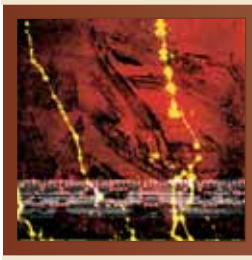
Conquering Complexity

Gerard J. Holzmann, NASA/JPL Laboratory for Reliable Software



In complex systems, combinations of minor software defects can lead to large system failures.

In his book *Normal Accidents: Living with High-Risk Technologies* (Princeton University Press, 1984), sociologist Charles Perrow discussed the causes of failure in highly complex systems, concluding that they were virtually inevitable. He argued convincingly that when seemingly unrelated parts of a larger system fail in some unforeseen combination, dependencies can become apparent that couldn't have been accounted for in the original design.

In safety-critical systems, the potential impact of each separate failure is normally studied in detail and remedied by adding backups. Failure combinations, though, are rarely studied exhaustively; there are just too many of them, and most have a very low probability of occurrence.

A compelling example in Perrow's book is a description of the events leading up to the partial meltdown of the nuclear reactor at Three Mile Island in 1979. The reactor was carefully designed with multiple backups that should have ruled out what happened. Yet, a small number of relatively minor failures in different parts of the system conspired to defeat those protections. A risk assessment of the probability of the scenario that unfolded would probably have concluded that it had a vanishingly small chance of occurring.

No software was involved in the Three Mile Island accident, but we can draw important lessons from it, especially in the construction of safety-critical software systems.

SOFTWARE DEFECTS

We don't have to look very far to find highly complex software systems that perform critically important functions: The phone system, fly-by-wire airplanes, and manned spacecraft are a few examples.

The amount of control software needed to, say, fly a space mission is rapidly approaching a million lines of code. If we go by industry statistics, a really good—albeit expensive—development process can reduce the number of flaws in such code to somewhere in the order of 0.1 residual defects per 1,000 lines. (A *residual* defect is one that shows up after the code has been fully tested and delivered. The larger total number of defects hiding in the code are often referred to as the *latent* defects.)

Thus, a system with one million lines of code should be expected to experience at least 100 defects while in operation. Not all these defects will show up at the same time of course, and not all of them will be equally damaging.

BIG FAILURES OFTEN START SMALL

Knowing that software components can fail doesn't tell us any more about a software system than knowing that a valve can get stuck in a mechanical system. As in any other system, this is only the starting point in the thought process that should lead to a reliable design.

We know that adding fault protection and redundancy can mitigate the effects of failures. However, adding backups and fault protection inevitably also increases a system's size and complexity. Designers might unwittingly add new failure modes by introducing unplanned couplings between otherwise independent system components.

Backups are typically designed to handle independent component failures. In software systems, they can help offset individual software defects that could be mission-ending if they strike. But what about the potential impact of combinations of what otherwise would be minor failures?

Given the magnitude of the number of possible failure combinations, there simply isn't enough time to address them all in a systematic software testing process. For example, just 10^2 residual defects might occur in close to 10^4 different combinations.

This, then, opens up the door to a "Perrow-class" accident in increasingly complex software systems. As before, the probability of any one specific combination of failures will be extremely low, but as experience shows, this is precisely what leads to major accidents. In fact, most software failures in space missions can be reasoned back to unanticipated combinations of otherwise benign events.

Minor software defects typically have no chance of causing outright disaster when they occur in isolation, and are therefore not always taken very seriously. Perrow's insight is that reducing the number of minor flaws can also reduce the chances for catastrophic failures in unpredictable scenarios.

CODING STANDARDS

We can reduce the number of minor defects in several ways. One way is to adopt stricter coding rules, like the ones I described in this column in "The Power of Ten: Rules for Developing Safety-Critical Code" (*Computer*, June 2006, pp. 95-97). Among these recommendations is the required use of strong static source code analyzers such as Grammatech's CodeSonar, or the analyzers from Coverity or Klocwork, on every build of the software from the start of the development process. Another is to always compile code with all warnings enabled, at the highest level available.

Safety-critical code should pass such checks with zero warnings, not even invalid ones. The rationale is that such code should be so clearly correct that it confuses neither humans nor tools.

DECOUPLING

Another way to reduce Perrow-class failures is to increase the amount of decoupling between software components and thereby separate independent system functionality as much as possible. Much of this is already standard in any good development process and meshes well with code-structuring mechanisms based on modularity and information hiding.

Many coding standards for safety-critical software development, such as the automotive industry's MISRA C guidelines, require programmers to limit the scope of declarations to the minimum possible, avoiding or even eliminating the use of global declarations in favor of static and constant declarations—using, for example, keywords such as *const* and *enum* in C.

One of the strongest types of decoupling is achieved by executing independent functions on physically distinct processors, providing only limited interfaces between them.

High-end cars already have numerous embedded processors that each perform carefully separated functions. Mercedes-Benz S-Class sedans, for example, reportedly contain about 50 embedded controllers, jointly executing more than 600,000 lines of code (K. Grimm, "Software Technology in an Automotive Company—Major Challenges," *Proc. 25th Int'l Conf. Software Engineering*, IEEE CS Press, 2003, pp. 498-503).

Many robotic spacecraft also use redundant computers, although these typically operate only in standby mode, executing the same software as the main computer. When the controlling computer crashes due to a hardware or software fault, the backup takes over.

Adopting stricter coding rules is one way to reduce the number of minor defects.

This strategy, of course, offers limited protection against software defects. In principle, decoupling could be increased by having each computer control a different part of the spacecraft, thereby limiting the opportunities for accidental cross-coupling in the event of a failure. When the hardware for one of the computers fails, the other(s) can still assume its work load, preserving protection against hardware failures.

CONTAINMENT

The separation of functionality across computers is also a good example of a defect containment strategy. If one computer fails, it affects only the functionality controlled by that computer.

Another example of defect containment is the use of memory protection to guarantee that multiple threads of execution in a computer can't corrupt each other's address spaces. Memory protection is standard in most consumer systems, but it's not always used in embedded software applications. For safety- and mission-critical systems, this should probably become a requirement rather than an option.

MARGIN

Another common defense against seemingly minor defects is to provide more margin than is necessary for system operation, even under stress. The extra margin can remove the opportunity for close calls, whereas temporary overload conditions can provide one of the key stepping stones leading to failure.

Margin doesn't just relate to performance, but also to the use of, for example, memory. It can include a prohibition against placing mission-critical data in adjacent memory locations. Because small programming defects can cause overrun errors, it's wise to place safety margins around and between all critical memory areas. These "protection zones" are normally filled with an unlikely pattern so that the effects of errant programs can be detected without causing harm.

The recent loss of contact with the Mars Global Surveyor spacecraft illustrates the need for these strategies. An update in one parameter via a memory patch missed its target by one word and ended up corrupting an unrelated but critical earth-pointing parameter that happened to be next to the first parameter in memory. Two minor problems now conspired to produce a major problem and ultimately led to the loss of the spacecraft. Had the key parameters been separated in memory, the spacecraft would still be operating today.

LAYERING

The use of multiple layers of functionality—less gloriously known as workarounds—is yet another way to provide redundancy in safety-critical code.

For example, most spacecraft have a nonvolatile memory system to store data; on newer missions, this is typically flash memory. Several independent means are used to store and access critical data on these devices. A spacecraft can often also function entirely without an external file system, in so-called "crippled mode," by building a shadow file system in main

memory. The switch to the backup system can be automated.

We can't predict where software defects might show up in flight, but we can often build in enough slack to maximize the chances that an immediately usable workaround is available no matter where the bugs hide.

MODEL-BASED ENGINEERING

Perhaps the main lesson that can be drawn from Perrow's discussion of complex systems is the need to scrutinize even minor software defects to reduce the chances that combinations of minor flaws can lead to larger failures.

The strategies mentioned so far deal mostly with defect prevention and defect containment. We've left the most commonly used strategy for last: defect *detection*.

Defect detection in software development is usually understood to be a best effort at rigorous testing just before deployment. But defects can be introduced in all phases of software design, not just in the final coding phase. Defect detection therefore shouldn't be limited to the end of the process, but practiced from the very beginning.

Large-scale software development typically starts with a requirements-capture phase, followed by design, coding, and testing. In a rigorous model-based engineering process, each phase is based on the construction of verifiable models that capture the main decisions.

Engineering models, then, aren't just stylized specifications that can be used to produce nice-looking graphs and charts. In a rigorous model-based engineering process, requirements are specified in a form that makes them suitable to use directly in the formal verification of design, for example with strong model-checking tools such as Spin (www.spinroot.com).

Properly formalized software requirements can also be used to generate comprehensive test suites automatically, and they can even be used to embed runtime monitors as watchdogs in the final code, thereby significantly strengthening the final test phase (www.runtime-verification.org).

Finally, test randomization techniques can help cover the less likely execution scenarios so often missed in even thorough software testing efforts.

Defect prevention, detection, and containment are all important—none of these techniques should be skipped in favor of any of the others. Even if, for example, developers exhaustively verify all the engineering models used to produce a safety-critical software system, it's still wise to test the resulting code as thoroughly as possible. There are simply too many steps in safety critical software development to take anything for granted.

Gerard J. Holzmann is a Fellow at NASA's Jet Propulsion Laboratory, California Institute of Technology, where he leads the Laboratory for Reliable Software. Contact him at gholzmann@acm.org.

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

Editor: Michael G. Hinchey, Loyola College in Maryland; mhinchey@loyola.edu