

A Few Assertions on Information Hiding

J. Voas

Reliable Software Technologies Corporation

Suite 250

21515 Ridgetop Circle

Sterling, VA 20166

phone: (703) 404-9293

fax: (703) 404-9295

Software testing is performed for one of two reasons: (1) to detect the existence of defects, and (2) to estimate the “goodness”, i.e., the reliability of the code under the test profile. Testing is considered effective when it uncovers defects, and ineffective when no failures occur. Residual software defects 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, testing techniques that have the greatest ability to reveal defects before software release are continually sought.

The theoretical and practical limitations of software testing make it imperative that complementary approaches be exploited. In particular, running external test cases through a program in combination coupled with assertions (internal test mechanisms) can improve the fault-detection capability far beyond employing only one of these approaches, provided that adequate reachability is achieved. But assertions are no panacea. They suffer many of the same limitations as software testing, namely, how do you know that the assertion is providing the correct result? Oracles are notorious for themselves being wrong, meaning that when the software’s output is wrong, the oracles will say it is correct, and vice versa. If you are expecting assertions to perform duties similar to those of an oracle, then it is prudent to assume your assertions are incorrect.

Assertions have universally been used by developers to expedite software debugging during development, and several programming languages have gone as far as to provide built-in assertion capabilities as a part of their grammar. Our interest focuses on exploiting assertions in a slightly different way: as a means of boosting the value of testing in those regions of the code that are prime candidates for masking errors. The grand scheme here is to let software testing do its job in the regions of the software where it will be successful at detecting faults, and add assertions in regions where testing will be ineffective.

Assertions are 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, then it is equivalent to a program failure even if the output is correct.

Observability has long been a metric used in integrated circuit design that describes the ability of a chip to reveal internal problems in its logic through the production of incorrect output. When observability has been deemed as poor, BIST (Built-In Self Tests) have often been injected into the logic to force the circuitry to self validate. These BIST probes increase the observability of the circuit during test. Similarly, software assertions increase software observability.

In order for software to be assessed as having a greater likelihood to reveal defects during testing, it must be likely that incorrect output occurs if defects exist. (This is often referred to as software having “high testability” or software having “low error masking ability.”) To understand what is involved in predicting this likelihood, it is necessary to know the three conditions that must occur in order to lead to incorrect output being produced.

1. An 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. Here, we will refer to this as the “propagation condition.”

If the first condition is unlikely to occur, the addition of an assertion will not help, because the defect that the assertion would hopefully flag will not be executed either. However, if the second and third conditions are the conditions that are unlikely to occur, then instrumenting assertions into the code may be the only opportunity to effectively force them to occur. From a testing standpoint, this is precisely what is desired.

For years, it has been speculated that procedural languages were less likely to hide faults during system-level testing than were OO languages. As it turns out, it is the propagation condition that is particularly hard to enforce in OO systems.

When we talk about a potential problems with the object-oriented design (OOD) philosophy and detecting defects during testing, we are only referring to system-level and integration testing, not unit-level testing. Small objects and classes can be tested quite thoroughly in isolation, but it is at the higher levels of object composition where we suspect there is an increase in error masking, e.g., due to capabilities afforded the developer such as **private** and **protected** data objects. There are clear-cut reuse benefits for why OO design methodologies should be used, and we are not in disagreement with these claims. Rapid-prototyping is another benefactor of OO technology. But from a system validation perspective, OO designs may enable catastrophic faults to hide for longer time intervals, and if true, this makes testing such systems more difficult, and hence the cost benefits of these systems must be brought into question.

As a simple test bed for exploring our concerns that OO software systems might be more likely to suffer from residual faults after integration and system level testing were completed, we developed a small Automated Teller Machine (ATM) program in C and C++ that simulated a single ATM connected to a bank. The C++ version exploited all of the key features of object-oriented technology: information hiding, abstract data types, polymorphism, and inheritance. In our experimentation using both versions, we showed that for the same 102 system-level test cases, the OO version (with encapsulation and information hiding) fared slightly worse in forcing the propagation condition to occur than did the C version. (We say “slightly” worse because there was not an enormous difference between the two; had we used a more complex system with more objects, we suspect that the differences would have been more pronounced.)

As shown in Table 1, our analysis of the C++ version pinpointed eight

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

Table 1: The “before and after” propagation point estimate location scores for OO-ATM.

places in the source code that appeared to be particularly good at error masking. For each of these locations, an appropriate assertion was injected to thwart the error masking, and an assessment of the new error masking capability was reperformed. The net effect of this was that the assertions that were added to ATM forced each propagation probability estimate to increase to 100% from scores near 0%, which is a remarkable increase given the 102 test cases. Since increased propagation estimates mean reduced error masking, which decreases the possibility of residual faults, we were able to thwart the negative effect of the OOD via a handful of intelligently-placed assertions. To be fair, we should state that there were regions in the C version that too would have benefited from assertion injection, but we did not concentrate effort on injecting assertions into that version. Had we, the results would have been an equivalent increase in the software’s ability to propagate errors.

The results from our study are contained in a technical report that was provided to The National Institute of Standards and Technology in December, 1994. Recommendations for how to maintain sufficiently low error masking while maintaining object-oriented design are detailed in the report. The key recommendations were:

1. Assertions represent a valuable aide to software testing. It can easily be shown that an assertion can *never* decrease the propagation condition, i.e., an assertion can only improve or make no change to the propagation

condition, which is very important for thwarting the negative impact of information hiding. Our conclusion that assertions are beneficial parallels the recommendations of Osterweil and Clarke in (Osterweil and Clarke, 1992), where they classified assertions as “among the most significant ideas by testing and analysis researchers.”

2. Information hiding and encapsulation are detrimental to state error propagation, which is very necessary if faults are to be found via software testing.
3. Abstract data types have little impact on software error masking as far as we can determine.
4. Inheritance is not necessarily detrimental to error masking; however, when combined with information hiding, it may become lethal. Unit testing costs increase as the depth of inheritance increases. This is directly related to the increase in the number of drivers and stubs. As a counter argument, subclasses tend to become simpler in deep, complex inheritance trees, and will therefore increase the ability to assess high reliability of the subclasses. This presents another related problem: composing reusable subclasses that are reliable.
5. Polymorphism is difficult to test, i.e., find test cases to exercise different binding scenarios. However, from our previous intuition and this effort’s experimentation, polymorphism, when faulty, will likely cause the faults to be of larger sizes, suggesting decreased error masking. Therefore, polymorphism is not problematic for system-level testing.

Conclusions

Our OO testing research explored an error-masking-based approach to assertion placement. Our effort showed, as expected, that information hiding and encapsulation can be detrimental to system-level and integration testing, but not always. For example, suppose that information that is deliberately hidden and accidentally corrupted corrupts other information that is not hidden, i.e., information that is allowed to freely flow throughout the program. Here the propagation condition will occur for the hidden information by “piggy-backing” on the good fortunes of a second party. The moral here is that

you cannot simply assume that information hiding enables error masking; sometimes it will, and sometimes it will not. Fortunately, methods exist for deciding when it will and when it won't.

For brevity, we have side-stepped several important, problematic issues with assertions: their intrusive nature, and how do you derive correct assertions? For instance, what if your assertions aren't able to recognize problems that have corrupted the state? Or equally annoying, what if your assertions trigger problems that are not there? There are no quick solutions here, and for some software systems, assertions will not be plausible. But for the majority of software systems, the advantages of intelligently-placed assertions should not be overshadowed by the limitations.

On a final note concerning our research with software assertions, we leave the reader with a hypothesis we have been considering: "Assertions that are injected in a program to boost the fault revealing effectiveness of test scheme D cannot lower the fault revealing effectiveness of a different scheme D' ." We do not know how often this will hold, and we can quickly derive counter-examples where it doesn't. But if it does hold in general, it could move us away from decades of debate over which testing technique is better. Instead it could move research efforts toward methods such as assertion instrumentation that add value to virtually any testing scheme. It is our contention that this could provide a key paradigm shift for software testing practice and research.

References

Osterweil, L. and Clarke, L. (1992) A Proposed Testing and Analysis Research Initiative. *IEEE Software*, pp. 89–96, September.