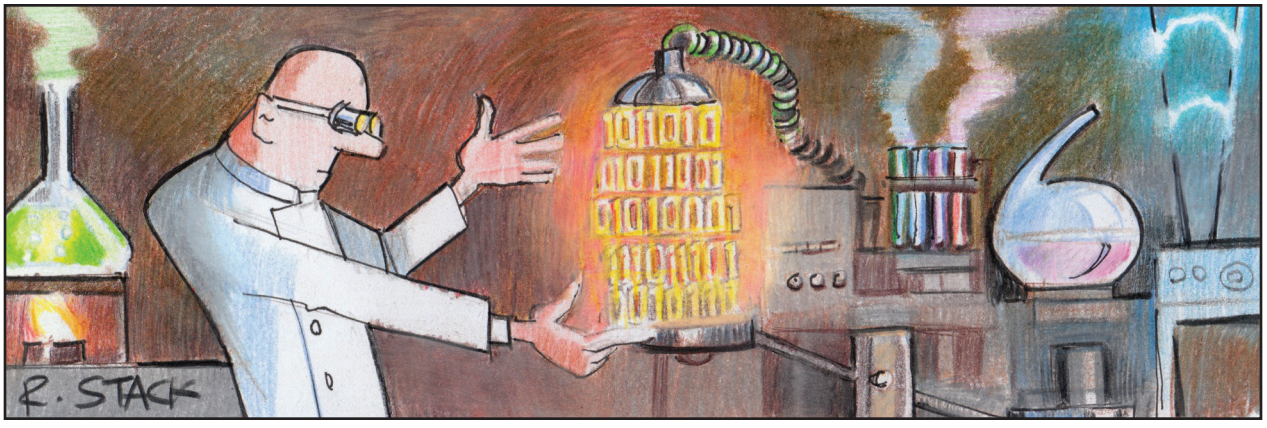
Editor: **Gerard J. Holzmann**
NASA/JPL
gholzmann@acm.org



Assertive Testing

Gerard J. Holzmann

A COLLEAGUE ASKED me recently, "Are there any generally accepted methods for accurately predicting software reliability?" 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 you can't measure it, you can't manage it.

If that sounds overly pessimistic, it's because the question was phrased more or less as an absolute. Instead of asking whether methods exist that can predict reliability accurately, it's perhaps more helpful to ask whether methods exist that can improve reliability. Here we're on firmer ground. Indeed, generally accepted methods exist that can measurably improve reliability. Software testing is an obvious example of such a method, but not the only, and perhaps not even the best, such method. Here, I look at simple, effective ways to augment standard software testing.

Measuring Reliability

How can we measure software reliability? Does a generally accepted metric exist? A familiar dictum is "If you can't measure it, you can't 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. This is similar to trying to define health as the absence of illness. If you're healthy, the probability that you'll get sick in some interval of time should be small, although it likely will never be zero. So it is for software.

To measure a software application's reliability, then, we can try to express the rate of discovery of defects that might lead to failure as a probability.

For instance, if the long-term probability of an application exhibiting a failure is p, that application's reliability (the probability of failure-free operation) is $1 - p$. If p is 10^{-9} per hour of operation, we shouldn't expect to see more than one failure per 100,000 years of operation on average, which should satisfy even the most demanding applications.

Reaching that target of 10^{-9} failures per hour can be extraordinarily difficult. For instance, a recent government report specified the required

 0740-7459/15/$31.00 © 2015 IEEE

period of failure-free operation for conventional takeoffs and landings of the F35 Joint Strike Fighter not as 100,000 years but as six hours.[1] This corresponds to an average probability of failure about eight orders of magnitude larger than 10^{-9}. The report also noted that this target hadn't yet been realized.

Latent Defects

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's often not cost-effective to chase down every last bug before a product is shipped. This means that in a fixed time period and with a fixed testing budget, only the more likely types of defects are typically caught. The remaining bugs are commonly called *latent defects*.

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

Probability and Impact

We can categorize software defects by their probability of occurrence or potential impact (see Figure 1).

Most defects are minor glitches that don't significantly affect users, although they can of course negatively affect the users' perception of code quality. Those defects fall on the left of the vertical line in Figure 1. The most likely glitches, on the upper left, are reliably caught in a standard software test regimen. The more problematic software defects are those that do have a significant impact. Again, the ones that will likely strike, in the upper right of Figure 1, can be expected to be caught early. That leaves the set of lower-probability defects with potentially significant impact, in the lower right 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, such failures are caused by unexpected combinations of low-probability events that can push a system beyond its design limits. For instance, the failure of a hardware component can occur during the execution of a fault-handling procedure for some unrelated off-nominal event. All of a sudden, the system can then enter a failure mode that was never tested.

It's 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, 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."[2] We're good at estimating consequences, but we're bad at estimating probabilities.

Formal methods target the discovery of these 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 formal methods 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 that's what I talk about next.

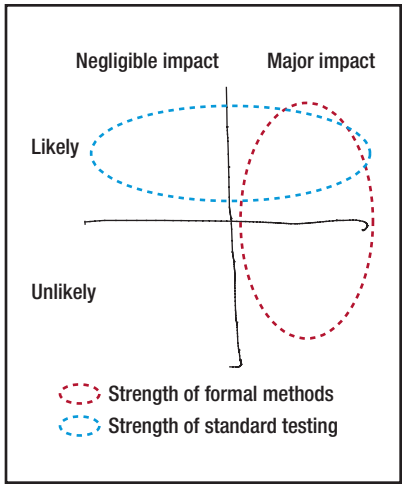


FIGURE 1. The probability and impact of software defects. An uncomfortably large proportion of the major software failures that we regularly learn about tends to fall into the lower-right quadrant.

Getting Testy

Let's first consider how to make standard software-testing approaches 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 a test suite's effectiveness.

You can structure a software test beyond the familiar phases of unit, system, and acceptance testing. A more structured approach consists of five additional steps that you can use in each of the standard testing phases:

1. *Ideal conditions*. Test the code under ideal conditions, to ensure that at the very least it can behave as designed.
2. *Nominal execution*. If the code passes step 1, test it under nominal conditions—the conditions it should encounter in normal day-to-day use.

3. *Boundary cases*. Test the code for the correct handling of boundary conditions, where the code is exercised at the edge of its operational profile.
4. *Stress testing*. Test the code under stress or overload conditions.
5. *Error handling*. Test the code for the correct handling of all conceivable error conditions, such as invalid inputs, and ideally for different combinations of component failures.

Error-handling code is often the least thoroughly tested part of any software system and therefore the most likely to contain latent defects. This is precisely the part of the system you want to be the most robust, but it rarely is. An effective technique in this stage is to use test randomization, also called fuzz testing, which has proven remarkably effective in finding unsuspected breaking points.

Another way to improve the rigor of software testing is to use model-based testing. First, the system engineer or software developer constructs a high-level model of how the software should work. This high-level model can then be used to derive, often automatically, 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 generated from the high-level model don't cover all of the code, the model is incomplete and should be extended. It's also possible that the software contains too many parts that are unrelated to the software requirements. This can mean that you should delete them to slim the code base down to a more manageable (and testable) size.

In running the tests, look for cases in which the results differ from the model's predictions. The problem can be with the model, the software, or the requirements. Model-based testing can also make it easier for formal-methods types like me to apply more rigorous forms of software verification—for instance, with the help of logic-model checkers.

Another way to improve the rigor of software testing is to use model-based testing.

Assert Yourself

Another way to improve the thoroughness of a software test, and with it the reliability of the target application, is relatively simple: use assertions. As a rule of thumb, aim for an average assertion density of one to two percent across all your code. If you follow this rule, you won't be alone: Microsoft follows it in the Office software suite,[3] and NASA's Jet Propulsion Laboratory (JPL) uses it in the development of its mission-critical flight code.

Using assertions can ensure 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 the end user.

For instance, you can place an assertion in the body of every loop in the code, to ensure 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 what upper bound to use, multiply your most generous guess by a thousand or more. The real problem you're 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 ensure you're not accidentally dividing by zero or a number very close to zero. Similarly, place an assertion before pointer dereference operations, to check that they can't 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 returned to a caller passes a sanity check. If you're worried that in a time-critical system, you can't afford the cost of evaluating a few extra Boolean expressions, you're operating too close to the margin. You should take this as an indication that it's time to refactor the code. No policeman will be persuaded either if you claim that you had no time to stop at a red traffic light.

Statement Coverage

A common goal in testing, inspired by guidelines such as DO-178B/C (which deals with software safety for airborne systems), is to ensure 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 in at least one test. What's sometimes forgotten is that it's not enough to merely execute a statement; a test must also actually check something. This is where assertions can again prove their value: they provide some additional independent checks of an execution's sanity.

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

An assert keyword appeared earlier in the 1972 definition of Algol W. The 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*.[4] Like modern 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 Lori Clarke and David Rosenblum noted.[5]

So now it's your turn again. Does your regression test suite (you do have one, don't you?) have any tests that fail to execute assertions? You can strengthen your tests by ensuring 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 Office, 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 in your code before they can do harm. In a sense, removing or disabling software assertions before shipping a system to customers would make as much sense as a car maker removing the seatbelts and airbags from a car after all crash tests have been completed.

References

1. *F-35 Joint Strike Fighter: Problems Completing Software Testing May Hinder Delivery of Expected Warfighting Capabilities*, GAO-14-322, US Government Accountability Office, Mar. 2014, p. 18; www.gao.gov/assets/670/661842.pdf.
2. C.M. Holloway, "Why You Should Read Accident Reports," presentation at the Software and Complex Electronic Hardware Standardization Conf., 2005.
3. C.A.R. Hoare, "Assertions: A Personal Perspective," *IEEE Annals of the History of Computing*, vol. 25, no. 2, 2003, pp. 14–25.
4. A. Van Wijngaarden, B.J. Mailloux, and J.E.L. Peck, *Revised Report on the Algorithmic Language Algol 68*, Springer, 1976.
5. L.A. Clarke and D.S. Rosenblum, "A Historical Perspective on Runtime Assertion Checking in Software Development," *ACM SIGSOFT Software Eng. Notes*, vol. 31, no. 3, 2006, pp. 25–37.

GERARD J. HOLZMANN works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.