Fault Intolerance

Gerard J. Holzmann

Jet Propulsion Laboratory, California Institute of Technology

Some people say that the best way to make software more reliable is to treat it like a mathematical object. In this view, a program is a formal statement written in a language with a precisely defined meaning, just like in mathematics, meaning—at least in principle—that we can reason about any program and prove that it is, or is not, correct.

Even if you're somewhat skeptical about the "precisely defined meaning" part, you'll probably agree that a specific interpretation of any program is ultimately provided by the compiler that converts it into machine code and the hardware that executes that code. If we're lucky, the specific interpretation will even conform to the applicable language standard, but that's just icing on the cake.

Edsger Dijkstra expressed this more precisely a few decades ago.[1] In his view, program development should start with an unambiguous statement of what the program is meant to accomplish. The program and the proof that it meets its purpose are then developed step by step, side by side. An immediate objection to this approach is that it can require substantial mathematical skills—moreover, it is far from clear if this approach could scale to programs that are larger than a few hundred lines.

In this premiere installment of the Reliable Code department, I want to convince you of two things that might seem a little contradictory. The first is that the two objections I just described aren't nearly as serious today as they were a few decades ago, when this approach was first proposed. But, I'll also argue that following this approach cannot suffice to make software systems more reliable. Something else is needed, and that something else is in fact much simpler to put into practice.

Chasing Completeness

Let's consider first what it takes to prove formally that a program has a specific set of properties. How hard this is depends, of course, on both the program and the property. To show, for instance, that a sequentially executing, stand-alone program without any loops or jumps will eventually terminate isn't that hard, even though we know that proving program termination in general isn't possible. Similarly, showing that a program can't use an uninitialized variable in a calculation is often simple, even though in its full generality, this problem, too, is unsolvable.

Even though there can never be a single general procedure for proving programs correct, quite a few methods work surprisingly well on the programs we encounter in practice. The progress that has been made in the past few decades in the automation of program

verification techniques is nothing short of impressive. Methods that in the late 1970s looked like they would forever doom human provers to spend months of precious time establishing relatively simple properties for 10-line programs are today embedded into proof assistants, logic model checkers, static source code analyzers, and even compilers that render verdicts quickly, even for complex code. Today's logic model checkers, for instance, can find counterexamples to subtle correctness properties even for complex multithreaded code in seconds or minutes, without requiring the user to first build a formal model of that code.[2]

But—and here comes the downer—proving that a program satisfies a given set of properties isn't the same as proving that it's "correct" in an absolute sense, or that it will be reliable in any environment. What if, for instance, we prove the wrong properties, or if our set of properties is incomplete? If the programmer produces both the program and its specification, can we really assume that the latter is more accurate than the former?

We can try to get around this problem by saying that the proof's purpose is to reveal any discrepancies between what we think the program should do and what it actually does. But if our understanding of what our program should do is incomplete, we might still end up with an incorrect program, and that's a problem.

To give a very simple example of an incomplete program specification, consider the task of sorting numbers in numerically increasing order. A specification for an algorithm solving this problem could state that for any two subsequent numbers in the output, the second number should be larger than or equal to the first. To see why this specification is incomplete, note that it says nothing about how the output relates to the input. If this were the only requirement, we could write a "correct" program that always generates just two numbers: 1 and 2. Minimally, what's missing is a requirement that the output must be a valid permutation of the input: it must contain all numbers from the input, without additions or deletions.

We can feel better about the specification now, but it still might not be enough. Should the algorithm also work for floating-point numbers? If so, what should its accuracy be? The specification doesn't say what the largest or smallest number is that it should be able to handle, nor does it does say if there can be a maximum to input length. We can't take the answer to these questions for granted, and we probably don't want to leave it to chance.

Dealing with Error

The most difficult part of a formal specification, and the part that's most often incomplete, is to state clearly what should happen under various error conditions. What should the sorting algorithm do when the user includes non-numbers in the input or entries that have a non-numeric suffix? Should the algorithm skip such entries, or should it flag an error and stop? Note that if we allow the algorithm to skip over any part of the input, or to stop early, we're creating a conflict with the earlier requirement of no additions or deletions.

Especially for more complex systems it can be very hard to foresee all possible off-

nominal events that a system might encounter in real life and to define unambiguously how it should respond to them. Consider the development of an embedded software controller for a safety-critical system, such as a car, medical device, or spacecraft. The objective is here to make the system as a whole reliable. The control software is part of the system, but not the only part. It's generally wise to assume that, just like all the other parts of the system, the software could fail for unpredictable reasons, no matter how carefully it was constructed. We might show rigorously that the code satisfies certain key properties, but most likely that set of properties doesn't include something that reality later shows to be relevant. This applies especially to the system's response to various off-nominal conditions, which is of central importance in safety-critical system design.

Reality can be remarkably creative when it comes to showing where our imagination falls short. A great example of this is the scenario that led to the loss of the Mars Global Surveyor (MGS) spacecraft not that long ago. MGS was launched in November 1996 and went into orbit around Mars about 10 months later, in September 1997. The spacecraft faithfully performed its service to map the surface of the red planet and to act as a relay for communication with rovers that had been on the surface of Mars since January 2004. On Thursday, 2 November 2006, the MGS spacecraft received a routine command to adjust its solar panels for the start of winter, but the spacecraft failed to respond. After many attempts to reestablish contact over the next few weeks and months, the spacecraft was declared dead. What happened?

Spacecraft software is designed in such a way that if anything unexpected happens, the system automatically switches to a special safe mode that stops all routine operations, alerts operators on the ground with details about the mishap, and awaits instructions for recovery. It doesn't happen very often, but each spacecraft we launch hits this mode at least once in its lifetime, so there's quite a bit of experience in dealing with this type of event.

It turned out that the soft-stop value for the panel rotation had become corrupted, and as a result, the panels rotated until they hit a physical hard stop. Because this isn't supposed to happen, the spacecraft invoked its safe mode and suspended all further routine operations. So far, there was no real problem: the control software did precisely what it was designed to do. The corruption of the soft-stop value on the solar panels alone would have been easy to diagnose and correct.

In safe mode, a spacecraft has two priorities: make sure that the batteries have sufficient charge for continued operation, and establish communication with ground controllers on Earth, so that they can start diagnosing and fixing the problem. The MGS spacecraft couldn't satisfy both priorities at the same time, given the apparent problem with the solar arrays: the fault protection software had determined the anomaly to be related to the use of the solar panels, so it declared them off limits for further adjustment. This was indeed a reasonable, conservative decision, but it meant that to point the solar panels at the sun for battery charging, the whole spacecraft had to be turned. Again, this wasn't a problem, but it had one unforeseen side effect: when the spacecraft was rotated to point the solar panels toward the sun, the batteries pointed toward the sun as well. Consequently, the batteries

quickly heated up, which the fault protection software interpreted as a signal that they were overcharging. At this point, the spacecraft reoriented itself to point its antenna at Earth to send its safe-mode alert signal.

Then came surprise number two. The parameter for Earth pointing was located in memory right next to the parameter that held the value for the soft stop of the solar arrays, and the same event that corrupted the value of the soft stop had also corrupted the Earth-pointing parameter. This meant that the attempt to communicate with Earth didn't succeed either. At this point, the fault protection software accurately determined that the battery charge was still too low, forcing it to resume its attempt to recharge the batteries and leading it step by step into an endless loop of trying to recharge the batteries and communicate with Earth. Within a few hours, the batteries were dead, followed by the spacecraft. It's likely still orbiting Mars, although its location is currently unknown.

Fault Intolerance

What makes this example so interesting is that every part of the software worked as designed and could have been proven to meet design specifications. Yet, the specification itself was incomplete.

This is a good example of how difficult it can be to predict what can happen during a safety-critical system's lifetime. In almost all cases where a safety-critical system fails, the cause is an unexpected combination of relatively-low-probability events. Each separate event can be anticipated and a response prepared, but the net effect of a seemingly arbitrary combination of unlikely events is very difficult to predict. In this case, it was the unanticipated combination of data corruption in two unrelated, but collocated, parameters, and the coincidence that the solar panels were stuck in a position that was aligned with that of the spacecraft batteries.

Now that we know what happened, we can see that some of the fault protection priorities were actually in conflict. Charging the batteries is one of the highest priorities for a good reason: without power, all other operations of the spacecraft cease. Another given is that the control software should avoid overcharging the batteries to avoid damaging them, with the same end result of having no power at all. These two priorities would appear not to be in conflict, because one normally takes precedence when the battery charge is low and the other when it's high. But nature found a case where they could be in conflict without the fault protection system realizing it.

The conclusion from these examples isn't that all hope is lost and we'll never be able to design truly reliable systems—rather, it leads us to a different philosophy of software design that we can call *fault intolerance*, which begins with the realization that our insights into complete requirements are always limited and unavoidably contain uncertainty, especially about the type of faults that could occur.

Fault intolerance leads to a defensive posture in software design in which nothing is

taken for granted. The knowledge that small flaws can combine to produce large failures inspires an approach where every potential risk, no matter how small or apparently limited in its consequences, is addressed upon discovery. This is in many ways the "broken windows" policy of software design: the removal of small problems early is an effective way to prevent big problems later.[3]

This basic approach of fault intolerance applies also to compiler warnings, even the pedantic ones. Code should compile cleanly, with all warnings enabled—similarly, a good static source code analyzer should find nothing to complain about in our code, nor should it run into anything that confuses it into thinking that something could be wrong.

You could say that the goal of the mathematical approach to software design is to achieve certainty, but to be able to design truly reliable systems, we have to take a much broader view, where some degree of uncertainty always lingers despite our best attempts to eliminate it. The uncertainty exists, for instance, in our limited knowledge of the ultimate environment in which our systems must be able to perform their function and the types of failure that could occur.

Say that your software application contains a function that takes an integer value as a parameter and uses that parameter to index an array. A static analyzer might warn you if the parameter is used without a check that its value is within array bounds. Doing so is taking a small risk: you might decide that the warning is invalid after checking that none of the calling functions can cause the parameter to take an invalid value. It seems logical, but it could violate the policy of fault intolerance because it fails to acknowledge the basic uncertainty we have about how the system might execute at some point in the future, perhaps under anomalous conditions.

If you ignore the warning, you might have to check and recheck your assumption's validity every single time one of the calling functions is modified. Will you remember to do so? Will anyone remember the dependency that's now dormant, and likely undocumented, in the code? Will the person who comes after you know to continue to check that assumption? What if one of the calling functions misbehaves because of some unpredictable fault somewhere else in the system? What if your function is reused years later in a different context, where the undocumented assumption isn't satisfied? Not providing the check amounts to removing a layer of protection from your code that could allow a minor fault to propagate and contribute to a much larger problem.

For well-written code, we should expect to see an assertion density of at least two percent, meaning that two out of every 100 executable statements are assertions. Typically, the assertions are placed at the entrance and exit points of nontrivial functions, to check that the assumptions the function would otherwise silently make about its input arguments are valid and that the result it returns to the caller can pass a basic sanity test. Technically, these assertions are just as redundant as the seat belts in your car.

The use of redundancy to achieve greater reliability is of course well understood in other fields. It forms the basis for information theory: to transmit information reliably over unreliable channels, you add redundant information. Similarly, in hardware design, you can use redundancy to protect yourself against the possibility of seemingly random component failures.

The Cassini spacecraft currently orbiting Saturn, for instance, has two main engines for major course adjustments, although it really needs just one to perform the mission. It also has two main flight computers, multiple transmitters and receivers, two sets of thrusters, and an extra reaction wheel that can help orient itself accurately in space. Almost all the backup equipment, except the second main engine and the backup flight computer, has been called into duty in the 17 years since the spacecraft was launched. Similarly, the two, still active, Voyager spacecraft launched 37 years ago each have no less than three dual-redundant computer systems. The principle is the same in each case: the redundant components might take up more precious space and mass, but they successfully protect the system from the unpredictable.

The development of reliable software requires not just a thorough analysis of all aspects of the code that are amenable to it, but also a basic approach of fault intolerance and the use of redundancy in the form of seemingly superfluous checks throughout the code. After all, the unpredictable is sometimes quite predictable.

References

1. E.W. Dijkstra, "Programming as a Discipline of Mathematical Nature," EWD 361, 23 May 1973; https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD361.html
2. G.J. Holzmann, "Mars Code," *Comm. ACM*, vol. 57, no. 2, 2014, pp. 64–73.
3. J.Q. Wilson, and G.L. Kelling, "Broken Windows: The Police and Neighborhood Safety," *The Atlantic*, Mar. 1982, http://www.manhattan-institute.org/pdf/_atlantic_monthly-broken_windows.pdf .

Acknowledgement

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