

Inspecting and ASSERTing Intelligently

J. Voas
RST Corporation
Suite 250, 21515 Ridgetop
Sterling, VA 20166
jmvoas@testability.RSTcorp.com

K. Miller
c/o Computer Science
U. of Illinois at Springfield
Springfield, IL 62794
miller@uis.edu

Abstract

Software testability is a misunderstood technology by most of the software testing community. Most testing experts believe that software testability is either a measure of complexity or a subjective assessment of how much it will cost to generate test cases, perform coverage testing, write drivers and/or stubs, etc. This perspective is incomplete. In the late 1980s, an alternative perspective on testability was proposed that attempted to answer the following question: “what is the probability that a program cannot fail even if it is incorrect.” This suggestion was in sharp contrast to the age old reliability question: “what is the probability that a program will fail”, and many experts failed to immediately pick up on the subtle difference. The difference represented a paradigm shift for software testing research, and provided new avenues for research that have already yielded unexpected benefits. This paper focuses on one of those benefits: deciding where to ASSERT and inspect code.

1 Introduction

Software testing is performed for one of two reasons: (1) to detect the existence of faults, and (2) to estimate the “goodness”, i.e., the reliability of the code. Testing is considered effective when it uncovers faults, and is often considered ineffective when no failures occur. Residual software faults can have significant and dangerous consequences after the software is released. Since debugging to improve the reliability of the code is effective only after failure is observed, test schemes that have the greatest ability to produce failures (when faults are present) are of paramount importance. This paper presents a methodology that can be employed by testers and developers to decrease the likelihood of residual faults.

For over 8 years now, our research has focused on investigating *how* faults “hide” from testing and then decreasing this possibility.¹ From a “testing for defect detection” standpoint, hiding faults are the number one concern. There are, however, classes of software

¹This is often referred to as “error masking.”

systems for which error masking in the released version is sought, specifically, *high integrity systems*. For this type of system, programs that hide defects during operation are considered as *robust*, and their enhanced error-masking capabilities are heralded as “desirable.” During validation, however, defect detection is the more sought characteristic, and in this paper, we will overview one method for engendering it. Interestingly enough, the defect-detection methods that we will recommend can be deactivated during software usage, thus providing defect-detection when it is desired, and providing robustness when it is not.

There are many perspectives on software testability. We define the 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 [8]. Other definitions look at testability as a measure of testing effort. Similar to our definition of testability, *software detectability* is defined as the probability that a particular testing strategy will force failures to be *observable* if faults were to exist and combined with the probability that the *oracle* will be precise enough to detect failures [1]. Detectability is a *reliability* accuracy concern; testability is not.

Our research has used one specific implementation of the testability definition in [8] called “*Sensitivity Analysis*” to identify *where* existing faults are more likely to hide. Low testability source locations are statements where it is believed that “small” faults will be difficult to detect during testing. These are portions of the code where residual faults are likely to exist after code release. In the upcoming sections, we will recommend thwarting the potential error masking of low testability locations through strategic assertion placement and/or software inspections; this will improve the efficiency of defect-detection which in turn should improve the *true* reliability of the code.²

2 Sensitivity Analysis

Sensitivity analysis identifies code regions (at different levels of granularity, *e.g.*, statements, functions, modules, units, blocks, etc.) where the current testing scheme is unlikely to be effective at defect-detection. Inspections and assertions can be more intelligently applied to these areas after the results of sensitivity analysis are generated.

Sensitivity Analysis is a technique composed of three subanalyses: (1) a *reachability* analysis to measure the probability that the test distribution can actually reach particular locations (Execution Analysis), (2) a fault-injection–based technique to measure the probability that incorrect locations cause data states to be corrupted (Infection Analysis), and (3) a fault-injection–based technique to measure how likely corrupt data states are to propagate (Propagation Analysis). Each of these techniques are based on the *fault/failure* model [8]. Each of the three analyses of sensitivity analysis produces a point estimate in $[0,1]$. Sensitivity analysis is a dynamic technique that executes the code over and over with the testing scheme to assess where the scheme appears deficient.

Our approach for where to ASSERT and inspect code is directly based on only one

²We say “true” reliability as opposed to “estimated” or “predicted” reliability, because these can be very different if the metric/model employed is inaccurate.

of these three analyses: Propagation analysis for assertion placement. There is, however, good reason to perform Execution analysis at the outset, because placing assertions in regions that are almost never executed by the testing scheme will themselves never get executed. Locations in the code that are not executed during testing need a better code coverage testing scheme and/or inspections.

Propagation analysis has demonstrated meaningful results on a variety of control applications, and hence propagation analysis is pivotal for our methodology [7, 9, 6]. Our *propagation analysis* is a fault-injection technique that estimates the probability that an infected data state at a statement will propagate to the output. To make this estimate, propagation analysis repeatedly *perturbs* the data state that occurs after some statement of a program, changing one value in the data state (hence one variable receives an altered value) for each execution. 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 variable (the variable that received the forced change into the data state) has on the program’s output at this statement. We find a propagation estimate for the variable being assigned at a statement, by injecting corruption into that variable immediately after it is assigned its value at the statement. This simulates the statement being faulty and corrupting the data state value.

If we were to find that at a particular statement a particular 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 statement. 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 variable at a particular statement will cause the output to differ after the variable’s value is changed in the statement’s data state.

3 Intelligently Localizing Where to Insert Assertions and/or Perform Inspections

Run-time assertion checking is a programming-for-validation trick that helps insure that a program satisfies certain semantic constraints. “Intelligent,” strategic run-time assertion checking presents the opportunity to thwart error masking at a more reasonable cost than *ad hoc* assertion placement. Code inspections have a similar impact as assertions; a different process is used for implementing them. Any time that assertions cannot be employed due to performance reasons, inspections can be substituted.

Assertions test the *correctness* (or any other semantic characteristic) of program states for individual program executions. For example, suppose that it were known the range of legal values for some intermediate computation in a program; this could be easily checked for via an assertion. In contrast, *correctness proofs* statically test that the entire program state satisfies certain logical constraints for all inputs. Dynamic software testing only checks the correctness of data states that are output for specific program executions. Loosely speaking, code inspections are a cross between static testing and manual correctness proofs.

Assertions check “intermediate data values”, and hence perform a different function than does software testing. A benefit of checking intermediate data values is that we can quickly be warned when the program has entered into an undesirable state. This is valuable information, since it is possible that the undesirable state will not propagate into a program failure, and hence the fault will not be caught during testing.

Assertions are Boolean functions that evaluate to TRUE when a program state appears to be satisfactory, and FALSE otherwise. As mentioned, satisfactory can be defined with any semantic interpretation. If an assertion evaluates to FALSE, we will consider it the same as if the execution of the program resulted in failure, even if the output for that execution is correct. Hence we will modify what we consider as failure in the output space: a *failure* will be said to have occurred 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.

Assertions that are placed at each statement in a program can automatically monitor the internal computations of a program execution. Software inspections are a manual and static counterpart of dynamic assertions. However, the advantages of placing assertions or performing inspections everywhere in a program come at a cost. A program with such extensive internal instrumentation will be substantially slower than the same program without the instrumentation. Performing detailed inspections everywhere is also quite costly. Recognize that some of the assertion’s and inspection’s net effectiveness at detecting problems may be needlessly redundant. And the task of instrumenting the code with so many assertions will be burdensome. And finally, it is foolish to assume that you can get that many assertions correct when you are incapable of getting the program itself correct.

3.1 Lemma of the Impact of Assertions on Testability

We will now give a simple lemma showing that the impact of assertions on error propagation must either be: (1) negligible, or (2) positive. Greater error propagation increases testability.

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 programmer accessible variables are contained in a ten-element array: $\mathbf{a}[0]$, $\mathbf{a}[1]$, ..., $\mathbf{a}[9]$. Some percentage, x , of that array, $0 < x \leq 100$, is output from P , where all information that is output from P is checked by an oracle, O . For any element in \mathbf{a} , there is a probability that a defect will cause the element to contain incorrect information; we denote these probabilities: $P_{\mathbf{a}[0]}$, $P_{\mathbf{a}[1]}$, ..., $P_{\mathbf{a}[9]}$. If incorrect information is asserted on or checked by O , then error propagation is 100%.

Asserting on an element of \mathbf{a} that is already being checked by O cannot decrease the likelihood of error propagation. But if $x \neq 100$, and we assert on a reachable member of \mathbf{a} that is not being checked by O , then the likelihood

of error propagation must increase, because of the basic probabilistic laws:³
Given two events A and B ,

$$\mathbf{P}(A) \text{ OR } \mathbf{P}(B) \geq \mathbf{P}(A), \text{ and}$$

$$\mathbf{P}(A) \text{ OR } \mathbf{P}(B) \geq \mathbf{P}(B).$$

For example, suppose that $\mathbf{a}[0]$ through $\mathbf{a}[1]$ are being checked by O ; then adding an assertion to $\mathbf{a}[9]$ cannot decrease the likelihood of error propagation, because:

$$P_{\mathbf{a}[0]} \text{ OR } P_{\mathbf{a}[1]} \leq \\ P_{\mathbf{a}[0]} \text{ OR } P_{\mathbf{a}[1]} \text{ OR } P_{\mathbf{a}[9]}.$$

□

4 Several Hypotheses about Assertions

In this section, we will put forth several hypotheses regarding the relationship between software assertions, test suites, and error masking ability.

4.1 Reachability

Static testability metrics cannot as easily nor accurately address software reachability as can dynamic metrics. Although collecting data about information loss is useful for assertion placement, reachability analysis is still necessary to know if the assertion will be exercised.

Let $T(P)_D$ represent the testability of program P when tested with test suite D . And let $(D \cup \Delta)$ represent test suite D after it has been augmented with additional test cases Δ such that all statements in P are exercised.⁴ Note that Δ could be empty, in which case $D = (D \cup \Delta)$. Since reachability is the first event in the fault/failure model, it is a necessary condition for improved testability, hence

$$T(P)_D \leq T(P)_{(D \cup \Delta)}.$$

4.2 Changing Test Suites

We now wish to look at what occurs to the testability of software when a test suite changes. We begin with the case where the code has no assertions in it, and then we will consider the case where assertions are instrumented.

For program P without assertions, we just showed regardless of what Δ adds to test suite D , we know that $T(P)_D \leq T(P)_{(D \cup \Delta)}$. Given a new test suite D' that is not

³A reachable data member does not exist in dead code.

⁴There are software test case generation tools (Godzilla [2], WhiteBox TGen) that are intelligent enough to find test cases that will exercise previously unreached statements in the code, but this is an unsolvable problem in general.

identical to D and neither test suite is a subset of the other, what can we say about the relation between $T(P)_D$ and $T(P)_{D'}$? Quite simply, we cannot say anything confidently without running a technique similar to sensitivity analysis with both D and D' . That is, we do not know whether $T(P)_D \leq T(P)_{D'}$ or $T(P)_D > T(P)_{D'}$.

Now let P_{A_D} represent program P with assertions which were designed to lower the error masking ability of D . Assertions increase the likelihood that the second and third conditions of the fault/failure model happen. Then according to the lemma,

$$T(P)_D \leq T(P_{A_D})_D.$$

From there, we will conjecture that:

$$T(P)_{D'} \leq T(P_{A_D})_{D'}, \tag{1}$$

regardless of if D and D' have common members. Our argument in support of this conjecture follows: if the assertions placed into P (that are based on D) are never exercised via inputs from D' , then $T(P)_{D'} = T(P_{A_D})_{D'}$; if as little as one assertion is exercised by some input in D' , then by the lemma it is possible that $T(P)_{D'} \leq T(P_{A_D})_{D'}$ will be true.

Another recent research result that supports our conjecture has been found by Michael [4]. Michael has been studying a phenomenon he terms “homogeneous propagation.” When several different forms of program state corruption impact the same program variable (at the same place in the code) for the same program input, and all of these corruptions exhibit the identical propagation behavior, *homogeneous propagation* has occurred. That means that either all of the corruptions propagated, or none of them did. If some propagate and some don’t, we call it *nonhomogeneous propagation*. Recent research results by Michael [4] tested homogeneous propagation on 3 programs: a backpropagation network, a module from an autopilot, and a large computer game (nethack). In each program, 20 data-state perturbations were made of each variable that appeared on the left-hand side of an assignment statement. This was done for one execution of the backpropagation network, 5 executions of nethack, and 500 executions of the autopilot. In the backpropagation program, homogeneous propagation occurred for 84% of the variables. For nethack, 84% of the variables showed homogeneous propagation on all five test runs. For the autopilot, 46% of the variables showed homogeneous propagation on all 500 test runs, and another 37% showed homogeneous propagation on at least 95% of the test runs. Thus, only about 5% of the variables showed homogeneous propagation on fewer than 95% of the test runs.

What this suggests about assertions is straightforward. Since assertions directly affect propagation, if propagation is homogeneous for some $i \in D$, then it is likely that it will be homogeneous for some other $j \in D$. Whether j is in D or D' is immaterial. In general then, assertions appear to have an impact on testability that is without respect for whether the inputs to the software are from one test suite or another. Hence since assertions improve the propagation prospects for inputs from D by increasing the cardinality or dimensionality of the output space, they should also improve the propagation prospects for D' .

And finally, we conjecture that it will generally be true that:

$$T(P)_D \ll T(P_{A_D})_{(D \cup \Delta)} \tag{2}$$

However there will be cases where

$$T(P)_D = T(P_{A_D})_{(D \cup \Delta)},$$

specifically when $D = (D \cup \Delta)$ and/or the assertions placed into P are not exercised.

5 Summary

Our conclusion that assertions are beneficial for testability parallels the comments by Osterweil and Clarke concerning the value of assertions to testing; in 1992, they classified assertions as “among the most significant ideas by testing and analysis researchers” [5]. Because of our previous work with the fault/failure model, we believe we have provided insights into why assertions work well and how their placement can be made more systematic.

Software assertions and code inspections are key weapons in an V & V arsenal; this paper has argued for making the employment of these techniques more rigorous and justified. We recommend employing these processes for those code regions that if incorrect, are likely to mask “smaller” faults during testing. Armed with this information, validation resources can be carefully placed to increase those fault sizes, which in turn decreases the error-masking ability of the testing scheme.

Our scheme incurs the following costs when assertions are used: (1) a decrease in software performance, (2) the costs of sensitivity analysis, and (3) the costs of deriving assertions. But by combining white-box sensitivity analysis, black-box testing, and “quasi” specification-based testing via assertions, we clearly provide a more intelligent way to validate software than by testing alone. And for many systems, the importance of releasing software with as few faults as possible is greater than the costs of isolating where assertions are needed and implementing them.

When inspections are used, the costs involve: (1) performing sensitivity analysis, and (2) the manual costs of sifting through the code and comparing it to the requirements and specification. Note that the software inspection process is not foolproof, and since it is manual, it can be as easily flawed as any other task during development. For this reason, it is recommended that any inspections performed follow the best recommendations provided in [3].

References

- [1] A. BERTOLINO AND L. STRIGINI. On the use of testability measures for dependability assessment. *IEEE Trans. on Software Engineering*, 22(2):97–108, August 1995.
- [2] R. A. DEMILLO AND A. J. OFFUTT. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [3] T. GILB AND D. GRAHAM. *Software Inspections*. Addison-Wesley, 1993.

- [4] 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.
- [5] L. OSTERWEIL AND L. CLARKE. A Proposed Testing and Analysis Research Initiative. *IEEE Software*, pages 89–96, September 1992.
- [6] J. VOAS, F. CHARRON, G. MCGRAW, K. MILLER, AND M. FRIEDMAN. Predicting How Badly ‘Good’ Software can Behave. *IEEE Software*. To appear.
- [7] J. VOAS, J. PAYNE, R. MILLS, AND J. MCMANUS. Software Testability: An Experiment in Measuring Simulation Reusability. In *Proc. of the ACM SIGSOFT Symp. on Software Reusability*, pages 247–255, Seattle, WA, April 1995. ACM Press.
- [8] J. VOAS AND K. MILLER. Software Testability: The New Verification. *IEEE Software*, 12(3):17–28, May 1995.
- [9] J. VOAS, A. BINNS, R. MILLS AND J. PAYNE. An Experiment Applying a Fault-injection-based Fault-tolerance Measure to an Automobile Control System. In *Proc. of the 2nd IFAC Workshop on Safety and Reliability in Emerging Control Technologies*, Daytona Beach, FL, November 1995.