

# Software Testability Measurement for Assertion Placement and Fault Localization

J. M. Voas

Reliable Software Technologies, Sterling, VA 20166 USA

**Abstract.** Software testability, the tendency for software to reveal its faults during testing, is an important issue for verification and quality assurance. Testability measurement can also be used to good advantage as a debugging aide. In this paper, we propose using testability measures for assertion placement and fault isolation. One measure of testability is a technique termed “Sensitivity Analysis.” This technique analyzes how likely a test scheme is to (1) propagate data state errors to the output space, (2) cause internal states to become corrupted when faults are exercised, and (3) exercise the code. By knowing where “small” faults are likely to hide for a particular test scheme, we have insight into where assertions are warranted and particularly beneficial. Even without hints from assertions as to where a fault might be resident, sensitivity analysis and a rough failure probability estimate (for a program during test or operation) provide enough information to formalize a testability-based debugging paradigm that can be used to identify possible fault sites. This model works well when hiding faults are of small size and causing infrequent failure.

## 1 Introduction

Software testing is performed to satisfy one of two goals: (1) to detect and then remove faults, and (2) to estimate the reliability of the code. Testing is usually only considered effective when it uncovers faults. Testing is often considered ineffective when it does not reveal existing faults, the effects of which may be significant and dangerous. In this situation, faults will remain to surface after the software is delivered. Since debugging is typically invoked after failure occurs, test schemes that have the greatest ability to produce failures (when faults are present) are preferable.

Our research has previously focused on *how* faults “hide” from testing.<sup>1</sup> Voas defines testability of a program  $P$  to be the probability that a particular testing strategy will force failures to be *observable* if faults were to exist. In contrast, *software detectability* of a program  $P$  is the probability that a particular testing strategy will force failures to be *observable* if faults were to exist and the probability that the *oracle* will be precise enough to detect failures. In this paper, we use an implementation of Voas’s testability definition [12] to identify *where* faults (if they exist) are likely to hide [12, 9, 6]. Low testability source locations are statements where it is believed that “small” faults will be difficult to detect during testing. We thwart the potential error masking at low testability locations through strategic assertion placement; this will serve to improve the efficiency of fault isolation. We will also demonstrate how

---

<sup>1</sup> This is traditionally referred to as “error masking.”

testability measurements can enhance automated debugging when assertions are not violated or used.

## 2 Assertions

Run-time assertion checking is a programming-for-validation “trick” that can help insure that a program satisfies certain semantic constraints. The wide-spread appeal of assertions can be seen recently in languages such as Anna [3] and Eiffel [5]. Anna (Annotated Ada) use comments to embed assertions; Eiffel uses object invariants that are inserted as pre- and post-conditions to all operations on the object. Blum and Raghavan have investigated *program correctness checkers* and shown that in many cases, it is possible to check a program’s output for an input [10]; by allowing for the possibility of incorrect results, the program designer is forced to confront the possibility of faults and what to do if the program does enter into an incorrect state. Designing programs to correct themselves provides an alternative to proving absolute program correctness.

Our interest in assertions is not whether languages or programming paradigms support their use, but rather how to embed assertions in a manner that has the greatest fault revealing ability. The conjecture that has motivated this paper follows:

Why place assertions on portions of the state if it is known *a priori* that if these portions of the state are in error, failure is nearly guaranteed to occur. Instead, place assertions on portions of the state when it is likely that incorrectness in those portions of the state will not be observable at the program exit point.

Assertions test the correctness of the entire program state or they test some portion of the program state. In *proofs of correctness*, the entire program state is demonstrated as satisfying certain logical constraints. Typically, software testing only checks the correctness of data states that are output. In contrast, assertions check “intermediate data values” that occur during an execution. A benefit of checking intermediate data values is that we know immediately whether the program has entered into an erroneous state.

We will assume that assertions evaluate to TRUE when the tested internal state is satisfactory, and FALSE otherwise. If an assertion evaluates to FALSE, then we consider the execution of the program to have resulted in failure, even if the eventual output is correct according to the specification. We modify what we consider as failure in the output space: a *failure* occurs if the output is incorrect *or* an assertion fails. This not only modifies what is considered failure, but it also modifies what is considered output.

“Observability” is a term used in hardware design that means the ability to detect problems in the inner logic of a chip. Hardware probes are placed into circuits to increase the observability of the circuit during test. Similarly, assertions increase observability in software by increasing the dimensionality (and/or cardinality) of the output space of the software. After testing a program  $N$  times and not observing failure (with the assertions in), we gain a confidence that faults are not hiding.

Once testing is completed, the assertions may be removed if the development team chooses.<sup>2</sup>

This paper advocates a middle ground between no assertions at all (the most common practice) and the theoretical ideal of assertions at every location. Our compromise is to place assertions only at locations where traditional testing is unlikely to uncover software faults. One type of testability measurement, *sensitivity analysis*, identifies the locations where the current testing scheme is unlikely to be effective. Assertions that will be placed in the code will be a function of sensitivity analysis. Sensitivity analysis (See Section 2.1) is a function of the testing scheme. What has surprised us from this research is that the assertions that are based on deficits in testing scheme *A* appear to still be valid at improving the fault-detection of testing scheme *B*.<sup>3</sup> Although this has not been formally nor empirically substantiated, if generally true, it suggests that current research into which testing scheme is better may be wasted effort; possibly any reasonable testing scheme can be massaged into an excellent fault-detection capability with plausible assertion placement schemes.

## 2.1 Sensitivity Analysis

Sensitivity Analysis is composed of three subanalyses: (1) one to measure the probability that the test distribution can actually reach particular locations, (2) one to measure the probability that incorrect locations cause data states to be corrupted, and (3) one to measure how likely corrupt data states are to propagate. Each of these analysis produces a point estimate in  $[0,1]$ , and the testability of a location,  $\theta_l$  is also in  $[0,1]$ . Here, we will concentrate on two of these: Infection analysis and Propagation analysis.<sup>4</sup>

*Infection analysis* estimates the probability that a *mutant* at a particular location will adversely affect the data state which immediately results when the location is executed. In other words, will the mutant produce a value in the following data state that is different than the value that is produced by the original location?

Infection analysis is similar to mutation testing [11]; what is different is the information collected. For a given location in a program, we do not know whether or not a fault is present, and we don't know what types of faults are possible at the location. So we create a set of mutants at each location. After creating a set of mutants, we obtain an estimate of the probability that the data state is affected by the presence of a mutant for each mutant in the set. We select a mutant from the set, mutate the code at the location, and execute each resulting mutant many times. The data states created by executing the mutants are checked against the data states from the original location to determine if the mutants have *infected* the data states. The proportion of executions that infect for a particular mutant are the *infection estimate* for that mutant.

<sup>2</sup> The removal of assertions is motivated by the desire for more efficient execution during production runs.

<sup>3</sup> One notable exception here is the use of assertions to thwart fault hiding of infrequently exercised code. because of the testing scheme

<sup>4</sup> There is no reason to concern ourselves with the analysis that estimates (1), because assertions in regions that are almost never executed by the testing scheme will themselves never get executed.

The algorithm for finding an infection estimate is:

1. Set variable **count** to 0,
2. Create a mutant for location  $l$  denoted as  $h$ ,
3. Present the original location  $l$  and the mutant  $h$  with a randomly selected data state from the set of data states that occur immediately prior to location  $l$  and execute both locations in parallel,
4. Compare the resulting data states and increment **count** when the function computed by  $h$  does not equal the function computed by  $l$  for this particular data state,
5. Repeat algorithm steps 3 and 4  $n$  times,
6. Divide **count** by  $n$  yielding an  $\hat{\lambda}_{h|PD}$

*Propagation analysis* estimates the probability that an infected data state at a location will propagate to the output. To make this estimate, we repeatedly *perturb* the data state which occurs after some location of a program, changing one value in the data state (hence one *live* variable receives an altered value) for each execution. We term a variable as being live at a particular location of a program if the variable has any potential of affecting the output of the program. For instance, a variable which is defined but never referenced is one example of a variable which would not be live. By examining how often a forced change into a data state affects the output, we calculate a *propagation estimate*, which is an estimate of the affect that a live variable (the variable that received the forced change into the data state) has on the program's output at this location. We find a propagation estimate for a set of live variables at each location (assuming there is more than one live variable at a location), thus producing a set of propagation estimates—one propagation estimate per live variable.

If we were to find that at a particular location a particular live variable had a near 0.0 propagation estimate, we would realize that this variable had virtually no affect on the output of the program at this location. This does not necessarily mean that this variable has no affect on the output—only that it has little effect. Hence propagation analysis is concerned with the likelihood that a particular live variable at a particular location will cause the output to differ after the live variable's value is changed in the location's data state.

Propagation analysis is based on changes to the data state. To obtain the data states that are then executed to completion, we use a mathematical function based upon a random distribution termed a *perturbation function*. A perturbation function inputs a variable's value and produces a different value chosen according to the random distribution—the random distribution uses the original value as a parameter to in when producing the different value.

An algorithm for finding a propagation estimate is:

1. Set variable **count** to 0,
2. Randomly select a data state from the distribution of data states that occur after location  $l$  (that are function of program  $P$ 's input distribution  $D$ ),
3. Perturb the sampled value of variable  $a$  in this data state if  $a$  is defined, else assign  $a$  a random value, and execute the succeeding code on both the perturbed and original data states,

4. For each different outcome in the output between the perturbed data state and the original data state, increment `count`; increment `count` if an infinite loop occurs (set a time limit for termination, and if execution is not finished in that time interval, assume an infinite loop occurred),
5. Repeat algorithm steps 2-4  $n$  times,
6. Divide `count` by  $n$  yielding  $\hat{\psi}_{alPD}$ .

## 2.2 Locations Needing Assertions

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 internal instrumentation will be slower than the same program without the instrumentation. Some of the assertions may be redundant. The task of instrumenting the code with correct assertions at each location is burdensome, and there is no guarantee that the assertions themselves will be correct.

Let  $L$  represent the set of all locations (statements) in a program,  $P$ , and suppose that propagation and infection analyses [12, 6] have been performed at each member of  $L$ . Let  $\Theta_l$  be the set of non-zero infection estimates and the propagation estimate for the variable on the left hand side of some location,  $l$ ;<sup>5</sup>  $\Theta_l$  does not contain  $l$ 's estimate of the likelihood of reaching  $l$ . Now let  $\theta_l = \min[\Theta_l]$ . The information produced by sensitivity analysis can be considered as an assessment of the likelihood that there is a fault hiding in  $l$  affecting the variable assigned at  $l$  given a program testing scheme,  $D$ .

By viewing a location as only affecting one variable of the data state (i.e., no side-effects), we can rank all locations in the program according to the  $\theta_l$ s. This ordering provides knowledge as to (1) which locations are likely to hide faults during testing, and (2) where assertions can be cost-effectively employed. If the propagation estimate dominates  $\theta_l$ , then there is some later location that may mask problems at  $l$ , and hence asserting on  $l$  allows us to be less concerned with such masking. If an infection estimate dominates, then an assertion decreases the likelihood that  $l$  will hide a fault.

To rank locations, we first establish a cut-off score,  $\varepsilon$ , such that locations with scores below the cut-off are judged to be dangerously insensitive to faults. The cut-off score marks the fault size that we consider to be too problematic to simply test for. For a variable assigned a value at one of these dangerous locations, we will place an assertion at that location. The assertion is devised to reflect a required state of the computation of that location; this information must be extracted from the specification. Our placement recommendation: *create assertions for all locations where  $\theta_l < \varepsilon$ , and place these assertions immediately after  $l$*  (this set of locations is denoted by  $L'$ ).

An *assertion* is a self-test that the software performs during execution. For example, a software assertion might look like `ASSERT(program_input_values, y, z+x-4)`, which passes the code input values and the variable being tested,  $y$ , into a procedure that returns TRUE if  $y$  is equal to  $z+x-4$  and FALSE if it is not. The value-added by inserting assertions into software is directly dependent on two factors:

<sup>5</sup> We only use non-zero infection estimates because we do not decide mutant equivalence.

1. Is the assertion correct? and
2. How “tight” is the assertion?

We say an assertion is *tight* if we are able to determine whether the value being tested is correct. An assertion is tight with respect to the value that the variable should have at that location in the code. Finding a tight assertion is, in general, nontrivial. The opposite of a tight assertion is a *loose* assertion that allows a variable to have a less well specified value; such assertions reveal less about the internal state of the computation.

Our plan for using assertions is simple:

1. Perform propagation and infection analysis on the original code.
2. Place assertions where warranted by the  $\theta_l$  rankings.
3. Test the code with the assertions in place.

There are several things to note: (1) you can place assertions at locations that are infrequently executed, but realize that those assertions may not get exercised until the code is released (assuming that they are left in). (2) Assertions have the greatest impact on propagation estimates, because when they test a portion of the program state, that data value tested is a function of other data values (which, if they are corrupt, are more likely to be exposed because of the assertion).

When testing is complete and the software is being released, the assertions may be removed. Executing software assertions decreases efficiency. Removing assertions after testing is analogous to compiling without the debug flag when we are no longer experiencing run-time errors. Assertions remaining in production software can be useful in detecting and diagnosing problems.

### 2.3 Incorrect Assertions

Deriving *correct* assertions is no trivial exercise; recognize that for any specification, there are an infinite number of correct programs that implement the desired functionality, and an infinite number of incorrect programs that are slight mutations of each correct program. Thus, the process of assertion placement is version specific, i.e., you must consider the version before injecting assertions. The rule of thumb: *one assertion may not fit all!* This fact highlights the difficulty of placing correct assertions in the correct place. Even if assertions are incorrect, they are likely to benefit assessed testabilities. But our goal is not merely to use assertions to increase testability, but rather to use assertions as windows into the computational state. Only correct assertions are useful to debugging.

Our testability-based assertion placement method combines formal mechanisms, empirical testing, and empirical testability analysis. When assertions are correct, they can deliver information both about testability and correctness. The correctness requirement for assertions is costly, but the need for a correct oracle has always been a limitation of testing techniques [4]. It is not reasonable to simply insert assertions and claim that the overall testability is increased; the assertions must also be an aide to testing and debugging.

## 2.4 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:** 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 ( $\geq 0$ ) 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.

## 2.5 Intrusive vs. Non-Intrusive Assertions

One common criticism against software assertions is that they are usually implemented via intrusive instrumentation. The reason for doing this is that it is far easier (meaning cheaper) to code assertions in the program language than it is to use a separate hardware processor (executing the assertions). When a separate processor is used, it probes the variable values in memory written to by the running program and executes the assertions at the appropriate times. But none the less, non-intrusive assertions are possible, particularly for applications that are timing sensitive. The point here is that assertions are not necessarily intrusive, but for practical purposes, they usually are. Assertions can be implemented in both an intrusive and non-intrusive environment; our current method of implementing the testability measurement is intrusive.

## 2.6 Experiment Using Testability-Guided Assertion Placement in Object-Oriented Study

We now summarize the results obtained by performing testability analysis on an object-oriented Automated Teller Machine (ATM) simulation [2]. The ATM system was coded in C++ from a simple ASCII specification. In this experiment, we were focused solely on using assertions to decrease the negative impact of encapsulation and information hiding on system level testing. 102 system level test cases were developed such that all locations in the program were exercised.

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 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.

Inputs to the program take the form:

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

In the C++ ATM code, we identified eight locations in the C++ code that are of particularly low testability (See Table 1 for the statement at the location of concern and the testability score). For each of the low testability locations, an assertion was manually placed immediately following the location, and the testability analysis was rerun. Here, we did not assume that the assertions were correct, but we believe that they were. These assertions have forced each testability score 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 1).

Statement	Testability Before Assertion	Testability After Assertion
<code>rec-&gt;type = ret_val;</code>	0.00	1.0
<code>rec-&gt;transaction = WITHDRAW;</code>	0.095	1.0
<code>rec-&gt;type = ret_val;</code>	0.00	1.0
<code>rec-&gt;transaction = DEPOSIT;</code>	0.00	1.0
<code>ret_val = CHECKING;</code>	0.00	1.0
<code>rec-&gt;type = ret_val;</code>	0.00	1.0
<code>RecordNumber = 0;</code>	.156	1.0
<code>RECORDMAX = 30;</code>	0.00	1.0

Table 1. The “before and after assertion” testability scores.

This experiment has demonstrated that assertions increase propagation (and hence testability) as the lemma showed. Not only did the assertions increase propagation point estimates, but to 1.0, which is the maximum score.

### 3 A Coarse Model for Applying Testability To Fault Localization

Even with assertions, a possibility exists that no assertions will fail even though the program fails. We now describe a scheme that uses the sensitivity analysis information (that guided the placement of assertions) to help isolate where small faults might be hiding even though no assertions were triggered; this was first introduced in [7].

If testability analysis correctly estimates the impact that mutated data states have on program output, and if these estimates can be correlated to impacts on the output caused by real programmer faults, then this information can be used in a semi-automated process for suggesting where faults might be occurring. Note that the debugging model that we are about to describe does not suggest what the fault might be, because the testability technique has no information concerning what or is not correct. This debugging paradigm using information concerning where faults are more likely to hide only suggests where faults of small size are likely to hide. This is very similar to the concept of applying software *slicing* to software *maintenance* [13, 14, 15].

Using testability measurements in a debugging capacity after a failure or failures have occurred requires the following information:

1.  $\hat{\tau}_{PD}$ , the *failure probability point estimate* for program  $P$  given input distribution  $D$ .
2.  $\hat{\epsilon}_{lPD}$ , an estimate of the probability that location  $l$  is exercised by program  $P$ 's input distribution,  $D$ , for each  $l$ .
3.  $\hat{\psi}_{aIPD}$ , the propagation estimate for each  $a$  and each  $l$ .

The method for using testability measures to predict whether or not a fault might be residing in location  $l$  and affecting variable  $a$  is provided in equation 1 and equation 2:

$$(\hat{\epsilon}_{lPD}) \cdot (\hat{\psi}_{aIPD}) > (\hat{\tau}_{PD}) \longrightarrow$$

no fault corrupting variable  $a$  at location  $l$  (1)

suggests that an assignment statement is *not* missing at location  $l$  that assigns a value to variable  $a$ .

$$(\hat{\epsilon}_{lPD}) \cdot (\hat{\psi}_{aIPD}) \leq (\hat{\tau}_{PD}) \longrightarrow$$

potential fault corrupting variable  $a$  at location  $l$  (2)

suggests that a fault *is* at location  $l$  that is corrupting variable  $a$ . Note that these equations do not say that a fault is or is not occurring; rather they say that the impact of a corrupted data state on the output is curiously similar to the rate at which failures are occurring.

This debugging model can be a particularly useful when  $\hat{\tau}_{PD}$  is small ( $< 10^{-4}$ ). The model admittedly has limited debugging value in the following situations:

1. The program is failing because of complex fault classes such as distributed faults: their effect on a data state is not currently simulated by sensitivity analysis.
2. Large failure probability estimates: they force virtually all locations to be suspects,
3. The program is failing because of many different faults combining into a greater overall failure probability. This debugging model will thus suggest more locations as potentially containing a fault, making the model useless.
4. The hiding faults are causing infrequent failures because the faults infrequently corrupt the program state; we currently do not have a plausible solution for this.
5. The input distribution used during sensitivity analysis does not match the distribution used during debugging.

Results using this model can be found in [7].

## 4 Summary

Software assertions are particularly useful during debugging; this paper has presented a scheme for identifying those regions of the code that if incorrect, are likely to hide “tiny” faults. Armed with this information, assertions can be carefully placed to increase those fault sizes, i.e., increase the fault-revealing ability of the testing scheme. We have presented a means for better deciding where assertions are needed for software systems that suffer from code regions that seem unlikely to propagate data state errors during testing; recently, others have published similar findings as

to benefit of assertions on testability [1], but without providing guidance on where the assertions should be inserted. Current schemes for the placement of assertions are often either *ad hoc* or brute-force, placing assertions everywhere.

This testability-based assertion placement method will improve the fault-detection capability of the current testing scheme, by adding fault-detection support to those portions of the code that appear to be “invincible” to this testing scheme. Thus not only do we gain confidence from an assertion that the location receiving the assertion is not hiding faults, but we gain confidence that faults are not hiding elsewhere in the code. This scheme has costs; those costs include: (1) the decrease in performance during testing, (2) the costs of performing testability analysis, and (3) 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. By combining white-box analysis via testability analysis, black-box analysis via testing, and specification-based testing via assertions, we may be able to more quickly debug systems in a systematic and automated manner.

This paper has also shown that even if assertions do not detect internal problems, the testability predictions used to guide assertion placement can sometimes be correlated with the probability of failure to suggest suspicious locations. In this manner, software testability information can be of limited help to debuggers when a program is experiencing infrequent failures. This debugging model could be greatly enhanced if it were combined with a trace of which locations previous inputs exercised that resulted in failure.

## Acknowledgement

This paper is an expanded version of [8]. This research was partially funded by NIST Contract 50-DKNA-4-00119.

## References

1. H. YIN AND J.M. BIEMAN. Improving software testability with assertion insertion. In *Proc. of International Test Conference*, October 1994.
2. RELIABLE SOFTWARE TECHNOLOGIES CORPORATION. Testability of Object-Oriented Systems . Technical report, Sterling, Virginia, December 1994. Final Report for NIST Contract 50-DKNA-4-00119.
3. D. LUCKHAM AND F. VON HENKE. An overview of ANNA, a specification language for Ada. *IEEE Software*, pages 9–22, March 1985.
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. B. MEYER. *Eiffel the Language*. Prentice-Hall, 1992.
6. J. VOAS, L. MORELL, AND K. MILLER. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2):41–48, March 1991.
7. J. VOAS AND K. MILLER. Applying A Dynamic Testability Technique To Debugging Certain Classes of Software Faults. *The Software Quality Journal*, 2:61–75, 1993.

8. J. VOAS AND K. MILLER. Putting Assertions in Their Place. In *Proc. of the International Symposium on Software Reliability Engineering*, Monterey, CA, November 1994. IEEE Computer Society Press.
9. J. VOAS AND K. MILLER. Software Testability: The New Verification. *IEEE Software*, 12(3):17–28, May 1995.
10. M. BLUM AND P. RAGHAVAN. Program Correctness: Can one Test for it? Technical Report Report RC 14038 (#62902), IBM T.J. Watson Research Center, September 1988.
11. 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.
12. J. VOAS. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, 18(8):717–727, August 1992.
13. J. LYLE AND M. WEISER. Experiments on slicing-based debugging tools. In *Proceedings of 1st Conf. on Empirical Studies of Programming*, pages 187–197, June 1986.
14. J. LYLE AND M. WEISER. Automatic program pug location by program slicing. In *Proceedings of 2nd IEEE Symposium on Computers and Applications*, pages 887–883, Beijing, China, June 1987.
15. M. WEISER. Programmers use slices when debugging. *CACM*, 25(7):446–452, July 1982.