Putting Assertions in Their Place

Jeffrey M. Voas
Reliable Software Technologies Corp.
PO Box 2393, Suite 250
11150 Sunset Hills Road
Reston, VA 22090
(703) 742-8873
jmvoas@isse.gmu.edu

Keith W. Miller
Department of Computer Science
Health Sciences Building 137
Sangamon State University
Springfield, IL 62794
(217) 786-7327
miller@eagle.sangamon.edu

Abstract

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.

In this paper we advocate 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 locations where testing is unlikely to be effective.

1 Introduction

Software testing is performed for two reasons: (1) to detect and then remove faults, and (2) to estimate the reliability of the code. Testing is effective when it uncovers faults in the code. When testing is ineffective, when it does not reveal existing faults, the effects are significant and potentially dangerous. The faults will remain to surface after the software is delivered. Reliability estimates will be artificially high. When hidden faults reside in safety critical software, the results can be catastrophic.

Our research focuses on how faults "hide" from testing; the best hidden faults are the most dangerous. In this paper, we apply a dynamic testability technique called "sensitivity analysis" to identify where faults (if they exist) are likely to hide during testing [9]. Such locations are termed "low testability" locations.

An *assertion* is a test on the state of an executing program or a test on some portion of the program state. 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 specification. A benefit of checking values internally is that we know as soon as possible whether the program has entered into an erroneous state.

We will assume that all assertions evaluate to TRUE when the internal state is satisfactory, and FALSE otherwise. In this methodology, 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. Hence we "artificially" modify what we consider as failure in the output space: a *failure* occurs if the output is incorrect *or* an assertion fails. Effectively, this not only modifies what is considered a failure, but it also modifies what is considered output.

"Observability" is a term used in hardware design that means we can detect the inner workings of a chip. An assertion is a method for increasing observability in software by increasing the dimensionality of the output space. Then after we have tested the program N times and not observed a failure (with the assertions in), we have a confidence that faults are not hiding in the code. We can then apply the "Squeeze Play" model [6] (first introduced at ISSRE92 and later at ISSTA93 [10]). Once testing is completed, the assertions

1071-9458/94 $4.00 © 1994 IEEE

may be removed if the development team chooses. (The removal of assertions is most often motivated by the desire for more efficient execution during production runs.)

2 Fault Simulation in Sensitivity Analysis

We define *software testability* as the tendency of code to reveal existing faults during random testing. This paper proposes to take software testability predictions into account during the quality assessment process.

Software testability, as measured by sensitivity analysis, is a function of a (program, input selection criteria) pair. The means by which inputs are selected is a parameter of the testing strategy: inputs can be selected *randomly* based on an input distribution,[1] they can be selected based upon the structure of the program, or they may be selected based on the tester's human intuition. The input selection criteria is fundamental to the information produced by sensitivity analysis; the criteria should be chosen to satisfy a specific testing goal. Although we limit ourselves to random black box testing in this paper, other types of testing could have their own testability measures. [3].

In order for software to be assessed as having "good" testability by this definition, it must be likely that a failure occurs during testing whenever a fault exists. To understand this likelihood, it is necessary to understand the sequence of events that leads up to a software failure. (By software failure, we mean an incorrect output or failed assertion that was caused by a flaw in the program, not caused by a problem with the environment in which the program is executing.) Software failure only occurs when the following three necessary and sufficient conditions occur in sequence:

1. A input must cause a fault to be *executed*.

2. Once the fault is executed, the succeeding data state must contain a *data state error*.

3. Once the data state error is created, the data state error must *propagate* to an output state. (We include here failed assumptions as output states.)

[1]Note that *randomly* does not imply *uniformly*; a uniform distribution is only one type of random distribution.

In order to make a prediction about the probability that existing faults will be revealed during testing, testability analysis must predict whether a fault will be executed, whether it will *infect* the succeeding data state creating a data state error, and whether the data state error will propagate its incorrectness into an output. When an existing data state error does not propagate into any output variable, we say that the data state error was *cancelled*. When all of the data state errors that are created during an execution are cancelled, the existence of the fault that trigged the data state errors remains hidden from testing, resulting in a lower software testability. These conditions provide a reasoned approach to for predicting the testability of software an approach that is tightly coupled to the fault/failure model of computation.

Sensitivity analysis [9] can be used to assess software testability; it is based on the fault/failure model. Sensitivity analysis is based on three subprocesses, each of which is responsible for estimating one condition of the fault/failure model: *Execution Analysis* estimates the probability that a statement is executed according to a particular input distribution; *Infection Analysis* estimates the probability that a syntactic mutant affects a data state; and *Propagation Analysis* estimates the probability that a data state that has been changed affects the program output after execution is resumed on the changed data state.

Sensitivity analysis makes predictions concerning future program behavior by estimating the effect that (1) an input distribution, (2) syntactic mutants, and (3) changed values in data states have on current program behavior. More specifically, the technique first observes the behavior of the program when the following programs are all run repeatedly with a the same input distribution: (1) the original program (2) copies of the program in which one location at a time is injected with syntactic mutants, and (3) copies of the program in which a data state (that is created dynamically by a program location for some input) has one of its values altered. After observing the behavior of the programs thus generated, the technique then predicts the future behavior of the original program under the assumption that a fault exists. These three types of program executions correspond to the three necessary and sufficient conditions for software failure to occur: (1) a fault must be executed, (2) a data state error must be created, and (3) the data state error must propagate to the output. Therefore the technique is based firmly on the conditions necessary for software failure.

In our research we define a "location" as a single assignment statement, a statement that can change the flow of control, an input statement, or an output statement. Our definition of what syntactic unit constitutes a location is based the need to isolate different portions of a program in order to examine the likelihood that the portion of code is in error. Further, we consider that what happens in a data state at a location affects a user-defined variable or the program counter of the data state. Even if there is a side-effect in a location, we can treat the location as two locations with two different variables affected.

We predict the probability that a fault in a location will change the data state of the program. This process is repeated several times for each location: a first-order syntactic mutation is made to the location in question. The program with this mutated location is then run some number of times with inputs selected at random according to the program's input distribution. For all the times the mutated location is executed, we record the proportion of times that the program with the mutated location produces a different data state than the original location; this proportion is our estimate of the probability that a fault at this location infects. In general, many different syntactic mutants are made for a single location, each yielding a probability estimate in this manner. The probability estimates (termed ***infection estimates***) for this location along with those for other locations in the software can then be used to predict the software's testability. Although sensitivity analysis only makes simple code modifications, evidence of the coupling-effect suggests that characteristics observed for simple mutants may frequently be similar for complex faults [8].

The final set of predictions concerns the probability that a data state error will propagate to the output, thereby causing a recognizable failure. This process is repeated several times (over a set of program inputs) for each location: the program is executed with an input selected at random from the input distribution. Program execution is halted just after executing the location, a randomly generated data value is injected into some variable, and program execution is resumed. If the location is in a loop, we customarily inject another randomly selected value into the same variable on each successive iteration. Specific details on how this process is performed are found in [9]. This process simulates the creation of a data state error during execution. We term this process "perturbing" a data state, since the value of a variable at some point during execution represents a portion of a data state. We automatically record any subsequent propagation of the perturbed data state to successor output states after execution is resumed. This process is repeated a fixed number of times, with each perturbed data state affecting the same variable at the same point in execution. This process is repeated, perturbing different variables, once variable at a time. Probability estimates found using the perturbed data states can be used to predict which regions of a program are likely and which regions are unlikely to propagate data state errors caused by genuine software faults. The probability estimates (termed ***propagation estimates***) for this location along with those for other locations in the software can then be used to predict the software's testability; a model for doing so is presented in [9].

3 Locations Needing Assertions

Let L represent the set of all locations in a program, P. Let Θ_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;[2] Θ_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 θ_ls. 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 θ_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, ε, such that locations with scores below the cut-off are judged to be dangerously insensitive to faults. 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

[2]We only use non-zero infection estimates because we do not determine mutant equivalence.

must be extracted from the specification. Our recommendation: ***create assertions for all locations where $\theta_l < \varepsilon$, and inject 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_value, y, y > x)**, which passes the program input value and the variable being tested **y** into a procedure that returns TRUE if **y** is greater than **x** and FALSE if it is not. The value-added by inserting assertions into software is directly dependent on two factors:

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.

Assertions will have varying degrees of "tightness." For instance, we might not be able to determine exactly what value **y** should have at a location, but we might know that **y** should be in the range [0,10]. This is a "looser" assertion than knowing exactly what value it should have. An even looser assertion would be to say that **y** should have a value in [-maxint, maxint]. Clearly, the looser an assertion is, the less information it provides concerning correctness and the less confidence that we gain when the assertion is executed. The benefit derived from an assertion is directly related to its tightness and correctness.

Our plan for using assertions is simple:

1. perform propagation and infection analysis on the original code.
2. place assertions where warranted by the testability estimates.
3. reperform propagation and infection analysis; the increase in estimates from (1) will be a function of the tightness of the assertions. (This will demonstrate the quantifiable benefit that the assertions have provided.)
4. test the code according to the results of (3) 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) When you reperform the analysis, you can opt to only perform it at those locations that did not receive assertions. (3) 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.

4 When Assertions are Not Available

The ability to extract correct assertions from the specification cannot be automatically assumed; indeed, we anticipate that there will be situations where assertions are either too loose or non-existent. In this event, we must find ways (other than testing and assertions) to convince ourselves that the locations in L' do not contain faults.

Two options immediately present themselves:

1. Formal Fagan style code inspections, and
2. Specialized unit testing on the functions that are comprised of one or more members of L'.

Manual code inspections and walkthroughs have been shown repeatedly to detect faults and be cost effective; one study found that these techniques caught 30-70% of the total number of faults [7]. Essentially, these are static, manual assertions. Given that there have been many anecdotal successes reported with these techniques, organizations are turning to inspections as a way to reduce software development costs, and inspections are also being applied to earlier phases of the software life-cycle. Of course the concern with inspections is that faults do escape detection, and hence an inspection that does not discover any faults is not a guarantee of correctness.

Specialized unit testing can be performed in many different ways. Recall that sensitivity analysis is usually performed at the system level D. (This does not preclude you from performing sensitivity analysis using D' as a unit testing distribution on P' as a module in isolation away from the remainder of the system.) Possibilities for unit testing schemes include data flow and simple coverages. More advanced schemes such as mutation can be applied if tools are available for the source language of the program P.

5 Incorrect Assertions

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. Thus, correct assertions are far more valuable than incorrect assertions.

The testability-based assertion injection model 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 enough to simply insert assertions and claim that the overall testability is increased.

6 Simple Example

We modified the tiny example program used in [5] to contain *one* assertion on variable **x** at location 10; the assertion was tight enough such that if **x** were corrupted in any way before location 10 is executed, then the assertion would fail:[3]

```
1. read(a,b,c);
2. if a <> 0 then begin
3.   d:=b*b - 5*a*c;
4.   if d < 0 then
5.        x:=0
     else
6.        x:=(-b + trunc(sqrt(d))) div (2*a)
    end
   else
7.   x:= -c div b;
8. if (a*x*x + b*x +c = 0) then
9.      writeln(x, 'is an integral solution')
   else
10.     writeln('There is no integral solution')
```

[3]We did not assume that the assertion on **x** was correct.

The propagation estimates for that example program were in the range $[10^{-4} \ldots 10^{-2}]$. After using the assertion to increase the amount of internal information being checked, we assessed propagation estimates in the range $[10^{-1} \ldots 1.0]$. This represents a several orders of magnitude increase in the function's testability, which can be translated into a several orders of magnitude decrease in testing costs.

7 Summary

Assertions represent a valuable tool for assessing the quality of software. We have presented a means for better deciding where assertions are needed for software systems that suffer from 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 everywhere. We contend that this assertion placement scheme is a practical way of making a squeeze play occur to derive a more accurate estimation of the reliability of highly reliable software.

The benefit derived from this use of assertions will be a function of the tightness on the assertions. Extremely loose assertions should not be considered sufficient to ignore their testabilities during a squeeze, and locations with such assertions are candidates for some other analysis.

This testability-based assertion method will:

1. make a "squeeze" more likely because the testability of a location receving an assertion will less negatively impact the the prediction of how far we must test to, and

2. increase propagation estimates of predecessor locations (determined dynamically during execution), whose computations are referenced by the variable being asserted on.

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. Assertions benefit both our testing and our testability results.

Hecht [1] has indicated that rarely encountered faults may be the dominant cause of safety and mission critical failures. Rarely encountered faults can hide for long periods of time even while a system is undergoing testing and debugging that is aimed at reliability improvement. If just one rare fault has the potential to lead to a catastrophic failure, then we would have a reliable, *unsafe* system. Hence the goal of this assertion-placement model is to reduce the likelihood of *any* fault hiding in the system, by attacking the program at its "weakest" points with assertions.

Note that the use of assertions for increasing the confidence in safety-critical applications is not unique; what is unique about this method is the use of "testability-based" assertions that are placed in only those locations in the code where the degree of error masking is predicted to be high. In [2], Guiho & Hennebert stated that the validation of SACEM, a partially embedded system which controls the speed of all trains on the RER Line A in Paris France, required about 100 man-years of effort; assertions were needed for formal proofs. The assertions were a critical part of that validation effort; these assertions could have also been placed in the code during testing, but apparently were not. The interesting result here is that the developers feel that they have achieved a 10^{-9} probability of failure by equipment per hour, i.e., this new SACEM system is as safe as the old system that it replaces that had a much lower traffic throughput. Hence assertions are a vital entity to have during validation, whether they are used for formal proofs (to verify safety code) or as internal self-tests during testing.

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 actually assess a high confidence that faults are not hiding during the testing.

References

[1] H. Hecht. Rare Conditions: An Important Cause of Failures. In *Proc. of the Eighth Annual Conference on Computer Assurance*, pages 81–85, National Institute of Standards, June 1993.

[2] Gerard Guiho and Claude Hennebert. SACEM Software Validation. *IEEE Experience Report*, pages 186-191, 1990.

[3] William E. Howden and Y. Huang. Analysis of Testing Methods Using Failure Rate and Testability Models. Technical Report CS93-296, University of California at San Diego, June 1993.

[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] J. Voas, L. Morell, and K. Miller. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2):41–48, March 1991.

[6] J. Voas and K. Miller. Improving the Software Development Process Using Testability Research. In *Proc. of the 3rd Int'l. Symposium on Software Reliability Engineering.*, pages 114–121, Research Triangle Park, NC, October 1992. IEEE Computer Society.

[7] G. Myers. *The Art of Software Testing.* Wiley, 1979.

[8] A. J. Offutt. The Coupling Effect: Fact or Fiction. *Proceedings of the ACM SIGSOFT 89 Third Symposium on Software Testing, Analysis, and Verification*, December 1989. Key West, FL.

[9] J. Voas. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, 18(8):717-727, August 1992.

[10] R. Hamlet and J. Voas. Faults on Its Sleeve: Amplifying Software Reliability Assessment. In *Proc. of ACM SIGSOFT ISSTA'93.*, pages 89–98, Cambridge, MA, June 1993.