The Atlantic Follow
Politics, culture, business, science, technology, health, education, global affairs, and more.
Sep 27, 2017 · 37 min read

The Coming Software Apocalypse

A small group of programmers wants to change how we code—before catastrophe strikes.

By James Somers

Illustration: Lynn Scurfield

There were six hours during the night of April 10, 2014, when the entire population of Washington State had no 911 service. People who called for help got a busy signal. One Seattle woman dialed 911 at

least 37 times while a stranger was trying to break into her house. When he finally crawled into her living room through a window, she picked up a kitchen knife. The man fled.

The 911 outage, at the time the largest ever reported, was traced to software running on a server in Englewood, Colorado. Operated by a systems provider named Intrado, the server kept a running counter of how many calls it had routed to 911 dispatchers around the country. Intrado programmers had set a threshold for how high the counter could go. They picked a number in the millions.

Shortly before midnight on April 10, the counter exceeded that number, resulting in chaos. Because the counter was used to generating a unique identifier for each call, new calls were rejected. And because the programmers hadn't anticipated the problem, they hadn't created alarms to call attention to it. Nobody knew what was happening. Dispatch centers in Washington, California, Florida, the Carolinas, and Minnesota, serving 11 million Americans, struggled to make sense of reports that callers were getting busy signals. It took until morning to realize that Intrado's software in Englewood was responsible, and that the fix was to change a single number.

Not long ago, emergency calls were handled locally. Outages were small and easily diagnosed and fixed. The rise of cellphones and the promise of new capabilities—what if you could text 911? or send videos to the dispatcher?—drove the development of a more complex system that relied on the internet. For the first time, there could be such a thing as a national 911 outage. There have now been four in as many years.

It's been said that software is "eating the world." More and more, critical systems that were once controlled mechanically, or by people, are coming to depend on code. This was perhaps never clearer than in the summer of 2015, when on a single day, United Airlines grounded its fleet because of a problem with its departure-management system; trading was suspended on the New York Stock Exchange after an upgrade; the front page of *The Wall Street Journal*'s website crashed; and Seattle's 911 system went down again, this time because a different router failed. The simultaneous failure of so many software systems smelled at first of a coordinated cyberattack. Almost more frightening was the realization, late in the day, that it was just a coincidence.

"When we had electromechanical systems, we used to be able to test them *exhaustively*," says Nancy Leveson, a professor of aeronautics and astronautics at the Massachusetts Institute of Technology who has been studying software safety for 35 years. She became known for her report on the Therac-25, a radiation-therapy machine that killed six patients because of a software error. "We used to be able to think through all the things it could do, all the states it could get into." The electromechanical interlockings that controlled train movements at railroad crossings, for instance, only had so many configurations; a few sheets of paper could describe the whole system, and you could run physical trains against each configuration to see how it would behave. Once you'd built and tested it, you knew exactly what you were dealing with.

Software is different. Just by editing the text in a file somewhere, the same hunk of silicon can become an autopilot or an inventory-control system. This flexibility is software's miracle, and its curse. Because it can be changed cheaply, software is constantly changed; and because it's unmoored from anything physical—a program that is a thousand times more complex than another takes up the same actual space—it tends to grow without bound. "The problem," Leveson wrote in a book, "is that we are attempting to build systems that are beyond our ability to intellectually manage."

Our standard framework for thinking about engineering failures—reflected, for instance, in regulations for medical devices—was developed shortly after World War II, before the advent of software, for electromechanical systems. The idea was that you make something reliable by making its parts reliable (say, you build your engine to withstand 40,000 takeoff-and-landing cycles) and by planning for the breakdown of those parts (you have two engines). But software doesn't *break*. Intrado's faulty threshold is not like the faulty rivet that leads to the crash of an airliner. The software did exactly what it was told to do. In fact it did it perfectly. The reason it failed is that it was told to do the wrong thing. Software failures are failures of understanding, and of imagination. Intrado actually had a backup router, which, had it been switched to automatically, would have restored 911 service almost immediately. But, as described in a report to the FCC, "the situation occurred at a point in the application logic that was not designed to perform any automated corrective actions."

This is the trouble with making things out of code, as opposed to something physical. “The complexity,” as Leveson puts it, “is invisible to the eye.”

The attempts now underway to change how we make software all seem to start with the same premise: Code is too hard to think about. Before trying to understand the attempts themselves, then, it’s worth understanding why this might be: what it is about code that makes it so foreign to the mind, and so unlike anything that came before it.

Technological progress used to change the way the world looked—you could watch the roads getting paved; you could see the skylines rise. Today you can hardly tell when something is remade, because so often it is remade by code. When you press your foot down on your car’s accelerator, for instance, you’re no longer controlling anything directly; there’s no mechanical link from the pedal to the throttle. Instead, you’re issuing a command to a piece of software that decides how much air to give the engine. The car is a computer you can sit inside of. The steering wheel and pedals might as well be keyboard keys.

Like everything else, the car has been computerized to enable new features. When a program is in charge of the throttle and brakes, it can slow you down when you’re too close to another car, or precisely control the fuel injection to help you save on gas. When it controls the steering, it can keep you in your lane as you start to drift, or guide you into a parking space. You couldn’t build these features without code. If you tried, a car might weigh 40,000 pounds, an immovable mass of clockwork.

Software has enabled us to make the most intricate machines that have ever existed. And yet we have hardly noticed, because all of that complexity is packed into tiny silicon chips as millions and millions of lines of code. But just because we can’t see the complexity doesn’t mean that it has gone away.

The programmer, the renowned Dutch computer scientist Edsger Dijkstra wrote in 1988, “has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to

face before." Dijkstra meant this as a warning. As programmers eagerly poured software into critical systems, they became, more and more, the linchpins of the built world—and Dijkstra thought they had perhaps overestimated themselves.

What made programming so difficult was that it required you to think like a computer. The strangeness of it was in some sense more vivid in the early days of computing, when code took the form of literal ones and zeros. Anyone looking over a programmer's shoulder as they pored over line after line like "100001010011" and "000010011110" would have seen just how alienated the programmer was from the actual problems they were trying to solve; it would have been impossible to tell whether they were trying to calculate artillery trajectories or simulate a game of tic-tac-toe. The introduction of programming languages like Fortran and C, which resemble English, and tools, known as "integrated development environments," or IDEs, that help correct simple mistakes (like Microsoft Word's grammar checker but for code), obscured, though did little to actually change, this basic alienation—the fact that the programmer didn't work on a problem directly, but rather spent their days writing out instructions for a machine.

"The problem is that software engineers don't understand the *problem* they're trying to solve, and don't care to," says Leveson, the MIT software-safety expert. The reason is that they're too wrapped up in getting their code to work. "Software engineers like to provide all kinds of tools and stuff for coding errors," she says, referring to IDEs. "The serious problems that have happened with software have to do with requirements, not coding errors." When you're writing code that controls a car's throttle, for instance, what's important is the rules about when and how and by how much to open it. But these systems have become so complicated that hardly anyone can keep them straight in their head. "There's 100 million lines of code in cars now," Leveson says. "You just cannot anticipate all these things."

In September 2007, Jean Bookout was driving on the highway with her best friend in a Toyota Camry when the accelerator seemed to get stuck. When she took her foot off the pedal, the car didn't slow down. She tried the brakes but they seemed to have lost their power. As she swerved toward an off-ramp going 50 miles per hour, she pulled the emergency brake. The car left a skid mark 150 feet long before running

into an embankment by the side of the road. The passenger was killed. Bookout woke up in a hospital a month later.

The incident was one of many in a nearly decade-long investigation into claims of so-called unintended acceleration in Toyota cars. Toyota blamed the incidents on poorly designed floor mats, "sticky" pedals, and driver error, but outsiders suspected that faulty software might be responsible. The National Highway Traffic Safety Administration enlisted software experts from NASA to perform an intensive review of Toyota's code. After nearly 10 months, the NASA team hadn't found evidence that software was the cause—but said they couldn't prove it wasn't.

It was during litigation of the Bookout accident that someone finally found a convincing connection. Michael Barr, an expert witness for the plaintiff, had a team of software experts spend 18 months with the Toyota code, picking up where NASA left off. Barr described what they found as "spaghetti code," programmer lingo for software that has become a tangled mess. Code turns to spaghetti when it accretes over many years, with feature after feature piling on top of, and being woven around, what's already there; eventually the code becomes impossible to follow, let alone to test exhaustively for flaws.

Using the same model as the Camry involved in the accident, Barr's team demonstrated that there were actually more than 10 million ways for the onboard computer to cause unintended acceleration. They showed that as little as a single bit flip—a one in the computer's memory becoming a zero or vice versa—could make a car run out of control. The fail-safe code that Toyota had put in place wasn't enough to stop it. "You have software watching the software," Barr testified. "If the software malfunctions and the same program or same app that is crashed is supposed to save the day, it can't save the day because it is not working."

Barr's testimony made the case for the plaintiff, resulting in $3 million in damages for Bookout and her friend's family. According to *The New York Times*, it was the first of many similar cases against Toyota to bring to trial problems with the electronic throttle-control system, and the first time Toyota was found responsible by a jury for an accident involving unintended acceleration. The parties decided to settle the case before punitive damages could be awarded. In all, Toyota recalled

more than 9 million cars, and paid nearly $3 billion in settlements and fines related to unintended acceleration.

There will be more bad days for software. It's important that we get better at making it, because if we don't, and as software becomes more sophisticated and connected—as it takes control of more critical functions—those days could get worse.

The problem is that programmers are having a hard time keeping up with their own creations. Since the 1980s, the way programmers work and the tools they use have changed remarkably little. There is a small but growing chorus that worries the status quo is unsustainable. "Even very good programmers are struggling to make sense of the systems that they are working with," says Chris Granger, a software developer who worked as a lead at Microsoft on Visual Studio, an IDE that costs $1,199 a year and is used by nearly a third of all professional programmers. He told me that while he was at Microsoft, he arranged an end-to-end study of Visual Studio, the only one that had ever been done. For a month and a half, he watched behind a one-way mirror as people wrote code. "How do they use tools? How do they think?" he said. "How do they *sit* at the computer, do they touch the mouse, do they not touch the mouse? All these things that we have dogma around that we haven't actually tested empirically."

The findings surprised him. "Visual Studio is one of the single largest pieces of software in the world," he said. "It's over 55 million lines of code. And one of the things that I found out in this study is more than 98 percent of it is completely irrelevant. All this work had been put into this thing, but it missed the fundamental problems that people faced. And the biggest one that I took away from it was that basically *people are playing computer inside their head*." Programmers were like chess players trying to play with a blindfold on—so much of their mental energy is spent just trying to picture where the pieces are that there's hardly any left over to think about the game itself.

John Resig had been noticing the same thing among his students. Resig is a celebrated programmer of JavaScript—software he wrote powers over half of all websites—and a tech lead at the online-education site Khan Academy. In early 2012, he had been struggling with the site's

computer-science curriculum. Why was it so hard to learn to program? The essential problem seemed to be that code was so abstract. Writing software was not like making a bridge out of popsicle sticks, where you could see the sticks and touch the glue. To "make" a program, you typed words. When you wanted to change the behavior of the program, be it a game, or a website, or a simulation of physics, what you actually changed was text. So the students who did well—in fact the only ones who survived at all—were those who could step through that text one instruction at a time in their head, thinking the way a computer would, trying to keep track of every intermediate calculation. Resig, like Granger, started to wonder if it had to be that way. Computers had doubled in power every 18 months for the last 40 years. Why hadn't programming changed?

The fact that the two of them were thinking about the same problem in the same terms, at the same time, was not a coincidence. They had both just seen the same remarkable talk, given to a group of software-engineering students in a Montreal hotel by a computer researcher named Bret Victor. The talk, which went viral when it was posted online in February 2012, seemed to be making two bold claims. The first was that the way we make software is fundamentally broken. The second was that Victor knew how to fix it.

Bret victor does not like to write code. "It sounds weird," he says. "When I want to make a thing, especially when I want to create something in software, there's this initial layer of disgust that I have to push through, where I'm not manipulating the thing that I want to make, I'm writing a bunch of text into a text editor."

"There's a pretty strong conviction that that's the wrong way of doing things."

Victor has the mien of David Foster Wallace, with a lightning intelligence that lingers beneath a patina of aw-shucks shyness. He is 40 years old, with traces of gray and a thin, undeliberate beard. His voice is gentle, mournful almost, but he wants to share what's in his head, and when he gets on a roll he'll seem to skip syllables, as though outrunning his own vocal machinery.

Though he runs a lab that studies the future of computing, he seems less interested in technology per se than in the minds of the people who use it. Like any good toolmaker, he has a way of looking at the world that is equal parts technical and humane. He graduated top of his class at the California Institute of Technology for electrical engineering, and then went on, after grad school at the University of California, Berkeley, to work at a company that develops music synthesizers. It was a problem perfectly matched to his dual personality: He could spend as much time thinking about the way a performer makes music with a keyboard—the way it becomes an extension of their hands—as he could thinking about the mathematics of digital signal processing.

By the time he gave the talk that made his name, the one that Resig and Granger saw in early 2012, Victor had finally landed upon the principle that seemed to thread through all of his work. (He actually called the talk "Inventing on Principle.") The principle was this: "Creators need an immediate connection to what they're creating." The problem with programming was that it violated the principle. That's why software systems were so hard to think about, and so rife with bugs: The programmer, staring at a page of text, was abstracted from whatever it was they were actually making.

"Our current conception of what a computer program is," he said, is "derived straight from Fortran and ALGOL in the late '50s. Those languages were designed for punch cards." That code now takes the form of letters on a screen in a language like C or Java (derivatives of Fortran and ALGOL), instead of a stack of cards with holes in it, doesn't make it any less dead, any less indirect.

There is an analogy to word processing. It used to be that all you could see in a program for writing documents was the text itself, and to change the layout or font or margins, you had to write special "control codes," or commands that would tell the computer that, for instance, "this part of the text should be in italics." The trouble was that you couldn't see the effect of those codes until you printed the document. It was hard to predict what you were going to get. You had to imagine how the codes were going to be interpreted by the computer—that is, you had to play computer in your head.

Then WYSIWYG (pronounced "wizzywig") came along. It stood for "What You See Is What You Get." When you marked a passage as being

in italics, the letters tilted right there on the screen. If you wanted to change the margin, you could drag a ruler at the top of the screen—and *see* the effect of that change. The document thereby came to feel like something real, something you could poke and prod at. Just by looking you could tell if you'd done something wrong. Control of a sophisticated system—the document's layout and formatting engine—was made accessible to anyone who could click around on a page.

Victor's point was that programming itself should be like that. For him, the idea that people were doing important work, like designing adaptive cruise-control systems or trying to understand cancer, by staring at a text editor, was appalling. And it was the proper job of programmers to ensure that someday they wouldn't have to.

There was precedent enough to suggest that this wasn't a crazy idea. Photoshop, for instance, puts powerful image-processing algorithms in the hands of people who might not even know what an algorithm is. It's a complicated piece of software, but complicated in the way a good synth is complicated, with knobs and buttons and sliders that the user learns to play like an instrument. Squarespace, a company that is perhaps best known for advertising aggressively on podcasts, makes a tool that lets users build websites by pointing and clicking, instead of by writing code in HTML and CSS. It is powerful enough to do work that once would have been done by a professional web designer.

But those were just a handful of examples. The overwhelming reality was that when someone wanted to do something interesting with a computer, they had to write code. Victor, who is something of an idealist, saw this not so much as an opportunity but as a moral failing of programmers at large. His talk was a call to arms.

At the heart of it was a series of demos that tried to show just how primitive the available tools were for various problems—circuit design, computer animation, debugging algorithms—and what better ones might look like. His demos were virtuosic. The one that captured everyone's imagination was, ironically enough, the one that on its face was the most trivial. It showed a split screen with a game that looked like *Mario* on one side and the code that controlled it on the other. As Victor changed the code, things in the game world changed: He decreased one number, the strength of gravity, and the Mario character

floated; he increased another, the player's speed, and Mario raced across the screen.

Suppose you wanted to design a level where Mario, jumping and bouncing off of a turtle, would *just* make it into a small passageway. Game programmers were used to solving this kind of problem in two stages: First, you stared at your code—the code controlling how high Mario jumped, how fast he ran, how bouncy the turtle's back was—and made some changes to it in your text editor, using your imagination to predict what effect they'd have. Then, you'd replay the game to see what actually happened.

CUSEC / Vimeo

Victor wanted something more immediate. "If you have a process in time," he said, referring to Mario's path through the level, "and you want to see changes immediately, you have to map time to space." He hit a button that showed not just where Mario was right now, but where he would be at every moment in the future: a curve of shadow Marios stretching off into the far distance. What's more, this projected path was reactive: When Victor changed the game's parameters, now controlled by a quick drag of the mouse, the path's shape changed. It was like having a god's-eye view of the game. The whole problem had been reduced to playing with different parameters, as if adjusting levels on a stereo receiver, until you got Mario to thread the needle. With the right interface, it was almost as if you weren't working with code at all; you were manipulating the game's behavior directly.

When the audience first saw this in action, they literally gasped. They knew they weren't looking at a kid's game, but rather the future of their industry. *Most*software involved behavior that unfolded, in complex ways, over time, and Victor had shown that if you were imaginative enough, you could develop ways to see that behavior and change it, as if playing with it in your hands. One programmer who saw the talk wrote later: "Suddenly all of my tools feel obsolete."

When John Resig saw the "Inventing on Principle" talk, he scrapped his plans for the Khan Academy programming curriculum. He wanted the site's programming exercises to work just like Victor's

demos. On the left-hand side you'd have the code, and on the right, the running program: a picture or game or simulation. If you changed the code, it'd instantly change the picture. "In an environment that is truly responsive," Resig wrote about the approach, "you can completely change the model of how a student learns … [They] can now immediately see the result and intuit how underlying systems inherently work without ever following an explicit explanation." Khan Academy has become perhaps the largest computer-programming class in the world, with a million students, on average, actively using the program each month.

Chris Granger, who had worked at Microsoft on Visual Studio, was likewise inspired. Within days of seeing a video of Victor's talk, in January of 2012, he built a prototype of a new programming environment. Its key capability was that it would give you instant feedback on your program's behavior. You'd see what your system was doing right next to the code that controlled it. It was like taking off a blindfold. Granger called the project "Light Table."

In April of 2012, he sought funding for Light Table on Kickstarter. In programming circles, it was a sensation. Within a month, the project raised more than $200,000. The ideas spread. The notion of *liveness*, of being able to see data flowing through your program instantly, made its way into flagship programming tools offered by Google and Apple. The default language for making new iPhone and Mac apps, called Swift, was developed by Apple from the ground up to support an environment, called Playgrounds, that was directly inspired by Light Table.

But seeing the impact that his talk ended up having, Bret Victor was disillusioned. "A lot of those things seemed like misinterpretations of what I was saying," he said later. He knew something was wrong when people began to invite him to conferences to talk about programming tools. "Everyone thought I was interested in programming environments," he said. Really he was interested in how people see and understand systems—as he puts it, in the "visual representation of dynamic behavior." Although code had increasingly become the tool of choice for *creating* dynamic behavior, it remained one of the worst tools for understanding it. The point of "Inventing on Principle" was to show that you could mitigate that problem by making the connection between a system's behavior and its code immediate.

In a pair of later talks, “Stop Drawing Dead Fish” and “Drawing Dynamic Visualizations,” Victor went one further. He demoed two programs he’d built—the first for animators, the second for scientists trying to visualize their data—each of which took a process that used to involve writing lots of custom code and reduced it to playing around in a WYSIWYG interface. Victor suggested that the same trick could be pulled for nearly every problem where code was being written today. “I’m not sure that programming has to exist at all,” he told me. “Or at least software developers.” In his mind, a software developer’s proper role was to create tools that removed the need for software developers. Only then would people with the most urgent computational problems be able to grasp those problems directly, without the intermediate muck of code.

Of course, to do that, you’d have to get programmers themselves on board. In a recent essay, Victor implored professional software developers to stop pouring their talent into tools for building apps like Snapchat and Uber. “The inconveniences of daily life are not the significant problems,” he wrote. Instead, they should focus on scientists and engineers—as he put it to me, “these people that are doing work that actually matters, and critically matters, and using really, really bad tools.” Exciting work of this sort, in particular a class of tools for “model-based design,” was already underway, he wrote, and had been for years, but most programmers knew nothing about it.

"If you really look hard at all the industrial goods that you’ve got out there, that you’re using, that companies are using, the only non-industrial stuff that you have inside this is the code.” Eric Bantégnie is the founder of Esterel Technologies (now owned by ANSYS), a French company that makes tools for building safety-critical software. Like Victor, Bantégnie doesn’t think engineers should develop large systems by typing millions of lines of code into an IDE. “Nobody would build a car by hand,” he says. “Code is still, in many places, handicraft. When you’re crafting manually 10,000 lines of code, that’s okay. But you have systems that have 30 million lines of code, like an Airbus, or 100 million lines of code, like your Tesla or high-end cars—that’s becoming very, very complicated.”

Bantégnie's company is one of the pioneers in the industrial use of model-based design, in which you no longer write code directly. Instead, you create a kind of flowchart that describes the rules your program should follow (the "model"), and the computer generates code for you based on those rules. If you were making the control system for an elevator, for instance, one rule might be that when the door is open, and someone presses the button for the lobby, you should close the door and start moving the car. In a model-based design tool, you'd represent this rule with a small diagram, as though drawing the logic out on a whiteboard, made of boxes that represent different states —like "door open," "moving," and "door closed"—and lines that define how you can get from one state to the other. The diagrams make the system's rules obvious: Just by looking, you can see that the only way to get the elevator moving is to close the door, or that the only way to get the door open is to stop.

It's not quite Photoshop. The beauty of Photoshop, of course, is that the picture you're manipulating on the screen *is* the final product. In model-based design, by contrast, the picture on your screen is more like a blueprint. Still, making software this way is qualitatively different than traditional programming. In traditional programming, your task is to take complex rules and translate them into code; most of your energy is spent doing the translating, rather than thinking about the rules themselves. In the model-based approach, all you *have* is the rules. So that's what you spend your time thinking about. It's a way of focusing less on the machine and more on the problem you're trying to get it to solve.

"Typically the main problem with software coding—and I'm a coder myself," Bantégnie says, "is not the skills of the coders. The people know how to code. The problem is *what* to code. Because most of the requirements are kind of natural language, ambiguous, and a requirement is never extremely precise, it's often understood differently by the guy who's supposed to code."

On this view, software becomes unruly because the media for describing what software *should* do—conversations, prose descriptions, drawings on a sheet of paper—are too different from the media describing what software *does* do, namely, code itself. Too much is lost going from one to the other. The idea behind model-based design is to close the gap. The very same model is used both by system

designers to express what they want and by the computer to automatically generate code.

Of course, for this approach to succeed, much of the work has to be done well before the project even begins. Someone first has to build a tool for developing models that are natural for people—that feel just like the notes and drawings they'd make on their own—while still being unambiguous enough for a computer to understand. They have to make a program that turns these models into real code. And finally they have to prove that the generated code will always do what it's supposed to. "We have benefited from fortunately 20 years of initial background work," Bantégnie says.

Esterel Technologies, which was acquired by ANSYS in 2012, grew out of research begun in the 1980s by the French nuclear and aerospace industries, who worried that as safety-critical code ballooned in complexity, it was getting harder and harder to keep it free of bugs. "I started in 1988," says Emmanuel Ledinot, the Head of Scientific Studies for Dassault Aviation, a French manufacturer of fighter jets and business aircraft. "At the time, I was working on military avionics systems. And the people in charge of integrating the systems, and debugging them, had noticed that the number of bugs was increasing." The 80s had seen a surge in the number of onboard computers on planes. Instead of a single flight computer, there were now dozens, each responsible for highly specialized tasks related to control, navigation, and communications. Coordinating these systems to fly the plane as data poured in from sensors and as pilots entered commands required a symphony of perfectly timed reactions. "The handling of these hundreds of and even thousands of possible events in the right order, at the right time," Ledinot says, "was diagnosed as the main cause of the bug inflation."

Ledinot decided that writing such convoluted code by hand was no longer sustainable. It was too hard to understand what it was doing, and almost impossible to verify that it would work correctly. He went looking for something new. "You must understand that to change tools is extremely expensive in a process like this," he said in a talk. "You don't take this type of decision unless your back is against the wall."

He began collaborating with Gerard Berry, a computer scientist at INRIA, the French computing-research center, on a tool called Esterel

—a portmanteau of the French for "real-time." The idea behind Esterel was that while traditional programming languages might be good for describing simple procedures that happened in a predetermined order—like a recipe—if you tried to use them in systems where lots of events could happen at nearly any time, in nearly any order—like in the cockpit of a plane—you inevitably got a mess. And a mess in control software was dangerous. In a paper, Berry went as far as to predict that "low-level programming techniques will not remain acceptable for large safety-critical programs, since they make behavior understanding and analysis almost impracticable."

Esterel was designed to make the computer handle this complexity for you. That was the promise of the model-based approach: Instead of writing normal programming code, you created a model of the system's behavior—in this case, a model focused on how individual events should be handled, how to prioritize events, which events depended on which others, and so on. The model becomes the detailed blueprint that the computer would use to do the actual programming.

Ledinot and Berry worked for nearly 10 years to get Esterel to the point where it could be used in production. "It was in 2002 that we had the first operational software-modeling environment with automatic code generation," Ledinot told me, "and the first embedded module in Rafale, the combat aircraft." Today, the ANSYS SCADE product family (for "safety-critical application development environment") is used to generate code by companies in the aerospace and defense industries, in nuclear power plants, transit systems, heavy industry, and medical devices. "My initial dream was to have SCADE-generated code in every plane in the world," Bantégnie, the founder of Esterel Technologies, says, "and we're not very far off from that objective." Nearly all safety-critical code on the Airbus A380, including the system controlling the plane's flight surfaces, was generated with ANSYS SCADE products.

Part of the draw for customers, especially in aviation, is that while it is possible to build highly reliable software by hand, it can be a Herculean effort. Ravi Shivappa, the VP of group software engineering at Meggitt PLC, an ANSYS customer which builds components for airplanes, like pneumatic fire detectors for engines, explains that traditional projects begin with a massive requirements document in English, which specifies everything the software should do. (A requirement might be something like, "When the pressure in this section rises above a

threshold, open the safety valve, unless the manual-override switch is turned on.") The problem with describing the requirements this way is that when you implement them in code, you have to painstakingly check that each one is satisfied. And when the customer changes the requirements, the code has to be changed, too, and tested extensively to make sure that nothing else was broken in the process.

The cost is compounded by exacting regulatory standards. The FAA is fanatical about software safety. The agency mandates that every requirement for a piece of safety-critical software be traceable to the lines of code that implement it, and vice versa. So every time a line of code changes, it must be retraced to the corresponding requirement in the design document, and you must be able to demonstrate that the code actually satisfies the requirement. The idea is that if something goes wrong, you're able to figure out why; the practice brings order and accountability to large codebases. But, Shivappa says, "it's a very labor-intensive process." He estimates that before they used model-based design, on a two-year-long project only two to three months was spent writing code—the rest was spent working on the documentation.

As Bantégnie explains, the beauty of having a computer turn your requirements into code, rather than a human, is that you can be sure—in fact you can mathematically prove—that the generated code actually satisfies those requirements. Much of the benefit of the model-based approach comes from being able to add requirements on the fly while still ensuring that existing ones are met; with every change, the computer can verify that your program still works. You're free to tweak your blueprint without fear of introducing new bugs. Your code is, in FAA parlance, "correct by construction."

Still, most software, even in the safety-obsessed world of aviation, is made the old-fashioned way, with engineers writing their requirements in prose and programmers coding them up in a programming language like C. As Bret Victor made clear in his essay, model-based design is relatively unusual. "A lot of people in the FAA think code generation is magic, and hence call for greater scrutiny," Shivappa told me.

Most programmers feel the same way. They *like* code. At least they understand it. Tools that write your code for you and verify its correctness using the mathematics of "finite-state machines" and

"recurrent systems" sound esoteric and hard to use, if not just too good to be true.

It is a pattern that has played itself out before. Whenever programming has taken a step away from the writing of literal ones and zeros, the loudest objections have come from programmers. Margaret Hamilton, a celebrated software engineer on the Apollo missions—in fact the coiner of the phrase "software engineering"—told me that during her first year at the Draper lab at MIT, in 1964, she remembers a meeting where one faction was fighting the other about transitioning away from "some very low machine language," as close to ones and zeros as you could get, to "assembly language." "The people at the lowest level were fighting to keep it. And the arguments were so similar: 'Well how do we know assembly language is going to do it right?'"

"Guys on one side, their faces got red, and they started screaming," she said. She said she was "amazed how emotional they got."

Emmanuel Ledinot, of Dassault Aviation, pointed out that when assembly language was itself phased out in favor of the programming languages still popular today, like C, it was the assembly programmers who were skeptical this time. No wonder, he said, that "people are not so easily transitioning to model-based software development: They perceive it as another opportunity to lose control, even more than they have already."

The bias against model-based design, sometimes known as model-driven engineering, or MDE, is in fact so ingrained that according to a recent paper, "Some even argue that there is a stronger need to investigate people's perception of MDE than to research new MDE technologies."

Which sounds almost like a joke, but for proponents of the model-based approach, it's an important point: We already know how to make complex software reliable, but in so many places, we're choosing not to. Why?

In 2011, Chris Newcombe had been working at Amazon for almost seven years, and had risen to be a principal engineer. He had

worked on some of the company's most critical systems, including the retail-product catalog and the infrastructure that managed every Kindle device in the world. He was a leader on the highly prized Amazon Web Services team, which maintains cloud servers for some of the web's biggest properties, like Netflix, Pinterest, and Reddit. Before Amazon, he'd helped build the backbone of Steam, the world's largest online-gaming service. He is one of those engineers whose work quietly keeps the internet running. The products he'd worked on were considered massive successes. But all he could think about was that buried deep in the designs of those systems were disasters waiting to happen.

"Human intuition is poor at estimating the true probability of supposedly 'extremely rare' combinations of events in systems operating at a scale of millions of requests per second," he wrote in a paper. "That human fallibility means that some of the more subtle, dangerous bugs turn out to be errors in design; the code faithfully implements the intended design, but the design fails to correctly handle a particular 'rare' scenario."

Newcombe was convinced that the algorithms behind truly critical systems—systems storing a significant portion of the web's data, for instance—ought to be not just good, but perfect. A single subtle bug could be catastrophic. But he knew how hard bugs were to find, especially as an algorithm grew more complex. You could do all the testing you wanted and you'd never find them all.

This is why he was so intrigued when, in the appendix of a paper he'd been reading, he came across a strange mixture of math and code—or what looked like code—that described an algorithm in something called "TLA+." The surprising part was that this description was said to be mathematically precise: An algorithm written in TLA+ could in principle be *proven* correct. In practice, it allowed you to create a realistic model of your problem and test it not just thoroughly, but *exhaustively*. This was exactly what he'd been looking for: a language for writing perfect algorithms.

TLA+, which stands for "Temporal Logic of Actions," is similar in spirit to model-based design: It's a language for writing down the requirements—TLA+ calls them "specifications"—of computer programs. These specifications can then be completely verified by a

computer. That is, before you write any code, you write a concise outline of your program's logic, along with the constraints you need it to satisfy (say, if you were programming an ATM, a constraint might be that you can never withdraw the same money twice from your checking account). TLA+ then exhaustively checks that your logic does, in fact, satisfy those constraints. If not, it will show you exactly how they could be violated.

The language was invented by Leslie Lamport, a Turing Award–winning computer scientist. With a big white beard and scruffy white hair, and kind eyes behind large glasses, Lamport looks like he might be one of the friendlier professors at the American Hogwarts. Now at Microsoft Research, he is known as one of the pioneers of the theory of "distributed systems," which describes any computer system made of multiple parts that communicate with each other. Lamport's work laid the foundation for many of the systems that power the modern web.

For Lamport, a major reason today's software is so full of bugs is that programmers jump straight into writing code. "Architects draw detailed plans before a brick is laid or a nail is hammered," he wrote in an article. "But few programmers write even a rough sketch of what their programs will do before they start coding." Programmers are drawn to the nitty-gritty of coding because code is what makes programs go; spending time on anything else can seem like a distraction. And there is a patient joy, a meditative kind of satisfaction, to be had from puzzling out the micro-mechanics of code. But code, Lamport argues, was never meant to be a medium for thought. "It really does constrain your ability to think when you're thinking in terms of a programming language," he says. Code makes you miss the forest for the trees: It draws your attention to the working of individual pieces, rather than to the bigger picture of how your program fits together, or what it's supposed to do —and whether it actually does what you think. This is why Lamport created TLA+. As with model-based design, TLA+ draws your focus to the high-level structure of a system, its essential logic, rather than to the code that implements it.

Newcombe and his colleagues at Amazon would go on to use TLA+ to find subtle, critical bugs in major systems, including bugs in the core algorithms behind S3, regarded as perhaps the most reliable storage engine in the world. It is now used widely at the company. In the tiny universe of people who had ever used TLA+, their success was not so

unusual. An intern at Microsoft used TLA+ to catch a bug that could have caused every Xbox in the world to crash after four hours of use. Engineers at the European Space Agency used it to rewrite, with 10 times less code, the operating system of a probe that was the first to ever land softly on a comet. Intel uses it regularly to verify its chips.

But TLA+ occupies just a small, far corner of the mainstream, if it can be said to take up any space there at all. Even to a seasoned engineer like Newcombe, the language read at first as bizarre and esoteric—a zoo of symbols. For Lamport, this is a failure of education. Though programming was born in mathematics, it has since largely been divorced from it. Most programmers aren't very fluent in the kind of math—logic and set theory, mostly—that you need to work with TLA+. "Very few programmers—and including very few teachers of programming—understand the very basic concepts and how they're applied in practice. And they seem to think that all they need is code," Lamport says. "The idea that there's some higher level than the code in which you need to be able to think precisely, and that mathematics actually allows you to think precisely about it, is just completely foreign. Because they never learned it."

Lamport sees this failure to think mathematically about what they're doing as the problem of modern software development in a nutshell: The stakes keep rising, but programmers aren't stepping up—they haven't developed the chops required to handle increasingly complex problems. "In the 15th century," he said, "people used to build cathedrals without knowing calculus, and nowadays I don't think you'd *allow* anyone to build a cathedral without knowing calculus. And I would hope that after some suitably long period of time, people won't be allowed to write programs if they don't understand these simple things."

Newcombe isn't so sure that it's the programmer who is to blame. "I've heard from Leslie that he thinks programmers are afraid of math. I've found that programmers aren't aware—or don't believe—that math can help them handle complexity. Complexity is the biggest challenge for programmers." The real problem in getting people to use TLA+, he said, was convincing them it wouldn't be a waste of their time. Programmers, as a species, are relentlessly pragmatic. Tools like TLA+ reek of the ivory tower. When programmers encounter "formal

methods" (so called because they involve mathematical, "formally" precise descriptions of programs), their deep-seated instinct is to recoil.

Most programmers who took computer science in college have briefly encountered formal methods. Usually they're demonstrated on something trivial, like a program that counts up from zero; the student's job is to mathematically prove that the program does, in fact, count up from zero.

"I needed to change people's perceptions on what formal methods were," Newcombe told me. Even Lamport himself didn't seem to fully grasp this point: Formal methods had an image problem. And the way to fix it wasn't to implore programmers to change—it was to change yourself. Newcombe realized that to bring tools like TLA+ to the programming mainstream, you had to start speaking their language.

For one thing, he said that when he was introducing colleagues at Amazon to TLA+ he would avoid telling them what it stood for, because he was afraid the name made it seem unnecessarily forbidding: "Temporal Logic of Actions" has exactly the kind of highfalutin ring to it that plays well in academia, but puts off most practicing programmers. He tried also not to use the terms "formal," "verification," or "proof," which reminded programmers of tedious classroom exercises. Instead, he presented TLA+ as a new kind of "pseudocode," a stepping-stone to real code that allowed you to exhaustively test your algorithms—and that got you thinking precisely early on in the design process. "Engineers think in terms of debugging rather than 'verification,'" he wrote, so he titled his internal talk on the subject to fellow Amazon engineers "Debugging Designs." Rather than bemoan the fact that programmers see the world in code, Newcombe embraced it. He knew he'd lose them otherwise. "I've had a bunch of people say, 'Now I get it,'" Newcombe says.

He has since left Amazon for Oracle, where he's been able to convince his new colleagues to give TLA+ a try. For him, using these tools is now a matter of responsibility. "We need to get better at this," he said.

"I'm self-taught, been coding since I was nine, so my instincts were to start coding. That was my only—that was my way of thinking: You'd sketch something, try something, you'd organically evolve it." In his view, this is what many programmers today still do. "They google, and

they look on Stack Overflow" (a popular website where programmers answer each other's technical questions) "and they get snippets of code to solve their tactical concern in this little function, and they glue it together, and iterate."

"And that's completely fine until you run smack into a real problem."

In the summer of 2015, a pair of American security researchers, Charlie Miller and Chris Valasek, convinced that car manufacturers weren't taking software flaws seriously enough, demonstrated that a 2014 Jeep Cherokee could be remotely controlled by hackers. They took advantage of the fact that the car's entertainment system, which has a cellular connection (so that, for instance, you can start your car with your iPhone), was connected to more central systems, like the one that controls the windshield wipers, steering, acceleration, and brakes (so that, for instance, you can see guidelines on the rearview screen that respond as you turn the wheel). As proof of their attack, which they developed on nights and weekends, they hacked into Miller's car while a journalist was driving it on the highway, and made it go haywire; the journalist, who knew what was coming, panicked when they cut the engines, forcing him to a slow crawl on a stretch of road with no shoulder to escape to.

Although they didn't actually create one, they showed that it was possible to write a clever piece of software, a "vehicle worm," that would use the onboard computer of a hacked Jeep Cherokee to scan for and hack others; had they wanted to, they could have had simultaneous access to a nationwide fleet of vulnerable cars and SUVs. (There were at least five Fiat Chrysler models affected, including the Jeep Cherokee.) One day they could have told them all to, say, suddenly veer left or cut the engines at high speed.

"We need to think about software differently," Valasek told me. Car companies have long assembled their final product from parts made by hundreds of different suppliers. But where those parts were once purely mechanical, they now, as often as not, come with millions of lines of code. And while some of this code—for adaptive cruise control, for auto braking and lane assist—has indeed made cars safer ("The safety features on my Jeep have already saved me countless times," says

Miller), it has also created a level of complexity that is entirely new. And it has made possible a new kind of failure.

"There are lots of bugs in cars," Gerard Berry, the French researcher behind Esterel, said in a talk. "It's not like avionics—in avionics it's taken very seriously. And it's admitted that software is different from mechanics." The automotive industry is perhaps among those that haven't yet realized they are actually in the software business.

"We don't in the automaker industry have a regulator for software safety that knows what it's doing," says Michael Barr, the software expert who testified in the Toyota case. NHTSA, he says, "has only limited software expertise. They've come at this from a mechanical history." The same regulatory pressures that have made model-based design and code generation attractive to the aviation industry have been slower to come to car manufacturing. Emmanuel Ledinot, of Dassault Aviation, speculates that there might be economic reasons for the difference, too. Automakers simply can't afford to increase the price of a component by even a few cents, since it is multiplied so many millionfold; the computers embedded in cars therefore have to be slimmed down to the bare minimum, with little room to run code that hasn't been hand-tuned to be as lean as possible. "Introducing model-based software development was, I think, for the last decade, too costly for them."

One suspects the incentives are changing. "I think the autonomous car might push them," Ledinot told me—"ISO 26262 and the autonomous car might slowly push them to adopt this kind of approach on critical parts." (ISO 26262 is a safety standard for cars published in 2011.) Barr said much the same thing: In the world of the self-driving car, software can't be an afterthought. It can't be built like today's airline-reservation systems or 911 systems or stock-trading systems. Code will be put in charge of hundreds of millions of lives on the road and it has to work. That is no small task.

"Computing is fundamentally invisible," Gerard Berry said in his talk. "When your tires are flat, you look at your tires, they are flat. When your software is broken, you look at your software, you see nothing."

"So that's a big problem."

This story was originally published in The Atlantic.