Testing Maps The Road To Confidence-From-Use

Electronic Design

Chris Hobbs

Chris Hobbs, Contributing Technical Expert

Fri, 2014-03-07 09:13

The role of testing for software-based systems has changed significantly in the past few years. Last year, ISO/IEC 29119 emerged for software testing. But most test groups already had adopted the change toward risk-based rather than requirement-based testing, either tacitly or explicitly. The testing of a software system now can be seen as a means of producing evidence of confidence-in-use rather than a demonstration of the correctness of the system.

The verification of an embedded system that includes software faces several challenges. For instance, the state space of any executing program is so large, 100% test coverage is impossible. Testing, then, must be seen as a tiny statistical sampling. Even a superficially trivial program can have more states than there are protons in the universe.

Related Articles

- Understanding ISO 26262 ASILs
- Clear SOUP And COTS Software Can Reliably Serve Safety-Critical Systems
- The Limits Of Testing In Safe Systems

In light of the state space size, the effect of even the simplest change to a piece of software or its operating environment (processor, clock speed, etc.) is unpredictable. Embedded systems, particularly those used in mission-critical or safety-critical systems, traditionally have been verified in part by "proven-in-use" figures: "We have had a cumulative 1141 years of failure-free operation of this software, so we can be 95% confident that its failure rate is less than 10^{-7} per hour of operation." However, this claim is valid only if all those years were on identical systems.

The Software State Space

A modern, preemptible, embedded operating system (OS) such as QNX Neutrino or Linux with about 800 assembler instructions in its core has more than 10^{300} possible internal states. To put this into perspective, the Eddington Number (the number of protons in the observable universe) is about 10^{80}.

But this is not all. This OS runs on a processor chip with internal data and instruction caches that might at any time be in many states, all invisible to the user and unreproducible by the tester.

Then there is the application program. Consider the following program, designed to compute 2 + 2 and send the answer to stdout:

```
#include

int main()

  {

  char  x;

  x = 2 + 2;

  printf("%d\n", x);

  return 0;

  }
```

Like many programmers, I don't bother to check the return code from printf()! Many programmers see this as quite a simple program, but, of course, it's not. It's linked to libc to provide printf, and its execution environment includes that OS with at least 10^{300} states. To demonstrate correctness of the 2 + 2 computation, we would have to test it in all those states —clearly impossible.

I have just executed the 2 + 2 program 1000 times on my Linux laptop, and the answer was 4 every time. This, of course, means nothing from the point of view of test coverage. We can't tell whether, during any of those 1000 tests, the timer interrupt on the computer happened to co-incide with the _Lockfileatomic(stdout); instruction in printf, or whether another thread was also trying to lock stdout at the same time as this program. That would certainly put the program into different states. What if some other process had the lock on stdout and failed before completing its operation? Would stdout remain locked and our calculation be indefinitely suspended? That would be a very difficult condition to establish during testing, but should be one of the test conditions.

Proven-In-Use

Recently, Peter Ladkin published a very readable paper, "Assessing Critical SW as 'Proven-in-Use': Pitfalls and Possibilities," that discusses how apparently simple changes to a program's environment can invalidate all previous proven-in-use figures.[1]

In Ladkin's paper, the software for a temperature sensor is transferred from an end-of-life processor to one that is "guaranteed to be completely op-code compatible." Problems occur because the new sensor is more sensitive to temperature than the one it has replaced, though, so it produces interrupts more frequently, overwhelming the program. Although not mentioned by Ladkin, it is fairly certain that the replacement processor also has a different instruction caching algorithm, hidden from the user.

The term *proven-in-use* used in standards such as IEC 61508 and ISO 26262 is unfortunate, because the technique does not involve proving. *Confidence-from-use* is a much more useful term and appears in ISO 26262, but only in the context of tools.

IEC 61508 provides a formula and table for the number of hours of failure-free operation needed to justify a particular confidence in a failure rate. But this assumes that all of those failure-free hours passed under identical circumstances with no change of processor, even if the new processor were "op-code compatible."

Why Do We Test?

Given its number of potential states, any tests of the sophisticated program that computes 2 + 2 cannot do more than scratch the surface of its state space. So why do companies bother with testing at all? A company performing testing on software is effectively dedicating expensive resources to explore far less than 10^{-100}% of the system's possible states. Is that an effective use of resources?

ISO 13849 approaches this from the other direction: "When validation by analysis is not conclusive, testing shall be carried out to complete the validation. Testing is always complementary to analysis and is often necessary." In other words, testing is what we do when other methods of validation have failed.

However, approached correctly, testing may be effective. To take Ladkin's example of the new temperature sensor, the paper says that it failed in the field "about every two weeks." So if the company had bothered to install 10 of them in the laboratory under field conditions for a month, the chances are about 99.999999794% that the problem would have been noticed.

How does this dichotomy between "testing is useless because of the size of the state space" and "testing actually finds problems" arise? Because what is traditionally called testing is actually gaining confidence-from-use. When we ask people in a company's test group what they do for a living, which answer would we prefer?

• "I carry out a sort of digital homeopathy by setting up the random conditions to exercise 0.00000000000000000000000000000000000000001% of the system's possible states."

or:

• "I use my knowledge and experience to produce realistic conditions that stress the system and then create those situations to increase our confidence-from-use. The other day, for example, someone brought me a program that calculates 2 + 2. I looked at it and realized that it had an enormous number of states. Using my engineering judgement, I selected test cases, such as running many copies of it simultaneously to create contention for the stdout lock, that, while not beginning to explore the enormous state space, significantly increased our confidence-from-use."

This latter approach can be seen as fulfilling the idea of *risk-based testing* proposed in the recently published standard, ISO/IEC 29119, on software testing:

"It is impossible to test a software system exhaustively, thus testing is a sampling activity. A variety of testing concepts… exist to aid in choosing an appropriate sample to test and these are discussed… in this standard. A key premise of this standard is the idea of performing the optimal testing within the given constraints and context using a risk-based approach."

In practice, do we have any evidence other than confidence-from-use? Consider the 2 + 2 program again. If it has been in continuous use in the field for 5 million hours ("it" here means the compiled program + libc + the OS + the hardware) without producing any result other than 4, then we can be 95% confident that it has a failure rate of less than 10^{-6} per hour.

Introducing A Change

Given the enormous state space of our 2 + 2 program, it isn't surprising that any modification could completely change its behavior. For example, let's say a new customer needs to know the value of 2 + 3. Could we make a "simple" change to

our program and rely on our proven-in-use data for the 2 + 2 version? Certainly, under test, the program seems consistently to indicate that 2 + 3 = 5, but that may just be coincidence.

Yet in spite of our reservations, we ship the 2 + 3 version and, after some years, our confidence grows. When a customer then comes along with the requirement to compute 2 + 127, we know exactly what to do. Don't we?

That 2 + 127 = –127 rather than 129 is a reminder that any change, however apparently trivial, can significantly affect the program's operation, invalidating the historical data. But given the number of states of software systems, we inevitably ask how anything ever works. We hear of many software-based systems that fail, but, of course, many continue to function correctly.

Part of the answer might lie in the observations from combinatorial testing.[2] Although most systems depend on a large number of parameters (say *N*), each of which has a number of possible states (say *v*), in practice forcing the system through all combinations of a much smaller number of parameters (say *M<) is an effective way of uncovering problems. Even better, although there is no closed-form formula for the number of conditions required, it is known to grow as* $v^M log(N)$*— the growth in log(N) is encouraging.*

It is not clear from the literature whether this result is simply an empirical observation or whether it results from some underlying characteristic of software systems. Either way, it would indicate that most software behavior is linked to a relatively small number of interactions between environmental conditions. This implies that, with the appropriate care, confidence-from-use values can be re-applied to modified systems.

The problem at the moment is that we have no theoretical explanation of "appropriate care." We rely on human knowledge and experience to appreciate the significant difference between turning char x = 2 + 2 into char x = 2 + 3 and turning char x = 2 + 2 into char x = 2 + 127.

Conclusion

We can never claim to have tested our software-based systems completely, nor can we rely on confidence-from-use data gathered on one version of a system to give us figures for a slightly modified version. On the other hand, given the state space of a software system, testing is no longer a scientific discipline. It is simply providing a measure of confidence-from-use. This means that we don't have to discard confidence-from-use figures, but we do have to provide some foundations on how confidence-from-use figures from different systems can be statistically combined in a justifiable manner.

References

1. "Assessing Critical SW as 'Proven-in-Use': Pitfalls and Possibilities," Peter Ladkin,

www.rvs.uni-bielefeld.de/publications/WhitePapers/LadkinPiUessay20130614.pdf

2. Introduction to Combinatorial Testing, Kuhn, Kacker, and Lei, ISBN 978-1-4665-5229-6

***Chris Hobbs** is an operating-system kernel developer at QNX Software Systems, specializing in "sufficiently available" software (software created with the minimum development effort to meet the availability and reliability needs of the customer) and in producing safe software (in conformance with IEC61508 SIL3). He is also a specialist in WBEM/CIM device, network, and service management and the author of A Practical Approach to WBEM/CIM Management (2004). His blog, Software Musings, focuses "primarily on software and analytical philosophy." He earned a BSc, honours, in*

pure mathematics and mathematical philosophy at the University of London's Queen Mary and Westfield College.

Source URL: *http://electronicdesign.com/embedded/testing-maps-road-confidence-use*