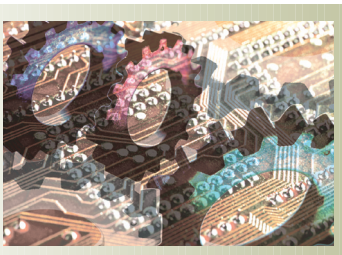
If You Didn't Test It, It Doesn't Work

Bob Colwell

Science is supposed to learn more from its failures than its successes. Schools actively teach trial and error: If at first you don't succeed, try, try again. On the other hand, we're told to look before we leap but that he who hesitates is lost—so maybe we're all doomed no matter what.

This notion that trying and failing is normal, and natural, can lead you astray in your professional life, however. In practice, it's a lot harder to get a conference paper accepted that is of the form, "We wanted to help solve important problem X, so we tried Y, but instead of the hoped-for result Z, we got yucky result Q." And when I say "a lot harder," I mean "there's no way." This is true in science as well as engineering, and it's responsible for the overall impression you get when reading conference proceedings: Everything other people try works perfectly. You're the only one who goes down blind alleys and suffers repeated failures.

There's a good reason why conferences and journals are biased this way. Sometimes there's a very fine line between good ideas that didn't work out and just plain stupid ideas. And since there are a thousand wrong ways to do something for every way that works, a predisposition for papers with positive results is a quick, generally effective filter that saves conference committees a lot of time. (They need to save that time so they can use it later, debating how to break it to a fellow conference committee member that they want to reject his paper. As Dave Barry would say, I'm not making this up.) Of course, prospective authors know this, and they would be in danger of biasing their results toward the most positive light if it weren't for their dedication to the scientific ideal. (Are you buying any of that? Me neither.)



Most things you design can't be tested to saturation.

Scientists get to run their experiments multiple times, and they can gradually reduce or eliminate sources of systematic errors that might otherwise bias their results. Engineers tend to get only one chance to do something: one bridge to build, one CPU to design, one software development project to crank out late, slow, and.... Whoops. I got carried away there.

ANTICIPATION

Because engineers generally can't test their creations to the point of saturation, they must make do with a lot of substitutions: anticipation of all possible failure modes; a comprehensive set of requirements; dedicated validation and verification teams; designing with a built-in safety margin; formal verification where possible; and testing, testing, testing. If you didn't test it, it doesn't work.

In some cases, computers have become fast enough to permit testing every combination of bit patterns. If you're designing a single-precision floating-point multiplier, you *know* the unit is correct because you tested every possible way it could be used. Now all you have to do is worry about resets, protocol errors into and out of the unit, electrical issues, noise, thermals, crosstalk, and why your boss didn't smile at you when she said good morning yesterday.

Testing to saturation

But many, perhaps most, things you design can't be tested to saturation. Double-precision floating-point numbers have so many bit patterns that it's hard to see when, if ever, such units could be tested to saturation. Two to the 64th power is an extremely large number, and there are usually two operands involved. And that's for a functional unit with a fairly regular design and state sequence. Saturation testing of anything beyond trivial software is often even more out of the question.

So it behooves us to try to anticipate how our designs will be used, certainly under nominal conditions, but also under non-nominal conditions, which usually place the system under higher stress. Programmers have a range of techniques at their disposal, such as defensive programming (checking input values before blindly using them), testing assertions (automatically checking for conditions the programmer thought were impossible), and always checking function return values. Engineers design in fail-safe or fail-soft mechanisms because they know that the unexpected happens all the time. If you've ever driven a car that had the check-engine-light illuminated, you were in one of those modes.

Real-world uses

One of the more difficult tasks in engineering is trying to imagine all of the conditions under which your creation will be used in the real world—or, if you're an aerospace engineer, in the real universe. There are user interfaces to consider: Automobile manufacturers long ago resigned themselves to designing to the lowest common denominator of the human population in terms of intelligence applied to the operation of a motor vehicle. But idiot-proofing is difficult precisely because idiots are so clever.

There are also legal concerns—juries that award damages to people who think driving with a hot cup of coffee between their legs is a reasonable proposition may well fail to grasp the engineering tradeoffs inherent in any design. How do we protect idiots from themselves?

DESIGNING WITH MARGIN

As a conscientious designer, you try to anticipate all the ways your design may be used in the future. Since you do outstanding work, you expect that you or someone else will reuse your code or your silicon in the future. So you try to guess what that future designer will want and strive to provide it.

You've also been around this particular block a few times before, so you build in some flexibility because you know that project management will change its mind about certain major project goals as you go along. And since debugging was such a bear last time, you allow for lots of debug hooks and performance counters. If you keep traveling down this road, you're going to make yourself crazy. Yes, more than you are now. You need to compromise.

The art of compromise

Engineering is the art of compromise: now versus later, concise versus generalized, schedule versus thoroughness. And all around you, the other designers are falling behind and asking for help—*your* help.

This is one area of engineering in which you can't simply overwhelm the problem with force. You want an elegant, intelligent, balanced creation that meets all of the specs and also the intent of the product, stated or otherwise—not a sandbagged, bloated, turkey of a design. Yes, it will take compromises, but the magic is in getting those compromises right. Engineers know elegant designs when they see them. Aspire to produce them.

The tragedy of the commons occurs when designers decide to borrow more of the chip area than they were allocated.

Tragedy of the commons

Always keep in mind the "tragedy of the commons." The idea is that there are shared resources, initially allocated in the hope of having enough for all, but overall management of those resources is essentially a distributed local function.

If you're a silicon chip designer, your job is to implement your piece of the design in the area you were allocated, and to do so within the power, schedule, and timing constraints. You know that those constraints are somewhat fungible: If you had more schedule time, you could compact your real-estate footprint. With more thermal power headroom, you could meet your clocking constraint.

The tragedy of the commons occurs when, faced with the same problems you face, your fellow designers decide to borrow more of the chip area than they were allocated. It appears to each of them that they're using only a tiny fraction of the unallocated chip space, while substantially boosting the die area available to their particular function. And so they are. But if everyone does that, the chip's overall die size will exceed the target, with no margin left to do anything about it. Think locally *and* globally.

VALIDATION AND VERIFICATION

Did I mention this: If you didn't test it, it doesn't work. In *DragonFly: NASA and the Crisis Aboard MIR* (HarperCollins, New York, 1998), Bryan Burrough gives a riveting account of several near catastrophes on the Russian space station.

In one instance, an oxygen generator catches fire, and a blowtorch-like flame begins to bloom. Jerry Linenger, the sole American astronaut, grabs an oxygen mask, but it doesn't work. He grabs another one that does work. The station commander leads Linenger to the station module where the fire extinguishers are kept. Linenger grabs one "but is startled to find it is secured to the wall." Both men pull at it, but the wall wins. They try another extinguisher, with the same outcome.

Later analysis revealed that the transportation straps for the fire extinguishers were still installed—in the intervening 19 months of service, no one had thought to wield the wrench required to remove them. In perfect hindsight, this also suggests that whatever fire drills had been performed in those months weren't realistic enough or the astronauts would have found the problem earlier.

An important distinction

NASA draws a useful distinction between validation and verification. Verification is the act of testing a design against its specifications. Validation tests the system against its operational goals.

An example of why the distinction is important comes from *Endeavour*'s maiden flight, when it tried to rendezvous with the Intelsat satellite. As it turned out, the "state-space" part of the shuttle's programming had been done with double-precision floating-point arithmetic, and the "boundary checking" part was done with single-precision arithmetic. The shuttle's attempted rendezvous didn't converge

to a solution due to this mismatch; only a live patch from the ground saved the mission.

If the specs called for double precision, verification of the state-space code would never find this potential problem: different specs for different code. Validation, on the other hand, could reasonably have been expected to detect this mismatch before the astronauts encountered it hundreds of miles above the earth.

I'm very fond of validation. Verification is absolutely necessary, and it's as important (and difficult) a task as design or architecture. In my experience, however, the job of making sure a design correctly implements its specs isn't likely to be forgotten or overlooked. Verification may not always be done very well, but it probably won't be skipped entirely. Validation, on the other hand, is often misunderstood, and management sometimes doesn't remember why it's so necessary.

Seeing the big picture

In my experience, after months or years of intensive design and development, the designers are tired, they're being pressured to hit their production schedule, and they just want to finish the project. Some part of them really doesn't want to hear that there might be anything wrong with their baby. The architects often have moved into the initial phase of another design, and they may not even be available, much less actively engaged in final testing. So the validation folks are the only ones left who can try to see the Big Picture.

Especially on long design projects, fundamental changes may have occurred in the overall product landscape that haven't been fully incorporated into the specs: Competition has arisen, a lawsuit may have been resolved in an unfavorable direction, last-minute design changes were made that won't have benefited from the years of testing the design otherwise enjoyed.

Here's an example of what I mean. While testing the *Madden NFL 99* PC game, a validator wondered what would happen if he tried to catch a field goal. For readers who don't follow American football, a field goal is an attempt by the offensive team to kick the ball between the goal posts at the end of the field. During a field goal attempt, the offense just tries to keep the defense from blocking the kick. There would be no point in sending any offensive players downfield, so nobody ever does.

Having a validation mindset that allows an independent thinker to step outside a design project's orthodoxy is absolutely vital.

The validator was unswayed by the spec that there's no point in trying to catch the ball after it has been kicked; in true validation hero fashion, he simply noticed that it didn't seem to be impossible. After trying for a couple of hours, he succeeded in doing it. Since the game's designers had never conceived of anyone trying to do such a goofy thing, the game's specs didn't include a requirement for how to handle it. As the validator suspected, the game didn't handle the situation very well—it locked up.

Stepping outside the box

Having a validation mindset that allows an independent thinker to step outside a design project's orthodoxy is absolutely vital. Such people often are the last line of defense between something that's overlooked in design and a user who isn't happy with the product—or worse.

Although it's not a common experience in most people's daily lives, you know what this mindset feels like. An example borrowed from Douglas Hofstadter's *Fluid Concepts and Creative Analogies* (Basic Books, New York, 1995) simulates it. Hofstadter believes that analogies are among the highest levels of intelligent thought, and he uses numerical sequences as the clearest analogy-generators. You remember these—What comes next in this sequence: 5, 10, 15? Yep, 20 would be my bet too.

So what comes next in the following sequence: 0, 1, 2? It has to be 3, doesn't it? But wait, why would I give you a trivial sequence and claim it's going to stimulate your brain? What else could it be, if not 3?

Even when I give you the answer, you probably *still* won't know where this is coming from. The answer is 0, 1, 2, 720!. That's 720 factorial: 0, 1, 2, 720! = 0, 1, 2, 6!! = 0, 1!, 2!!, 3!!!.

The important thing about this example is not the actual math involved per se. It's that when you first saw the sequence 0, 1, 2, your immediate instinct probably was to answer "3" and move on. But some other part of your brain, your Validation Reflex, became instantly suspicious and said, "Not so fast. Something's not quite right here."

That's the voice you must learn to hear to do truly outstanding validation. That's the instinct that allows you to take a step back from the implicit assumptions that others around you are making so you can go in a different direction where you may see something everyone else is missing.

Yeah, I know—I cheated a little with that math sequence. It doesn't seem quite fair to throw in factorials when the original sequence didn't have them. You could complain to Hofstadter. But you should read his book first.

In the meantime, what's next in this sequence: If you didn't test it, …? ■

***Bob Colwell** was Intel's chief IA32 architect through the Pentium II, III, and 4 microprocessors. He is now an independent consultant. Contact him at bob.colwell@attbi.com.*