Assertive Testing

Gerard J. Holzmann

Jet Propulsion Laboratory, California Institute of Technology

A colleague asked me recently: “Are there any generally accepted methods for accurately predicting the reliability of software?” Sadly, the honest answer is “no.” Surely there are generally accepted, and practiced, methods, but no one would claim that they can make accurate predictions. And if the predictions aren’t accurate, how useful are they really?

If the above sounds overly pessimistic it is because the question is phrased more or less as an absolute. Instead of asking if there are methods that can predict reliability accurately, it is perhaps more helpful to ask if there are methods that can *improve* reliability. Here we are on firmer ground. There are indeed generally accepted methods that can measurably improve software reliability. Software testing is an obvious example of such a method, but not the only, and perhaps not even the best such method.

How can we measure software reliability? Is there a generally accepted metric? A familiar dictum is “If you cannot measure it, then you cannot manage it.”

Reliability clearly has something to do with the absence of failures. A common approach is therefore to define reliability by measuring its opposite: the probability of failure. It is similar to trying to define “health” as the absence of illness. If you’re healthy, the probability that you get sick in some interval of time should be small, although it likely will never really be zero. So it is for software.

To measure the reliability of a software application, then, we can try to express the rate of discovery of defects in that application as a probability.

If, for instance, the long-term probability of an application exhibiting a failure is p, then the reliability of that application (meaning the probability of failure-free operation) can be given as ($1-p$). If p is 10^{-9} per hour of operation, then we should not expect to see more than one failure per one hundred thousand years of operation, which should satisfy even the most demanding types of applications.

That target of 10^{-9} failures per hour of operation, though, can be extraordinarily hard to reach. A recent government report, for instance, specified the required period of failure free operation for conventional takeoffs and landings of the F35 joint strike fighter not as a hundred thousand years, but as six hours [1]. This corresponds to an average probability of failure about eight orders of magnitude larger than 10^{-9}. And, as the report also noted even that target of an average of six hours of failure free operation for this capability had not yet been realized.

Software failures are caused by coding or design defects that *could* have been caught if the right type of check had been performed before an application was released for general use. For a commercial company it is often not cost-effective to try to chase down every last bug before a product is shipped. This means that in a fixed period of time and with a fixed budget for testing, only the more likely types of defects are caught. The remainder is commonly referred to as the set of “latent defects” in an application.

It will not surprise anyone to learn that the number of latent defects in any non-trivial application typically outnumbers the number of discovered defects by a large margin, no matter how long an application has been in use. Of course, the more users there are, and the longer an application is used, the more of the latent defects will be found.

Probability and Impact

We can categorize software defects in two different ways. We can look at their probability of occurrence, or we can look at their potential impact, as illustrated in Figure 1.

The majority of software defects are generally minor glitches that have no significant impact on the end-user, although they can of course impact that user’s perception of code quality. Those defects fall on the left side of the vertical line in Figure 1. The glitches that are most likely to occur, in the upper-left quadrant, are reliably caught in a standard software test regimen. The more problematic software

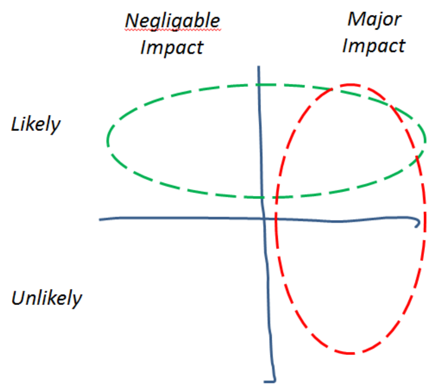
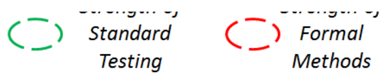


Figure 1 Probability and Impact



defects are the ones that do have significant impact. Again, the ones that are likely to strike, in the upper-right quadrant of Figure 1, can be expected to be caught early. That leaves the set of the troublesome lower-probability defects with significant impact, in the lower-right quadrant of Figure 1.

An uncomfortably large proportion of the major software failures that we learn about with some regularity tends to fall into this lower-right quadrant. Often failures of this type are caused by unexpected combinations of low probability events that can push a system beyond its design limits. The failure of a hardware component, for instance, could occur during the execution of a fault handling procedure for some unrelated off-nominal event, and all of a sudden the system can enter a failure mode that was never tested.

It is generally not a good idea to ignore potential failures simply because their probability of occurrence is deemed low. As C. Michael Holloway, a researcher at NASA Langley Research Center, once said: *“To a* *first approximation, we can say that accidents are almost always the result of incorrect estimates of the likelihood of one or more things."* We are good at estimating consequences, but we are bad at estimating probabilities.

Formal methods target the discovery of this type of low-probability but major impact defects. Compared to standard software testing methods, though, they can be harder to use. For critical systems, therefore, the use of a formal methods approach is often restricted to a relatively small number of critical modules. But is there then no middle ground between a pure formal methods approach that leaves no stone unturned, but requires more skill, and a more routine approach to software testing? There is, and it is what I will talk about next.

Getting Testy

Let us first consider how standard approaches to software testing can be made more thorough simply by providing a little more structure and diversity. I'll mention just some of the many possible techniques of this type that can improve the effectiveness of a test suite.

First then, let's talk about ways to structure a software test beyond the familiar phases of unit, system, and acceptance testing. A more structured approach consists of five additional steps that can be used in each of the standard phases of testing.

In the first of these steps, the code is tested under *ideal* conditions, to make sure that at the very least it *can* behave as designed. In the second step, when the code passes this test, it is tested under *nominal* conditions. That is, under the conditions that it can be expected to encounter in normal day-to-day use. Next, in a third step, it should be tested for the correct handling of *boundary* conditions, where the code is exercised at the edge of its operational profile. Fourth, the code should be tested under *stress* or *overload* conditions. Finally, in a fifth step, it should be tested for the correct handling of all conceivable *error* conditions, such as invalid inputs and ideally also for different combinations of component failures. Note that error handling code is often the least thoroughly tested part of any software system, and therefore also the most likely to contain latent defects. This is precisely the part of the system that you would like to be the most robust, but it rarely is. An effective technique in this stage is to use test randomization, also known as *fuzz testing*, which has proven to be remarkably effective in finding the unsuspected breaking points of a software system.

Another method to improve the rigor of software testing is to use a model-based approach: *model-based testing*. The key idea here is that either the system engineer or the software developer constructs a high-level model of how the software is intended to work. This high-level model can then be used to derive, often mechanically, a suite of test-cases. The model should encapsulate as many software requirements as possible, which means that the tests can check that the requirements are met. If the tests that are generated from the high-level model do not cover some of the code of an application, this indicates that the model is incomplete, and should be extended. Or, it could indicate that the software contains too many parts that are unrelated to the actual software requirements, which may mean that they should be deleted to allow the code base to slim down to a more manageable (and testable) size.

In running the tests we now look for cases where the test results differ from the model's predictions. In this case there is a problem with the model, with the software, or with the requirements. Another benefit from a model-based approach to software testing is that it can make it easier for formal methods types like me to perform also more rigorous types of software verification, for instance with the help of logic model checking tools.

Assert Yourself

Another way to improve the thoroughness of a software test, and with it the reliability of the target software application, is relatively simple: use assertions. As a rule of thumb, you should aim for an average assertion density across all of your code of two percent or more. If you follow this rule you are not alone: it is also followed by Microsoft in its Office software suite [2], and it is currently used at NASA/JPL in its development of mission critical flight code.

The use of assertions can make sure that you catch defects at the earliest possible point in an execution, not only during normal system test phases, but also later, when your code has reached a customer.

You can, for instance, place an assertion inside the body of every loop in the code, to make sure that a reasonable maximum number of iterations is never exceeded. You'd be surprised how many bugs this one measure can catch early in software development. If you're unsure about the upper-bound you should use, multiply your most generous guess by a thousand or more. The real problem you are defending against is an execution getting stuck in an infinite loop, for instance when a linked list accidentally becomes circular.

Another good strategy is to place an assertion before every division operation, to make sure you are not accidentally dividing my zero, or by a number that is very close to zero. Similarly, place an assertion before pointer dereference operations, to check that the dereference operation cannot cause a crash. You can use assertions similarly to check that parameters passed to a function are in a safe range, or that the result passed back to a caller passes a sanity check. And if you are worried that in a time critical system you cannot afford the cost of evaluating a few extra Boolean conditions, you are operating too close to the margin and should take this as an indication that it is time to refactor the code. Note that no policeman will be persuaded either if you try to claim that you had no time to put on your seat belts or to stop at a traffic light.

Statement Coverage

A common goal in testing, inspired by guidelines such as DO-178B/C, is to make sure that all your tests combined secure full statement and branch coverage. This means that each statement in your code must be exercised by at least one test, and every clause in every conditional test must independently evaluate to true and to false, each in at least one test. What is sometimes forgotten is that it is not enough to merely *execute* a statement: a test must also actually *check* something. This is where the use of assertions can again prove their value: they provide at least some additional independent checks of the sanity of an execution.

The insight that assertions can be effective to make systems more reliable isn't new of course. The familiar include file <assert.h> with a definition of a few macros to support the use of assertions in C code was added to the Unix C compilers as early as in 1978. A little bit of sleuthing helped me find that it was Mike Lesk (also responsible for the Unix tools *lex* and *uucp*) who first added this file, as one of several improvements he made to the C preprocessor at the time.

An *assert* keyword appeared a few years earlier in the language definition for Algol-W from 1972. The earlier language report on Algol-68, to which Algol-W was in many ways a response, also contained a notation for defining inline assertions. They were called *pragmats* in the *"Revised Report on the Algorithmic Language Algol 68."* Like modern day *pragmas* in C code, though, they were technically outside the language definition, and could freely be ignored by the compiler. Earlier still, we find references to the importance of assertions in the writings of both Alan Turing and John von Neumann, as also noted in Clarke & Rosenblum's study [3].

So now it's your turn. Are there any tests in your regression test suite (you do have one, don't you?) that fail to execute any assertions? You can strengthen your test suite by making sure that they all do.

And, oh yeah, don't disable those carefully crafted assertions when you ship a product to your customers. Microsoft doesn't do so in its Office tool suite, and neither does JPL when its embedded software hitches a ride to Mars. The assertions can help you detect, diagnose, and fix the latent defects hiding in your code before they can do any harm. In a sense, removing or disabling software assertions before a system is shipped to the customer would make as much sense as a car maker removing the seatbelts and airbags from a car after they've completed their internal testing.

References

1. *F-35 Joint Strike Fighter – Problems Completing Software Testing May Hinder Delivery of Expected Warfighting Capabilities*, US Government Accountability Office (GAO), Report to Congressional Committees, GAO-14-322, March 2014, pg. 18. URL: http://www.gao.gov/assets/670/661842.pdf
2. C. A. R. Hoare, *Assertions: a personal perspective*. IEEE Annals of the History of Computing 25(2): 14-25 (2003).
3. L.A. Clarke, and D.S. Rosenblum, *A historical perspective on runtime assertion checking in software development*. ACM SIGSOFT, Software Eng. Notes, May 2006, Vol. 31, No. 3, p. 25-37.

Acknowledgement

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.