September.
- Dahl, O.J., Dijkstra, E. W., and Hoare, C.A.R. (1972) *Structured Programming*. Academic Press.
- Mueller, B.A. and Hoshizaki, D.O. (1994) Using Semantic Assertion Technology to Test Application Software. In *Proceedings of Quality Week'94*, May.
- Howden, W. E. and Huang, Y. (1993) Analysis of Testing Methods Using Failure Rate and Testability Models. Technical Report CS93-296, University of California at San Diego, June.
- Voas, J., Morell, L., and Miller, K. (1991) Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2):41–48, March.
- Voas, J. and Miller, K. (1994) Putting Assertions in Their Place. In *Proc. of the International Symposium on Software Reliability Engineering*, November, IEEE Computer Society Press.
- Voas, J. and Miller, K. (1995) Software Testability: The New Verification. *IEEE Software*, 12(3):17–28, May 1995.
- Voas, J. (1991) Factors That Affect Program Testabilities. In *Proc. of the 9th Pacific Northwest Software Quality Conf.*, pp. 235–247, Portland, OR, October, Pacific Northwest Software Quality Conference, Inc.
- Voas, J. (1992) *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, 18(8):717–727, August 1992.

6 BIOGRAPHY

Jeffrey Voas is the Vice-President of Reliable Software Technologies (RST) Corp. and heads the research initiatives of the company. Voas is currently the principal investigator on research initiatives for NASA, National Institute of Standards and Technology, U.S. Air Force, and National Science Foundation. He has published over 60 journal and conference papers in the areas of software testability, software reliability, debugging, safety, fault-tolerance, design, and computer security. Voas has coauthored a text entitled Software Assessment: Reliability, Safety, Testability (John Wiley & Sons, 1995). In 1994, the Journal of Systems and Software ranked Voas 6th among the 15 top scholars in Systems and Software Engineering.