Software development and testing is an error-prone process. Despite developers’ best attempts at checking every detail, errors made during development often cause postrelease software failures. Software reliability theory is one of industry’s seminal approaches for predicting the likelihood of software field failures. Unfortunately, the assumptions that software reliability measurement models make do not address the complexities of most software, resulting in far less adoption of theory into practice than is possible.

Even though reliability models are quantitative, the industry only uses these results qualitatively. An example of this use is testing. When the mean time to failure (MTTF) falls below X according to a particular reliability model, testing stops, but developers and users cannot assume that the software will always behave with an MTTF less than X in the field.

Software reliability theory appears to work accurately in telecommunications and aerospace. There are two reasons for this successful use. First, governments essentially regulate product quality in these two fields: Telephones must work and airplanes must fly. In other disciplines, such as the production of commercial, shrink-wrap software, quality has historically been an add-on, of lesser market value than feature richness or short release cycles.

Second, telecom and aerospace software works in an embedded environment, and in many ways it is indistinguishable from the hardware on which it resides. Because the software assumes many characteristics of its hardware environment, it is not a huge leap to apply hardware reliability directly to embedded software.

Today, accurate quality measurement can no longer be confined to particular industries; it is especially needed in shrink-wrap software. The ubiquity of network computing has brought quality concerns to the forefront. Buggy operating systems, Web browsers, and even client-side desktop applications can cause security holes. Bugs provide entry points for viruses and cause denial of service. Hackers can exploit buffer overruns to execute malicious programs. Because consumers are starting to demand it, quality is a concern for any company that develops software designed to run on networked computers.

Even after a decade of practice in the confines of the hardware-dominated telecom and aerospace worlds, we wonder if the wider audience is ready for software reliability theory. Successful case studies and a thriving research community...
coexist with skepticism and doubt among potential practitioners. For example, the US Federal Aviation Administration’s standard for developing software embedded on airplanes (RTCA DO178-B) concludes “currently available (reliability) methods do not provide results in which confidence can be placed.” Of course, skepticism should shadow new ideas, but software reliability is understood in only a small sector of the software industry.

Our goal is to create a dialogue in the reliability community and to identify a base of technology that will widen interest in software reliability among practitioners outside the telecom and aerospace domains. We also propose a new method for software reliability research based on additional considerations that affect the software’s behavior.

SOFTWARE RELIABILITY DEFINED

Much of what developers call software reliability has been borrowed or adapted from the more mature field of hardware reliability.1,2 The influence of hardware is evident in the current practitioner community where hardware-intensive systems and typical hardware-related concerns predominate.

Two issues dominate discussions about hardware reliability: time and operating conditions. Software reliability—the probability that a software system will operate without failure for a specified time under specified operating conditions—shares these concerns.3,4 However, because of the fundamental differences between hardware and software, it is legitimate to question these two pillars of software reliability.

Rethinking the notion of time

When testing or using hardware, calendar time is a central concern. Hardware wears over time. Its couplings loosen and its parts overheat, becoming less reliable. Eventually, it will fail from wear. Users appreciate this reality. They purchase limited hardware product warranties that expire after a number of years and accept that repair or replacement costs are inevitable. Perfect hardware simply doesn’t exist.

Software doesn’t fit this way of thinking. Software can be perfect and remain perfect. Software fails because of embedded defects that didn’t surface during development and testing.

The reliability community should ask, “How much of a role does time play in exposing embedded software defects? Does it matter that we’ve tested an application for two weeks or two months?” The answer, of course, is “it depends.” Two months gives us more opportunity to test variations of input and data, but length of time is not the defining characteristic of complete testing. If we are good testers, two months may be far too much time. If we are weak testers, two months may be insufficient.

Consider the software that controls rocket attitude. If you were buying a rocket launch, you would want to know whether the hardware would survive long enough. But you’d also want to know if the software has been tested for every launch scenario that seems reasonable and for as many scenarios as possible that are unreasonable but conceivable. The real issue is whether testing demonstrates that the software is fit for its duty and if testers can make it fail under realizable conditions.

Alternative criteria

What criteria could better serve software developers? Consider the ubiquitous word processor. If you test a word processor for 2,000 hours, what does it tell you? Is that a lot or a little? The answer, once again, is, “It depends.”

First, it depends on the complexity of the word processor. If you are considering a simple text editor without fancy features like table editing, figure drawing, and macros, then 2,000 hours might be a lot of testing. For modern, feature-rich word processors, 2,000 hours is nothing.

Second, it depends on the completeness of testing. If during those 2,000 hours the software sat idle or the same features were tested over and over, then more testing is required. If testers ran a nonstop series of intensive, minimally overlapping tests, release might be justified.

The contemporary definition of software reliability based on time-in-test assumes that the testers fully understand the application and its complexity. The definition also assumes that testers applied a wide variety of tests in a wide variety of operating conditions and omitted nothing important from the test plan.

As Figure 1 shows, most reliability growth equations assume that as time increases, reliability increases, and the failure intensity of the software decreases. Instead of having a reliability theory that makes these assumptions, it would be better to have a reliability measure that actually had these considerations built into it.

The notion of time is only peripherally related to testing quality. Software reliability models typically ignore
application complexity and test coverage. Instead, they use a formula similar to this:
The reliability at time $t$ is given by

$$R(t) = \exp\left[\int_0^t -\lambda(\tau) d\tau\right]$$

where $\tau$ is just the variable of integration and $\lambda(\tau)$ is the failure rate. $R(t)$ is simply the probability that a failure will occur at time $\tau$ given that no failure occurred before time $t$.

Which parameters of this model account for application complexity? Which parameters account for the thoroughness of testing? There aren’t any. This formula treats every application and test case uniformly. All that it accounts for is successful operating time and some data points representing software failure.

For software reliability theory to be universally applicable to all types of software and to produce repeatable and accurate results, testers need to understand the complexity of the applications they are testing and how thoroughly they are testing them. Not only must we understand these things, but we also must be able to quantify them. If we want to continue considering time as a parameter of future reliability models, CPU time is probably more appropriate than calendar time.

Rethinking the operational profile

Usage is a central concern when testing or using hardware. Hardware warranties often contain disclaimers about unintended usage resulting in more than normal wear and tear. If you submerge a toaster in a swimming pool, the warranty is voided. The manufacturer regulates the operational profile for toasters, and the expectation is that usage will be appropriate.

What is the operational profile for software? Two classic software reliability references define it as the set of operations that software can execute along with the probability with which they will occur.$^{3,6}$

Anyone who uses software appreciates that there are two types of features: tried-and-true features that usually work and fancy ones that cause trouble. Using software is like walking in a jungle. As long as you stay on the well-beaten path, you shouldn’t get eaten by a lion, but if you stray from the path, you are asking for trouble.

This is a big source of developer frustration. Users won’t stay on the path, even when highly constrained by user interfaces. The software reliability argument against this concern is one for which no counter-argument can prevail—certified reliability is valid only for the profile used in testing. In other words, stay on the well-worn path or all bets are off.

There is an even more fundamental issue at hand. To specify an operational profile, you must account for more than just the primary system user. The operating system and other applications competing for resources can cause an application to fail even under gentle use by a human. Software operating environments are extremely diverse and complex. A simple distribution of inputs from a human user does not come close to describing the situation.

For example, even gentle use of a word processor can elicit failure when the word processor is put in a stressed operating environment. What if the document being gently edited is marked read-only by a privileged user? What if the operating system denies additional memory? What if the document auto-backup feature writes to a bad storage sector? All these aspects of a software environment can cause failures even though the user follows a certified usage profile. In this case, the well-worn path isn’t safe—lions are everywhere.

Most software applications have multiple users. At the very least, an application has one primary user (a human) and the operating system. Singling out the human user and proclaiming a user profile that represents only that user is naive. That user is affected by the operating environment and other system users. A specified operational profile should include all users and all operating conditions that can affect the system under test. This task is not yet possible using any existing profiling technique.

REVISITING SOFTWARE RELIABILITY FUNDAMENTALS

To compute software reliability, testers examine software according to an operational profile, counting and timing success and failure events. In general, the actual mathematics of the reliability model doesn’t account for the complexity of the software under test. Whether it has 10 or 10,000 function points isn’t considered pertinent.

Also, reliability models do not account for any specifics concerning the test cases, except that they represent some real user community. If real use occurs outside the assumed profile, all bets are off. Missing tests, tests repeated too often, and tests that do too little or too much don’t matter to the reliability model, which is a step away from the specifics of both the application and the test data.

There also are protocol problems that software reliability has yet to fully address. One of the most fundamental is software repair and modification. Once software changes, it becomes a completely new product—a related product to be sure, but one that is new. Repairing software might make its quality better, worse, or unchanged. The question is how to deal with software repair in a statistical sense. How do you treat failure recurrence? Do you count regression tests toward reliability? Can you keep old reliability data and append new data, or are you forced to start from...
Reliability is more complex than simply converting the notions of time and operational profile to something you can deal with mathematically. Given these arguments, we believe software reliability research needs to deal with three central issues governing software failure: application complexity, test effectiveness, and the complete operating environment.

Application complexity

Developers cannot abstract the application they are testing out of the reliability equation. How can they possibly count time between failure and claim with confidence that an application is reliable? Looking only at test data and defining random variables that describe it falls short. To get an accurate measure of an application’s reliability, developers must consider the application’s size and complexity.

Reliability models must account for an application’s inherent complexity and consider the amount of testing compared to the size of the testing endeavor.

Test effectiveness

For reliability measurement, test effectiveness depends on the number of test inputs and the diversity of those tests. We contend that this is a better measure of time-in-test than the current notion of calendar or execution time. Using time as a measure of test effectiveness (claiming a certain number of hours in test) begs the response, “So what?” The real question is, “What did testers do during that time?” Instead of counting clock cycles, testers should count inputs from every user and ensure that those inputs are diverse enough to form a complete set of test data. How to actually accomplish this goal is still a matter of research, but it is a problem worth noting.

Complete operating environment

Reliability models assume (but don’t enforce) testing based on an operational profile. Certified reliability is good only for usage that fits that profile. As we argued earlier, the operational profile is only one aspect of the operating environment. Changing the environment and usage within the profile can cause failure. The operational profile simply isn’t adequate to guarantee reliability.

We propose studying a broader definition of usage to cover all aspects of an application’s operating environment, including configuring the hardware and other software systems with which the application interacts. The question really is, “How much of the complete environment did the testers consider?” If the answer is “a lot of it,” testers would have observed more software behaviors during testing, and the resulting reliability predictions would be more believable because they were tied to a fuller description of the usage environment.

For example, you could write a reliability prediction as follows: This software has a reliability of 99.999 percent when it runs on operating system X and is connected to commercial off-the-shelf database Y. We include one cautionary note, however. A recent article noted that if all of the restrictions on using a particular drug were written on the side of the prescription bottle for the patient to read, the pill bottle would have to be two feet high. Similarly, if developers collect too much environmental information and tie reliability scores to that information, they might end up with a reliability score that is only valid for one machine in the universe. That would not help the use of reliability measurement become more ubiquitous.

In addition to the most used features, our view of the operational profile includes the operating environment or system, third-party application programming interfaces, language-specific runtime libraries, and external data files that the tested software accesses. The state of each of these other users can determine whether the system under test fails. If an e-mail program can’t access its database, that is an environmental problem that developers should incorporate into the definition of an operational profile.

We stress the need to do more than simply assign a reliability number and assume that no matter where the software is deployed, users will observe at least that level of quality. We are suggesting something similar to the fine print in a limited warranty or the drug interaction precautions that accompany prescription drugs. While this may sound tedious, users must be willing to learn more about the nuances of the software they license, and consider more precisely all of the assumptions under which developers compute a reliability score.

RESEARCH AGENDA FOR SOFTWARE RELIABILITY

Within the field of software reliability engineering, the focus has been on the mathematics of software reliability and on conducting statistical tests according to operational profiles. Outside the field, technology also exists to understand and measure application complexity, test effectiveness, and software operating environments. We encourage other researchers to help bring these and other new technologies into the field of software reliability prediction.

Measuring application complexity

From a testing perspective, software is complex for the following reasons:
to be useful for creating a more universally applicable reliability theory, code complexity metrics must be a function of the software's semantics and environment.

- There are many inputs to test and outputs to check for correctness.
- It is difficult to execute certain sections of code or to force software to exhibit certain behaviors simply by submitting inputs, such as forcing certain exceptional situations that may require faulty hardware or stressed computing or network resources. Checking to ensure that error handling routines work properly and that they cover the gamut of realizability situations is problematic.
- Several combinations of stored data values and environment conditions define a particular behavior. This means that testers must test a particular behavior repeatedly to ensure that the software functions properly in a wide range of environment and internal settings.
- Software features interact with one another through shared data or message passing. Testers must ensure that each of these features can coexist without failure. So testers must put each feature through its paces individually, in pairs, in triplets, and so on.
- Faults can hide in numerous places. Some software simply does not hide faults well, and nearly any sequence of inputs can find them. In the worst case, only a specific, convoluted series of inputs will uncover a fault and all other input sequences will miss it. The ability of a fault to hide is a crucial characteristic of software complexity. We call this fault size, and the bigger the fault, the easier it is to find during testing.

Measures of software complexity based on source code (Halstead's length measure and McCabe's cyclomatic complexity) are not sufficient to fully determine test complexity. Those measures gauge development complexity, and they fail to capture the variety of software usage (in terms of input sequencing and operating environment). They also don’t account for faults, real or potential. Both usage and failure are central to the model of test complexity. Moreover, the integration of source-based complexity measures with execution-based reliability models is problematic to say the least. Another technique is required.

**Code complexity metrics**

The fundamental problem with trying to correlate code metrics to software quality is that quality is a behavioral characteristic, not a structural one. A 100 percent reliable system can suffer from terrible spaghetti logic. While spaghetti logic may be difficult to thoroughly test using coverage techniques, and while it will be hard to debug, it still can be correct. Further, a simple straight-line chunk of code (without decision points) can be totally unreliable. These inconsistencies make generalizing code metrics impossible.

As evidence that reliability is not a function of code structure, consider the Centaur rocket and Ariane 5 problems. *Aviation Week and Space Technology* recently announced that the Centaur upper-stage failure was caused by a simple programming error. A constant was set to one-tenth of its proper value (−0.1992476 instead of −1.992476). This tiny miscoding of a constant caused the rocket failure. Code complexity was not the issue. The Ariane 5 disaster was caused by failing to consider the environment in which a software component was being reused.8 Once again, the complexity or lack thereof had nothing to do with the resulting software failure.

While it is true that metrics correlate to characteristics like readability or maintainability, they do not correlate well to reliability. Therefore, developers need complexity metrics with respect to the environment in which the code will reside. PIE (propagation, infection, and execution) provides a behavioral set of measures that assess how the structure and semantics of the software interact with its environment.

Consider the three algorithms of the PIE model: propagation analysis, infection analysis, and execution analysis. Execution analysis provides a quantitative assessment of how frequently a piece of code is actually executed with respect to the environment. For example, a deeply nested piece of code, if viewed only statically, appears hard to reach. This assumption could be false. If the environment contains many test vectors that toggle branch outcomes in ways that reach the nested code, executing this code will not be difficult. Similarly, infection analysis and propagation analysis also quantitatively assess the software semantics in the context of the internal states that are created at runtime.

Our point is that code complexity metrics must be a function of the software’s semantics and environment. If they are, they will be useful for creating a more universally applicable reliability theory.

**Measuring test effectiveness**

Given a certain strategy for testing a software system’s various complexities, one question naturally arises: What is the effectiveness of the tests generated by that strategy? In part, we must base an answer to this question on our definition of software complexity.

Because code complexity is a function of the source code, the execution environment, and the software’s ability to hide defects during testing, a theory must judge test effectiveness in all three areas.

Counting source lines of code executed during testing or the number of bugs is not nearly enough. If you claim that a test suite is successful because it found 40 failures, you cannot be certain that there are only 41 failures, and the suite found most of them. There could
be 400 (or even 4,000) failures, indicating that the suite did a lousy job. This type of count doesn’t tell testers what percentage of the faults they have found or how often the software will fail as a result.

Without knowing whether testing has detected all faults during testing (unless testing is exhaustive), testers cannot know when to stop testing. This means that testing could continue after all faults were detected and fixed. This is wasteful but unavoidable.

Instead of attempting to create criteria that would halt testing once it had detected all faults, we suggest a testing stoppage criteria based on test effectiveness. We develop a set of guidelines for creating a good test suite, and when we finish testing according to that suite, we stop testing. This does not mean that all the faults are gone. It means that we used test cases that were more likely to reveal errors than if we had merely generated ad hoc tests.

The question then becomes, “How do we compare tests to rank their ability to detect faults?” We could answer this if we knew where the real faults were, but if we don’t, is there any hope?

**Software fault injection**

Software fault injection offers some hope. This technique simulates a wide variety of anomalies, including programmer faults, human operator errors, and failures of other subsystems (software and hardware) with which the software under test interacts.

For this problem, however, we are interested only in simulating programmer faults. This provides a testing stoppage criteria based on test effectiveness. One of the earliest applications of software fault injection was mutation testing. Mutation testing builds a test suite that can detect all simulated, syntactic program faults. Because there are multiple definitions of what it means to detect all simulated syntactic programmer faults, there are multiple types of mutation testing. Once mutation testing builds the test suite, the suite is used during testing.

Simulated programmer faults are nothing more than semantic changes to the code itself. For example, changing $a = a + 1$ to $a = a - 1$ is a simulated fault. By making such modifications, testers can develop a set of test cases that distinguish these mutant programs from the original. The hypothesis is that test cases that are good at detecting hypothetical errors are more likely to be good at detecting real errors.

Using fault injection to measure test effectiveness is a two-step process. The first step builds test suites based on the effectiveness of test cases to reveal the simulated faults. The second step uses the test cases to test for real faults.

Note one subtlety here, however: Just as all test cases are not equally effective for fault detection, not all simulated faults are of equal value. This brings us to the notion of fault size. The size of a real fault (or simulated fault) is simply the number of test cases needed to detect the fault. When we inject a large simulated fault, most test cases can catch it. Therefore, it is more beneficial to inject smaller faults and create a test suite that reveals them. Small faults are harder to detect, and 50 test cases that detect tiny faults are more valuable than a 50-member test suite that catches only huge errors. A test that detects small errors will almost certainly detect huge errors. The reverse is not necessarily true. Therefore, given two suites of equal size, the one that detects smaller faults reveals more about the software’s quality than a suite that only reveals large errors. The former suite has greater test effectiveness.

Now we map fault injection back to our proposal for a new software reliability theory. When one test suite is more effective at catching faults, it reveals more information about the code’s reliability because it reveals an absence of both large and small faults. The other suite reveals an absence of large faults. For this reason, we contend that reliability theory must account for test effectiveness as opposed to considering only test suite size, as is the practice today.

**Modeling complete operating conditions**

Software does not execute in isolation; it resides on hardware. Operating systems are the lowest-level software programs we can deal with, and they operate with privileged access to hardware memory. Application software cannot touch memory without going through the operating system kernel. Device drivers are the next highest level. Although they must access hardware memory through an operating system kernel, device drivers can interact directly with other types of hardware, such as modems and keyboards. Application software communicates with either device drivers or an operating system. In other words, most software does not interact directly with humans; instead, all inputs come from an operating system and all outputs go to an operating system.

Too often, developers perceive humans as the only user of software. This misconception fools testers into defining operational profiles based on how human users interact with software. In reality, humans interact only with the drivers that control input devices. For example, the Win32 application programming interface, not application programs, actually handles user input in the popular Windows platform. The target software gets inputs that originate from humans, other software programs, the file system, or directly from the operating system itself.

The current practice for specifying an operational profile is to enumerate input only from human users and lump all other input under abstractions called

---

The software we think we are using actually gets our input by receiving events from the operating system. Most software receives input only from other software.
environment variables. For example, you might submit inputs in a normal environment and then apply the same inputs in an overloaded environment. Such abstractions greatly oversimplify the complex and diverse environment in which software operates.

Consider the sequence of interactions that a word processor might have with its environment upon invocation. This single user input causes a flood of interaction between the application and the other users in its environment. The word processor allocates memory for program execution and data storage; opens, reads, and closes files; and explicitly loads interfaces to other resources. Treating each adverse condition as a single abstract environment variable is not only inaccurate but also dangerous because such a viewpoint can overlook too many failure scenarios.

The industry must recognize not only that humans are not the primary users of software, but also that they often are not users at all. Most software receives input only from other software. Recognizing this fact and testing accordingly will ease debugging and make operational profiles more accurate and meaningful. Operational profiles must encompass every external resource and the entire domain of inputs available to the software under test.

One pragmatic problem is that current software testing tools are equipped to handle only human input. Sophisticated and easy-to-use tools to manipulate GUIs and type keystrokes are abundant. Tools capable of intercepting and manipulating software-to-software communication fall into the realm of hard-to-use system-level debuggers. It is difficult to stage an overloaded system in all its many variations, but it is important to understand the realistic failure situations that may result.

THE ROAD AHEAD

Fortunately, these problems have not affected the quality of existing products because software reliability measurement is far from ubiquitous. The problem is that we cannot accurately quantify how good (or bad) it is because existing reliability models are based on an incomplete model of how software really functions. Software operates in multiuser environments in which available resources (from the operating system, file system, or third-party application programming interfaces) can suddenly become unavailable, system loads can sharply increase, or users can instantly change usage patterns.

Such situations can have a significant impact on the quality of service an application provides. Software reliability modeling must account for these situations. In fact, we’ve encountered real situations in which these other dimensions of software usage have affected the software’s reliability more than the current definition of an operational profile allows.

We do not yet have a reliability equation that incorporates application complexity, test effectiveness, test suite diversity, and a fuller definition of the operational profile. We challenge the software reliability community to consider these ideas in future models. The reward for successfully doing so likely will be the widespread adoption of software reliability prediction by the majority of software publishers.

References


James A. Whittaker is an associate professor of computer science at the Florida Institute of Technology, Melbourne, and the director of the Center for Software Engineering Research. His research interests include software testing, computer security, and software reliability. Whittaker has a PhD in computer science from the University of Tennessee. He is a member of the ACM and the IEEE Computer Society. Contact him at jw@cs.fit.edu.

Jeffrey Voas is a cofounder and chief scientist of Cigital. His research interests include software dependability assessment, software certification, and software testing. Voas has a PhD in computer science from the College of William and Mary. Contact him at voas@cigital.com.