Right Code

Gerard J. Holzmann

These days, when you see the headline "How Safe are Airplanes?" the text more likely refers to the spread of disease than the likelihood of a crash. With steady safety improvements over at least the last five decades, airplane crashes have become exceedingly uncommon and they are rarely caused by flaws in the design of the airplane itself. The most common remaining causes are the usual suspects of human error, extreme weather events, and mechanical failure.

That perception of safety for air travel may have changed with the recent crashes of two new Boeing airplanes. The 737 MAX was introduced as a relatively small upgrade in a long line of successful airplane designs, which in this case was primarily meant to improve fuel efficiency.[1] The cause, or rather the causes of the two crashes has been sufficiently fleshed out by now, for instance in the detailed congressional report that was released in September 2020 titled "The design, development, and certification of the Boeing 737 MAX."[2] As the title of this report indicates, the failures cannot be attributed to a single mistake, but were found to have been caused by a sequence of shortcuts and errors, combined with a lack of meaningful oversight by the FAA.

Software Process

The lack of rigor in the vetting of new or revised designs has led to a series of problems, including in the development of the CST-100 Starliner system, Boeing's design for a reusable crewed spacecraft. A serious test failure of the Starliner craft in December 2019[3] was found to have been caused by multiple software bugs in a review by a NASA panel. The panel concluded, among many other findings, that NASA should "go beyond merely correcting the cause of the anomalies," but "scrutinize Boeing's entire software testing processes."

In a review like this, it is always tempting to try to come up with a single remedy that should fix all problems. The NASA study team that is investigating the problems may be leaning in that direction as well, with a recommendation that all software from NASA contractors would need to comply with a new restriction on the cyclomatic complexity of functions [1]. The cyclomatic complexity metric that was popular for a while in the eighties, measures the number of independent paths through the control flow graph of a function. Would it not be great if one could indeed demonstrate the existence of a correlation between cyclomatic complexity and post-release fault density of functions? Quite a few researchers have tried to look for such an effect, but alas most have found that there is no such correlation. So, is there no hope?

---

[1] https://www.defenseone.com/ideas/2019/11/whats-wrong-boeing/161245/ [GH: optional – footnote could be deleted]

[2] https://transportation.house.gov/committee-activity/boeing-737-max-investigation [GH -- optional -- footnote could be deleted]

[3] https://www.reuters.com/article/us-space-exploration-boeing/boeings-botched-starliner-test-flirted-with-catastrophic-failure-nasa-panel-idUSKBN20106A [GH: optional – footnote could be deleted]

Building Reliable Systems

In most disciplines, system reliability is achieved through the judicious use of redundancy. Virtually all spacecraft, for instance, have multiple ways of communicating with earth, using independent transmitters and receivers. Some spacecraft, such as the Cassini spacecraft that orbited Saturn, are designed with two separate main engines and two sets of thrusters, to avoid single points of failure that could end a mission prematurely.

To make data transmissions over noisy transmission channels more reliable, we can add redundant information in the form of error-correcting codes, so that small bursts of errors are recoverable. The big question is of course how we could do something similar in software design. The answer is not to merely duplicate a code base and then run it on two computers. This may protect against hardware faults, but it cannot do the same for the software. If there is a bug in the code, the same bug will obviously affect both executions which then provides no protection at all. This is also the scenario that led to the failure of the maiden flight of the Ariane-5 launch system in June 1996 [2]. The same software was running on both the main CPU and on the backup CPU, which meant that a single floating-point exception error could crash both, and cause a costly loss of the vehicle.

Another method for using redundancy in the design of software systems, called N-version programming, was popular for a while, but also failed to deliver the promised increase in reliability. The thought was to have multiple independent teams write software in parallel, and compare the results of the executions of all systems to detect inconsistencies caused by bugs. As Knight and Levison [3] pointed out in the late eighties, all teams working on the design still work from a common set of requirements and are likely to make similar types of mistakes. Splitting a team into N sub-teams can also lead to an increase in cost, a loss of productivity, and create tension between them. How does one, for instance, avoid cross-contamination of key findings during development, for instance if one team discovers significant omissions in the design requirements all teams work from?

Self-Checking Code

Although all of this sounds discouraging, there is in fact an effective method for using redundancy to improve system reliability, although we often do not think of it in quite that way. That method is to increase the use of assertions throughout a code base. As many software developers will tell you, these assertions are redundant and can safely be removed after testing. Do they help to improve reliability? The sobering news, at least for those developers, is that it can be shown that the number of assertions that are retained in the code after testing correlates strongly with post-release fault-density. More assertions mean fewer faults. It is as simple as that. This was first shown in a study done by Microsoft researchers, who studied post-release faults in the Office suite of tools and compared it with the assertion density of the failing code [4].

For the mission code that is developed at NASA/JPL we require that the average assertion density for each module is 2% or more. This means that 2% of the code performs self-checks at key steps in the computations performed, to make sure that integrity is maintained, even in anomalous execution scenarios. The use of assertions thus provides a form of software redundancy that can indeed make a system more reliable.

What About Testing

I have not mentioned testing in the discussion of reliability so far. Some say that since the cyclomatic complexity metric measures the number of paths in the control-flow graph of a function, this gives the number of tests that is required to test all those paths. Fewer required tests should then make it possible to test code more rigorously. This ignores, though, that most of the complexity in a program execution does not originate in the structure of the control-flow graph, but in the data that is processed. A single path in the control-flow graph can be executed in an astronomical number of different ways, which means that a single test of each path in the control-flow graph has little chance of revealing all the bugs that may hide on that path.

A function with a lower cyclomatic complexity does not necessarily require fewer tests to vet thoroughly. All code is written to satisfy some set of requirements and provide a desired functionality. We can spread that functionality across multiple modules and functions to create a large assembly of interconnected and mutually dependent small functions, but that will rarely be the right way to structure the code, and neither will it be able to simplify testing.

Requirements Testing

To put it most succinctly: tests should not be derived from the structure of a control-flow graph but from the requirements that prompted the creation of the code. To check if a requirement is met, the tester, or developer, should define a suite of tests that can evaluate the adequacy of the code to handle a range of cases, including expected cases, boundary cases, and even cases that outright violate the design assumptions. If after running that suite of tests for all requirements it turns out that parts of the code base were not reached, this indicates a flaw in either the requirements or in the code. The most common case is that the requirements are found to be incomplete and fail to cover things that the code must be able to handle. It can, however, also be that the code itself is redundant and parts of it serve no legitimate purpose to satisfy the requirements.

Code coverage metrics serve to check the adequacy of a test suite for a specific set of requirements. Adding test cases to an existing test suite that serve only to improve the coverage metric, without actually testing anything, is common practice, but yes, it cannot improve a system's reliability.

References

[1] T.J. McCabe (December 1976). "*A Complexity Measure,*" IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, 1976, pp. 308–320.

[2] Ariane 5 Flight 501 Failure, url: http://sunnyday.mit.edu/nasa-class/Ariane5-report.html

[3] J.C. Knight and N.G. Leveson, "*An Experimental Evaluation of the Assumption of Independence in Multi-version Programming,*" IEEE Trans. on Software Engineering,Vol. SE-12, No. 1, 1986, pp. 96-109.

[4] G. Kudrjavets, N. Nagappan, T. Ball, "*Assessing the Relationship between Software Assertions and Faults: An Empirical Investigation*," 17th Int. Symp. On Software Reliability Engineering, ISSRE, Nov. 2006, pp. 204-212.