

Using Assertions to Make Untestable Software More Testable

Jeffrey Voas
Reliable Software Technologies
Suite 250, 21515 Ridgetop Circle
Sterling, VA 20166
jmvoas@RSTcorp.com

Lora Kassab*
College of William and Mary
Dept. of Computer Science
Williamsburg, VA 23185
kassab@parker.itd.nrl.navy.mil

Abstract

Dynamically testing software that has been augmented with assertions increases the defect observability of the test cases provided that the assertions are reached during testing. This paper presents an approach to assertion localization that is based on finding regions of the code that appear to be untestable, and then making them more testable. This is accomplished through a combination of static and dynamic analyses. This paper also explores the phenomenon where assertions which are designed to boost the fault observability provided by test scheme D cannot lower the fault observability afforded by a different testing scheme D' , and in fact, may actually increase it. If true, this demonstrates a unique and cost-effective benefit of assertions not before exploited, and lays forth a new avenue for finding higher return-on-investment testing techniques.

Keywords

Assertions, faults, failure, testing, fault detectability, oracles, specifications, testability, formal specifications, fault propagation.

1 Introduction

Software testing is generally performed for one of two reasons (1) to detect the existence of defects (*faults*), and (2) to estimate the reliability of the code. Testing is considered effective when it uncovers defects. Testing will often be considered ineffective when no failures occur, since the notion of defect-free code is unthinkable. Residual software defects not uncovered

*Ms. Lora Kassab is now employed in the Center for High Assurance Computer Systems at the Naval Research Laboratory in Washington, D.C.

during testing 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 reveal defects (if present) before software release are sought.

Software testability is a characteristic that either suggests how easy software is to test, how well the tests are able to interact with the code to detect defects, or some combination of the first two [8]. Describing how easy software will be to test is valuable information for project schedules and project cost estimation. Yet this information provides little insight into how good your test case generation was at creating “defect-detecting” test cases. Because of this deficit, it is useful to employ the perspective that software testability is a measure of how good test cases will be at making defects *detectable*, and this will be the perspective used throughout this paper.

Using this definition, when software is assessed as having higher testability, it means that incorrect output will likely occur if a defect exists. To understand why faults hide during testing, it is necessary to know the sequence of events that must occur in order to observe incorrect output

1. A input must cause a defect to be *executed* (or what is sometimes referred to as “reached”).
2. Once the defect is executed, the succeeding data state must become corrupted. This data state is hence referred to as containing a *data state error*.
3. After a data state error is created, the data state error must *propagate* to an output state, meaning that incorrect output exits the software.

This sequence of events is sometimes called the “fault/failure” model, because it relates faults, data state errors, and failures [8]. Since faults trigger data state errors that in turn trigger software failures, any analysis that claims to suggest whether testing is capable of detecting defects must account for all three conditions¹.

This paper argues that by using heuristics for where faults cannot be detected by testing alone, we can identify *where* additional validation efforts (which may include more testing or non-testing approaches) should be performed. We provide a methodology that is complementary to traditional testing which thwarts the potential of defect hiding in those code regions. Our approach can be thought of as a strategic assertion placement heuristic. Doing so improves the likelihood of defect detection and hence improve the software’s reliability.

Over the years, we have observed that although many testers would like to begin using more assertions to improve the quality of their testing process, they find that their problem is that they do not know the code well enough to inject correct assertions. Assertions are primarily a tool used by developers, however this increases the likelihood that if the code is wrong, the assertions they add to augment testing will also be wrong. Both of these problems are disheartening, because as you will see, a handful of correct and strategically-placed assertions can make all of the difference in the quality of a finished software product.

¹Faults and defects may be used interchangeably as they are synonymous terms.

This paper unfortunately does not offer solutions to the problem associated with deriving correct assertions, but it does provide additional ammunition for why assertions must be used to improve the deficiencies of software testing, even if that requires deriving assertions from formal specifications. In fact, interest in assertions has become so great that several recent languages support assertion placement, including Anna [4] and Eiffel [7].²

The testing phase of the software life-cycle is the time when software assertions should be used. Assertions can benefit each of the key testing phases unit, integration, and system. Assertions will be easier to derive and more effective at finding defects during unit testing. But assertions can be very beneficial for warning when assumptions are violated after software units are integrated. Here, assertions can warn when assumptions that one component made about another do not hold. For example, if one component is expecting to receive input from another in a certain range which does not occur, then the receiving component can opt to not accept the input or send out a 'warning.' In this manner, assertions can act as pre-conditions during integration and system-level testing.

2 Run-time Software Assertions

Manually finding defects in a program's output is difficult if failures occur rarely. This simply is not a task that humans are accurate at. Hence *automated testing oracles* (or what throughout this paper we simply term "oracles") are essential when the software is of good enough quality such that failures are rare.

But building automated test oracles requires that the oracle know what is correct (with respect to the specification) and what is not. Fortunately, oracles can be designed directly from *formal specifications* (that describe exactly what the software is supposed to do without also describing the implementation details of the system) [11]. An *executable specification* is a formal specification that can be executed (like a program) to see if the behavior it defines satisfies the higher level requirements of the system. Executable specifications typically produce output, but do not check output. In theory, the output from an executable specification can be given to the oracle so that the oracle will know what is correct.

Run-time assertion checking is a "programming for better validation" trick that helps insure that a program state satisfies certain logical constraints. Unlike executable specifications, run-time assertions do check for the correctness of the output. *Run-time assertions* (or simply "assertions") are based on either the requirements or the specification. Some of the earliest writing concerning assertions can be traced to [5, 9], and more recent research into giving programs the ability to check themselves during execution can be found in [12, 13, 2, 16]. Further, Bieman and Yin recently discussed using assertions to increase fault detectability [1], however our contribution furthers the idea of run-time assertions by providing methods to place assertions where assertions are more desperately needed. Our method decides *what portion* of the software's state really needs to be checked and *where* that check needs to be placed. We term our assertion placement scheme as *strategic run-time assertion checking*.

²Anna (Annotated Ada) uses comments to embed assertions; Eiffel uses object invariants that are inserted as pre- and post-conditions to all operations on the object.

Our interest is to embed assertions in a manner that engenders testing with greater defect revealing ability. The conjecture that has motivating this work follows

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

Our strategic run-time assertion checking presents the opportunity to thwart defect hiding at a more reasonable cost than *ad hoc* assertion placement, which today is the usual heuristic for deciding where to put assertions.

Note the similarity between assertions and the debugging process. Traditionally, debugging has referred to either the manual task of instrumenting code with 'print' statements to reveal what values variables contain or it has been the process of using an automated debugger to walk through a program (statement by statement) watching data states change. Assertions are more closely related to the 'print' statement approach, with the key difference being that assertions can also perform tests during execution. Thus assertions can be used during debugging to test for various conditions.

Assertions can operate in either an "in-line" or "off-line" manner. When assertions are evaluated *in-line* (which is the more common approach), the Boolean tests on the current program state are performed by the assertions and the outcome of TRUE or FALSE occurs. The advantage of this is that in-line assertions can opt to terminate execution if evaluate to FALSE. If assertions are evaluated *off-line*, the assertions act much more like 'print' statements which simply dump out state information for analysis outside of the executing program. But off-line processing also has benefits. For example, the off-line approach can allow for a more complex analysis of the internal states to determine if something is incorrect than could be done in-line (due to performance constraints). In this paper, we will assume that in-line assertions are the form being used.

The typical format for an in-line assertion statement follows

```
ASSERT (<condition>, <message>) ;
```

The < *condition* > is a valid boolean statement written using an assertion language. The < *message* > is a text string that will be displayed when the < *condition* > evaluates to FALSE. In the following example, an ASSERT statement is used to insure that the variable x does not have a negative value after the assignment statement.

```
. . . .  
x = y ;  
ASSERT( x >= 0, "x has a negative value") ;  
cout << "Value of x is " << x << endl ;  
. . . .
```

If the < *condition* > evaluates 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. Because a program that did not have assertions is truly a different program after assertions are added, it is necessary for us to modify what we consider as a program failure a program *failure* will

be said to have occurred if the program output is incorrect *or* if the $\langle condition \rangle$ evaluates to FALSE. This not only modifies what is considered failure, but it also modifies what is considered output, because there now is one more bit of output each time when an assertion is executed.

3 Increasing Observability Through More Output

Whenever the amount of output increases (*e.g.*, outputting 2 64-bit floating point values as opposed to one), more of the internal (intermediate) calculations can be *observed*. *Observability* has long been a metric used in hardware design that describes the degree (or ability) of a chip to detect problems in the inner logic of a chip at the output of the chip. When observability is poor, BIST (Built-In Self Tests) have often been employed to force complex circuit to perform self validation. These hardware probes are placed into circuits to increase the observability of the circuit during test. Similarly, assertions increase software's observability by increasing the dimensionality and/or cardinality of the software's output space, which is precisely what you want if your goal of testing is to catch defects. (This is discussed in more detail later.)

Similar to executable specifications, *correctness proofs* can be thought of as a formal assertion checking system. The difference, however, is that correctness proofs statically test to ensure that the entire program satisfies certain logical constraints for all inputs, whereas an executable specification, like a program, is run on a per test case basis. Software assertions perform a different function than correctness proofs or software testing; they semantically test internal program states that are created during execution and that are currently not observable as stand-alone entities. For example, given a known range of legal values for some intermediate computation in a program, a software assertion can test the correctness of the program state at the instant when the state is created. Since assertions are able to check intermediate data state values, they can reveal when the program has entered into an undesirable state. This is vital, because it is possible that the undesirable state will not propagate into a program failure.

The effect of assertions on the dimensionality and cardinality of a program can be best explained through examples. Figure 1 illustrates a program that reads in an integer and outputs an integer; in this example, an input value of 5 produces 100, 6 produces 200, and 7 produces 300. Thus, the dimensionality of the output space in Figure 1 is one.

In Figure 2, the conditional branch in the code causes only certain inputs to execute the assertion. The assertion essentially acts as another output statement whose result will be checked by the oracle. Thus, for some inputs, the dimensionality of the output actually increases to two. In Figure 2, the inputs 5 and 6 execute the assertion and will therefore have outputs with a dimensionality of two, whereas an input value of 7 will have a dimensionality of one.

Now imagine a slightly different example where two unique input cases resulted in the same output value, and assume this value is of dimension n . By adding an assertion to the code that both input cases execute, it is possible that the variable asserted on now has different values, and hence each input case can be thought of as producing a unique output value of dimension $n + 1$. In this example, we see how an assertion can increase the

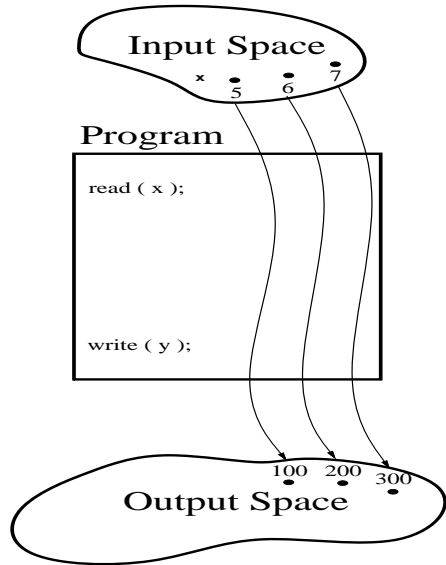


Figure 1 One-dimensional output.

cardinality of the output space.

Assertions have an interesting ability that is similar to the ability of oracles. It is their ability not to only warn of problems at the point in the code where they evaluate to FALSE, but instead a FALSE outcome may actually tell of a problem at a totally different place in the code, and not necessarily a problem with the preceding statement. This requires reversing the execution trace back to the preceding computations in order to localize the problem. It is this ability to warn of problems originating from various statements that increases the fault detectability provided by assertions.

Just as programs and oracles can be flawed, so can assertions. Assertions can fail to warn when they should and warn when they should not. Clearly then the benefit provided by assertions during testing is directly tied to the correctness of the assertions. But then the same can be said for testing in general “if the oracle is seriously flawed [6], why bother testing?” The best approach for increasing the likelihood of valid assertions is to employ someone to derive them who did not write the code but who understands the specification and who can understand the code.

4 Strategic Assertion Placement

We advocate a middle ground between no software assertions at all (the most common practice) and the theoretical ideal of assertions on every statement in a program. Our compromise is to place assertions *only* at locations where traditional testing is unlikely to uncover software defects.

Predicting where fault hiding is likely to occur is an expensive process, because there are so many considerations that must be factored in. We will describe two approaches here (1) dynamically execute the code with appropriate instrumentation to gather the data that the instrumentation outputs (See Section 4.1), and (2) a static analysis of the code that searches

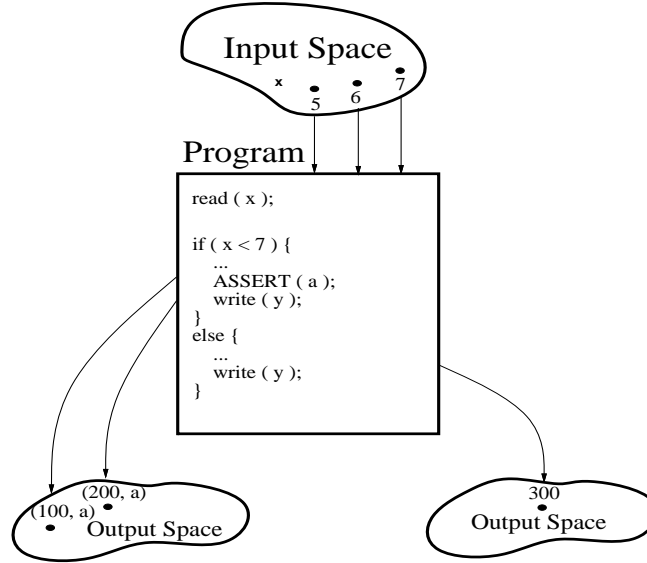


Figure 2 Two-dimensional output from added Assertion

for particular language constructs and interconnections between constructs (See Section 4.2). In this paper we will concentrate on methods for performing (2), however we will provide a high-level overview of (1). The benefit of the static approach is its ability to be applied to very large systems. The downside to the static approach is a lack of precision, because the static approach can not consider the program states that are dynamically created.

4.1 Dynamic Approach

Sensitivity analysis is a dynamic approach for predicting where faults will hide from test cases [8]. Sensitivity analysis predicts the likelihood that a test scheme will (1) exercise the code (i.e., enforce *reachability*), (2) cause internal states to become corrupted when defects are exercised, and (3) propagate data state errors to the output space. These three conditions must occur in order that defects can be observed. To assess reachability, sensitivity analysis tracks how frequently statements in a program are exercised. To assess (2) and (3), sensitivity analysis employs a variety of different *fault-injection* techniques that both mutate the software as well as mutate the internal program states that are created during execution.

4.2 Static Approach

Unlike dynamic analyses, static code-based analyses can provide a lower cost means for predicting where faults will hide. The static method that we will explore here examines code for a characteristic called “implicit information loss”, we can determine where faults are more likely to hide. *Implicit information loss* occurs when information computed during program execution is not communicated to the program’s output. This lack of communication increases the likelihood of fault hiding.

A simple example will best illustrate when this communication breakdown occurs between input and output. In the statement $a = a \text{ div } 2$, any information in the least significant bit

of a is eliminated, whereas the statement $a = a + 1$ does not result in implicit information loss.

The degree of implicit information loss can be roughly predicted using the *range/domain ratio* metric (RDR). Simply stated, RDR is the ratio between the cardinality of the range of a specific operator in a specific statement to the cardinality of the domain of that operator. So for example, the “mod 2” operator in $a = a \bmod 2$ always has a range of 2 if there is at least one even and one odd value for a before this statement is executed. But the domain could differ with respect to different domains for a if this statement occurs at different places in the program. A decrease in an RDR score for some code statement implies an increase in the degree of information loss. A decrease in an RDR score for some code statement also implies a decrease in propagation, which suggests a decrease in fault detectability.

Generally speaking, faults are more likely to go undetected as the domain’s size increases with respect to the range’s size. As the domain’s size increases with respect to the range, the likelihood that bad information will propagate decreases, and one reason that bad information often does not propagate is that the second condition in the fault/failure model does not occur. Thus this ratio provides insight into how likely it is that program states will become corrupted and propagate.

Consider the following analogy. Suppose that you have developed a complex software package that reads in large quantities of data, processes the data, and returns either a ‘0’ or ‘1’ with uniform frequency. Suppose that while you wait for the software to produce an output, you decide to flip a fair coin, with “heads” representing ‘1’ and “tails” representing ‘0’. For approximately 50% of the data sets, you and the software will agree. Why is this? It is because of the extraordinary simple output space that is uniformly distributed. Now suppose that the software is redesigned to not only output a ‘1’ or ‘0’, but also to output vast amounts of internal data long with each ‘1’ or ‘0’. Your likelihood of now matching the output of the software has just decreased to near zero.

Here we see an example where a ratio of “many-to-two” different inputs to outputs increases the possibility that the software can simply *guess* at the right answer and be correct. When programs can simply guess at their outputs can be correct, that suggests that faults could easily hide during testing. But notice that when the input to output ratio was reduced (by increasing the precision of the output data), it became harder for the software to simply guess at the right output, because now the software needs to also guess at what the correct values are for the additional outputted information. But decreasing this ratio, testing was afforded a much greater chance of detecting faults.

We will now discuss how we will look into code to see how much “guessing” ability the code actually has. The representation of the software that we employ for analyzing the degree of implicit information loss is a basic control flow graph. Our approach is twofold (1) find the regions where the implicit information loss occurs, and (2) traverse backwards through a control flow graph to localize where assertions are needed.

Using the control flow of a program P , we construct a directed flow graph $G = (V, E)$, where V corresponds to a set of locations in the program P , and E corresponds to the set of edges giving order to the locations in P . In determining where faults are likely to hide, we are only concerned with the locations in the code that are assignment statements. Therefore

```

for each vertex  $v \in V^R$ 
  if Assert_Required( $v$ )
    for each element  $e \in v_{rhs}$ 
      found  $\leftarrow$  FALSE
       $temp \leftarrow$  NULL
      for each reachable vertex  $u$  from  $v$ 
        if  $e \in u_{lhs}$ 
          if found
            Clear Post_Assert( $e$ ) on  $temp$ 
             $v_{rhs} \leftarrow$  Pre_Assert( $e$ )
            break
          else
             $u_{lhs} \leftarrow$  Post_Assert( $e$ )
             $temp \leftarrow u_{lhs}$ 
            found  $\leftarrow$  TRUE

```

Figure 3 Reverse flow graph traversal algorithm.

each vertex in V is an assignment statement in P .³

In order to determine proper assertion placement, we first determine statically whether the degree of information loss for each vertex is intolerable. This is determined by comparing the RDR score to the threshold value for each vertex. As mentioned earlier, the RDR score, ϕ , is computed by dividing the cardinality of the range ratio by the cardinality of the domain ratio. A threshold value, δ , is an assigned value in the range $(0, 1]$, indicating the level at which information loss is not tolerable. (From experience, we have traditionally used a threshold value of 0.001, but we have no convincing theoretical justification for doing so.) Now, if the value of ϕ is equal to or surpasses the assigned threshold level, δ , then the vertex (the assignment statement) requires an assertion and is marked accordingly.

In addition to marking whether each vertex requires an assertion, a set of left-hand side and right-hand side variables must be recorded for locating the variables that require an assertion. Once we've determined the regions where implicit information loss is intolerable, we determine where to place these assertions by using a reverse flow graph.

From the flow graph G , we construct a reverse flow graph, $G^R = (V^R, E^R)$.⁴ The set of edges in E^R is created by adding the edge (v, u) for every edge (u, v) in E . The set of vertices in V^R is identical to V . Since this is the reverse flow graph, all vertices reachable from the current vertex constitute all the possible assignment statements that could have been previously executed. The algorithm in Figure 3 determines where assertions are needed.

In the algorithm in Figure 3, if the Boolean function **Assert_Required**(v) evaluates

³It must be noted that the assignment statement program construct encompasses both input statements and parameter passing. An input statement is essentially an assignment statement where the input is retrieved from an external source, i.e. a file, or a user. Likewise, a function call is simply an assignment of the actual parameter to the formal parameter.

⁴Note that we could have created a regular flow graph and then backwards traversed it; this paper describes using the reverse flow graph because we believe that makes it easier for the reader to visualize what is happening.

TRUE for the assignment statement corresponding to vertex v , that is $\phi \geq \delta$, then vertex v requires an assertion. If so, we are interested in finding the assignment statement(s) that could affect the outcome of v (i.e., the left-hand side of v). This means that we must visit all previously executed assignment statements whose outcome affects *every* variable occurring on the right-hand side of v (i.e., v_{rhs}). If only one vertex, u , is found such that the variable e is in u_{lhs} and v_{rhs} , then an assertion on the variable e , **Post_Assert**(e), is placed *after* the assignment statement u is executed.

Even though an assertion has been placed for the variable e , we visit other vertices to see if a second assignment statement is found whose outcome affects the same variable e . If a second assignment statement is found, then the **Post_Assert**(e) that was previously placed on the vertex stored in *temp* is removed in order to perform a **Pre_Assert**(e) on the variable *before* execution of statement v . By placing the assertion prior to statement v , we eliminate the possibility of having multiple assertions for variable e . At this point, we no longer need to visit other nodes, and can begin traversal for the next variable on the right-hand side of statement v .

Hence placing one **Pre_Assert**(e) on variable e before v is theoretically equivalent to placing multiple **Post_Assert**(e)s on variable e at the different assignment statements where e is assigned. From an implementation standpoint, however, we must be careful that the test employed before v is the right test. (Placing an assertion farther away from where the “action” is happening may increase the likelihood that the assertion is not what is wanted, and this may be a much more complex assertions if there are multiple statements where $e \in u_{lhs}$ is true.) The advantage of one **Pre_Assert**(e) versus multiple **Post_Assert**(e)s is better performance since having to compile in hundreds of assertions versus a single assertions is undesirable. However it is likely that a single **Pre_Assert**(e) will require complex logic since it must account for all possible different computations from the different statements where e is assigned.

In object-oriented software, it is common to reach v from a host of unique places in the code, and each of these places might use a slightly different computation for assigning e . Since v depends on e , and we know that if e is corrupted it is very unlikely that testing will discover this, we need to “internally” test e . And we can either decide to test e right after it is assigned (everywhere), or once (right before it is used at v).

For examples of this algorithm, see Figures 4 and 5. In Figure 4, we see that a poor RDR score has been isolated at the statement $\mathbf{a} = \mathbf{a} \bmod 2$. Backtracking through V^R isolates the statement $\mathbf{a} = \mathbf{x} + 1$ as a statement where **Post_Assert**(a) is needed. Similarly in Figure 5, we see an example where multiple locations in V^R need **Post_Assert**(a)s, but instead of embedding multiple **Post_Assert**(a)s, one **Pre_Assert**(a) will be placed before $\mathbf{a} = \mathbf{a} \bmod 2$.

5 Generalized Observations and Recommendations

To close out our article, we wish to put forth several observations that we have discovered regarding the relationships between assertions, test cases, and fault hiding. We will also make recommendations on several other issues such as assertion documentation and how to remove assertions after testing is completed.

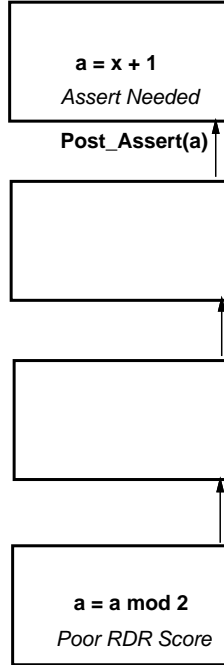


Figure 4 **Post Assert(a)**.

Many software quality professionals know that they should be regularly using assertions but they are unclear as to (1) how to get started, and (2) how to approach their management and argue why assertions need to become a standard part of their testing culture. Hopefully, some of these observations will provide the insights as to what assertions and reachability analysis can do towards making validation efforts more fruitful.

5.1 Reachability

Static 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 fault detectability of program P when tested with test suite D . And let $(D \cup \Delta)$ represent test suite D after it has been augmented with enough test cases such that all statements in P are exercised.⁵ 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 faults to be detected, hence

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

(This is similar to Weyuker's Monotonicity Axiom [15].)

⁵There are software test case generation tools (Godzilla [10], WhiteBox (TM) TGen) that are intelligent enough to sometimes find test cases that will exercise previously unreachable statements in the code, but this is an unsolvable problem in general.

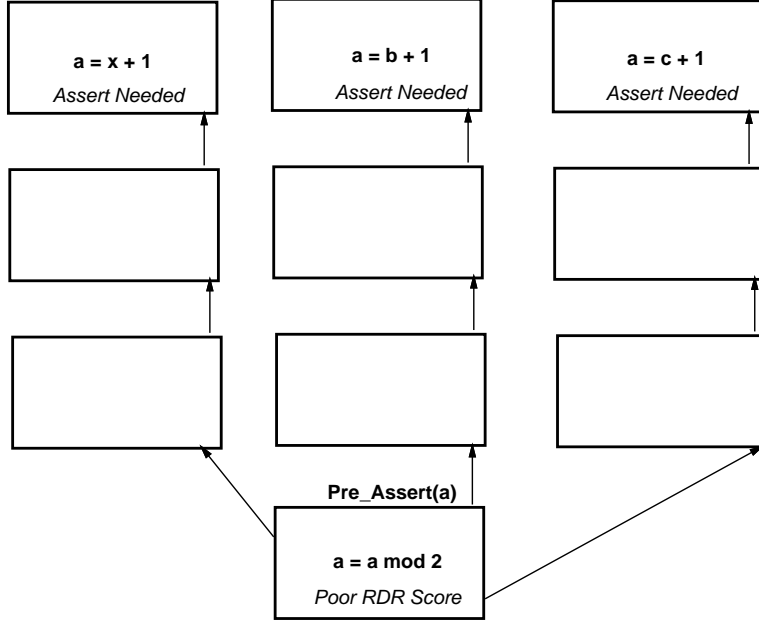


Figure 5 **Pre Assert**(a).

5.2 Changing Test Suites

Are faults more or less likely to be hiding after assertions are added and the manner by which test cases are generated changes? We begin to answer this question by looking at the situation where the code has no assertions, and then we will consider the situation where assertions are embedded.

For program P without assertions, we showed that regardless of what Δ added 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 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 boost the fault revealing ability of D . Assertions increase the likelihood that the second and third conditions of the fault/failure model happen. Then,

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

From there, we will conjecture that

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

regardless 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 it is possible that $T(P)_{D'} \leq T(P_{A_D})_{D'}$ will be true.

What this suggests for 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 fault detectability 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 suspect that it will generally be true that

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

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.

5.3 Testers vs. developers

But knowing that variable \mathbf{a} needs to be tested after assignment statement 100 does not indicate what constraint the assertion should be testing for (to determine whether a problem has occurred). Hence our recommendations have only addressed one part of the oracle/assertion problem, specifically placement, not assertion derivation.

As mentioned earlier, testers are likely to be incapable of deriving correct assertions, while developers are more likely to derive assertions that mimic the semantics of the code already there. If the code is faulty, the assertions will also be faulty.

This brings us to our final recommendation for how to team developers and testers in a partnership to strategically derive and embed assertions. Let the testers find where assertions are needed, and leave it to the developers to determine what the assertions should be. Note that this is different than the case where the developer determines both where to place the assertions and what they should be. Here, the developer is being forced to derive assertions that the tester needs for places in the code *where the developer might not be as sure as to what the assertion should be*. If this happens, it forces developers to dig deeper into the code and requirements than they might have already done. This plays a similar role as code inspections, except that the person digging into the code is also the person that is likely to have written the code. Although this is not a foolproof solution, it is the best recommendation that we can provide at this time. And certainly having developers spend more time comparing their previous understanding of the code to the existing requirements can only serve to improve the code's quality. Even if the assertion that the developer derives does not detect an error, the fact the the developer is forced to derive the assertion will increase the likelihood that the developers themselves find errors, because they are forced to get more familiar with less familiar internal computations in the code.

5.4 Assertion Documentation

Most people would agree that every statement in a program does not need to be commented. But for assertions, this is not true. Every assertion should be documented, particularly if

they are left in after the software is released. The reason for this is that assertions often contain implicit assumptions that are easily forgotten by the person that derived them. Further, such assumptions will be even less well understood by other persons. Therefore full documentation of all assumptions is prudent.

5.5 Assertion Removal

Once testing is completed, the assertions may or may not be removed. Leaving assertions in after testing is completed is reasonable if you are willing to accept reduced performance. Afterall, even off-line assertions require execution cycles. Also, assertions will continue to produce extra output which may not be desirable and may not be relevant (in the context of a software component reused in different environments). Thus the decision not to remove assertions requires justification.

If you desire to remove assertions, you run the risk of doing so incorrectly and causing other problems. In our opinion, the best approach to removing assertions is to have two versions of your software the original one before assertions were embedded and the version with the assertions. When you are done testing with the “ASSERT-ed” version, discard it and deploy the original version. This way you will avoid a flawed assertion removal process.

6 Summary

We have provided a methodology containing both static and dynamic analyses for predicting where faults will hide during testing, and our methodology either recommends generating new test cases or deriving assertions. To date, we have applied the recommended dynamic analyses (sensitivity analysis and reachability analysis for assertion placement), and will experiment with the static RDR heuristic once our tool is completed that performs this analysis.

Assertions are applicable to any software application, however the more critical the application (i.e., applications that must strive to be defect-free), the greater the return-on-investment from assertions. After all, if an application only needs to demonstrate modest degrees of quality, then the cost of assertions may not be warranted, as that quality can be demonstrated via basic types of testing.

Note also that there are different forms of assertions for different applications. *Cleansing assertions* [14] are a type that modify internal states when an assertions evaluates to FALSE, and *protective assertions* allow mobile agents to test the potential maliciousness and benevolence of host systems on behalf on the agent’s owner. Thus for specialized applications, specialized assertions may be necessary.

Assertions are not the only verification and validation tricks that could be employed once it is known where testing is unlikely to be capable of detecting faults. Manual inspections, extensive unit testing, or formal analyses could also be applied to ensure that defects are not hiding.

Our conclusion that assertions are beneficial to software testing parallels the comments by Osterweil and Clarke concerning the value of assertions to testing; in their 1992 *IEEE Software* article, they classified assertions as “among the most significant ideas by testing and

analysis researchers” [3]. From our previous work studying why faults hide during testing, we believe that we have provided insights into why assertions work well and how their placement can be made more systematic and practical.

We conjectured that assertions that are based on deficits in some testing suite D may still be valuable tools for improving the defect observability rates of another testing suite, D' . If generally true, this suggests that current research into which testing approach is better may be wasted effort, i.e., possibly any testing technique can be massaged into an excellent fault detector after assertions are instrumented to test untestable regions. Most research in software testing is geared toward finding some method K that will produce a test suite D for which $T(P)_D \approx 10$. This suggests that when high fault detection is the goal, then instead of looking for the ultimate K , derive assertions and generate additional tests, Δ , such that $T(P_{AD})_{(D \cup \Delta)} \approx 10$. In summary, assertions and additional test cases (that ensure reachability) can transform your code from being untestable to testable. Let them.

Acknowledgement

Voas has been partially supported by DARPA Contract F30602-95-C-0282, National Institute of Standards and Technology Advanced Technology Program Cooperative Agreement No. 70NANB5H1160, and National Institute of Standards and Technology Contracts 50-DKNA-4-00119 and 50-DKNA-5-00185. THE VIEWS AND CONCLUSIONS CONTAINED IN THIS DOCUMENT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL POLICIES, EITHER EXPRESSED OR IMPLIED, OF DARPA, THE NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, OR THE U.S. GOVERNMENT.

References

- [1] H. YIN AND J.M. BIEMAN. Improving software testability with assertion insertion. In *Proc. of International Test Conference*, pages 831–839, October 1994.
- [2] M. BLUM. Designing Programs To Check Their Work. Technical report, University of California at Berkley, December 1988.
- [3] L. OSTERWEIL AND L. CLARKE. A Proposed Testing and Analysis Research Initiative. *IEEE Software*, pages 89–96, September 1992.
- [4] D. LUCKHAM AND F. VON HENKE. An overview of ANNA, a specification language for Ada. *IEEE Software*, pages 9–22, March 1985.
- [5] C. A. R. HOARE. An axiomatic basis for computer programming. *CACM*, October 1969.
- [6] 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.
- [7] B. MEYER. *Eiffel the Language*. Prentice-Hall, 1992.

- [8] J. VOAS AND K. MILLER. Software Testability The New Verification. *IEEE Software*, 12(3)17–28, May 1995.
- [9] P. NAUR. Proof of algorithms by general snapshots. *BIT*, 6(4)310–316, 1966.
- [10] R. A. DEMILLO AND A. J. OFFUTT. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9)900–910, September 1991.
- [11] D. J. RICHARDSON, S. L. AHA, AND T. O. O'MALLEY. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, May 1992.
- [12] D. ROSENBLUM. Towards a method of programming with assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, May 1992.
- [13] R. RUBINFELD. A mathematical theory of self-checking, self-testing, and self-correcting programs. Technical Report TR-90-054, Int. Computer Science Institute, October 1990.
- [14] J. VOAS. Building Software Recovery Assertions from a Fault Injection-based Propagation Analysis. In *Proc. of COMPSAC'97*, pages 505–510, Washginton D.C., August 1997.
- [15] E. J. WEYUKER. Axiomatizing software test data adequacy. *IEEE Trans. on Software Engineering*, 12(12)1128–1137, December 1986.
- [16] J.M. BIEMAN AND H. YIN. Designing for software testability using automated oracles. In *Proc. of International Test Conference*, pages 900–907, September 1992.