What’s a Robot?

I am a big science fiction fan and robots have played a major role in some of my favorite speculative universes. The prototypical robot story came in the form of a play by Karel Čapek called “R.U.R.” that stood for “Rossum’s Universal Robots.” Written in the 1920s, it envisaged android-like robots that were sentient and were created to serve humans. “Robot” came from the Russian word “робота” (“робота,” which means “work”). Needless to say, the story does not come out well for the humans. In a more benign and very complex scenario, Isaac Asimov created a universe in which robots with “positronic” brains serve humans and are barred by the Three Laws of Robotics from harming humans:

1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey the orders given to it by human beings, except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Laws.

A “zeroth” law emerges later:

0. A robot may not harm humanity, or, by inaction, allow humanity to come to harm.

In most formulations, robots have the ability to manipulate and affect the real world. Examples include robots that assemble cars (or at least parts of them). Less facile robots might be devices that fill cans with food or bottles with liquid and then close them up. The most primitive robots might not normally even be considered robots in normal parlance. One example is a temperature control for a home heating system that relies on a piece of bi-metal material that expands differentially causing a circuit to be closed or opened depending on the ambient temperature.

I would like to posit, however, that the notion of robot could usefully be expanded to include programs that perform functions, ingest input and produce output that has a perceptible effect. A weak notion along these lines might be simulations in which the real world remains unaffected. A more compelling example might be high-frequency stock trading systems whose actions have very real-world consequences in the financial sector. While nothing physical happens, real-world accounts are impacted and, in some cases, serious consequences emerge if the programs go out of control leading to rapid market excursions. Some market meltdowns have been attributed to large numbers of high-frequency trading programs all reacting in similar ways to inputs leading to rapid upward or downward motion of the stock market.

Following this line of reasoning, one might conclude that we should treat as robots any programs that can have real-world, if not physical, effects. I am not quite sure where I am heading with this except to suggest that those of us who live in and participate in creation of software-based “universes” might wisely give thought to the potential impact that our software might have on the real world. Establishing a sense of professional responsibility in the computing community might lead to increased safety and reliability of software products and services. This is not to suggest that today’s programmers are somehow irresponsible but I suspect that we are not uniformly cognizant of the side effects of great dependence on software products and services that seems to increase daily.

A common theme I hear in many conversations is concern for the fragility or brittleness of our networked-and software-driven world. We rely deeply on software-based infrastructure and when it fails to function, there can be serious side effects. Like most infrastructure, we tend not to think about it at all until it does not work or is not available. Most of us do not lie awake worried that the power will go out (but, we do rely on some people who do worry about these things). When the power goes out, we suddenly become aware of the finiteness of battery power or the huge role that electricity plays in our daily lives. Mobile phones went out during Hurricane Sandy because the cell towers and base stations ran out of power either because of battery failure or because the back-up generators could not be supplied with fuel or could not run because they were underwater.

I believe it would be a contribution to our society to encourage deeper thinking about what we in the computing world produce, the tools we use to produce them, the resilience and reliability that these products exhibit and the risks that they may introduce. For decades now, Peter Neumann has labored in this space, documenting and researching the nature of risk and how it manifests in the software world. We would all do well to emulate his lead and to think whether it is possible that the three or four laws of robotics might motivate our own aspirations as creators in the endless universe of software and communications.
In Search of Dependable Design

How can software and hardware developers increase the reliability of their designs?

In 1994, an obscure circuitry error was discovered in Intel's Pentium I microprocessor. Thomas R. Nicely, a mathematician then affiliated with Lynchburg College in Virginia, noticed that the chip gave incorrect answers to certain floating-point division calculations. Other researchers soon confirmed the problem and identified additional examples. And though Intel initially tried to downplay the mistake, the company eventually responded to mounting public pressure by offering to replace each one of the flawed processors.

"It was the first error to make the evening news," recalls Edmund Clarke of Carnegie Mellon University. The cost to the company: around $500 million.

Nearly 15 years later, the Pentium bug continues to serve as a sobering reminder of how expensive design flaws can be. The story is no different for software: a $170 million virtual case management system was scrapped by the FBI in 2005 due to numerous failings, and a flawed IRS tax-processing system consumed billions of dollars in the late 1990s before it was finally fixed. And in an era in which people rely on computers in practically every aspect of their lives—in cars, cell phones, airplanes, ATMs, and more—the cost of unreliable design is only getting higher. Data is notoriously difficult to come by, but a 2002 study conducted by the National Institute of Standards and Technology (NIST) estimated that faulty software alone costs the U.S. economy as much as $59.5 billion a year in lost information, squandered productivity, and increased repair and maintenance.

But it’s not just a matter of money—increasingly, people’s lives are at stake. Faulty software has plunged cockpit displays into darkness, sunk oil rigs, and caused missiles to malfunction.

“There have been only a few real disasters due to software. But we’re walking closer and closer to the edge,” says MIT’s Daniel Jackson.

Experts agree that flaws typically arise not from minor bugs in code, but during the higher-level design process. (Security flaws, which tend to be caused by implementation-level vulnerabilities, are often an exception to this rule.) One class of problems arises at the requirements phase: program design requirements are often poorly articulated, or poorly understood. Another class arises from insufficient human factors design, where engineers make unwarranted assumptions about the environment in which software or hardware will operate. If a program isn’t capable of handling those unforeseen conditions, it may fail.

But mistakes can happen at any time. “Since humans aren't perfect, humans make mistakes, and mistakes can be made in any step of the development process,” cautions Gerard Holzmann of the NASA/JPL Laboratory for Reliable Software.

Holzmann is among a small group of researchers who are committed to developing tools, techniques, and procedures for increasing design reliability. Currently, most programs are debugged and then refined by random testing. Testing can be useful to pinpoint smaller errors, say researchers, but inadequate when it comes to identifying structural ones. And tests designed for specific scenarios may not be able to explore combinations of behavior that fall outside of anticipated patterns. The search is therefore on for additional strategies.

One promising technique is known as model checking. The idea is to verify the logic behind a particular software or hardware design by constructing a mathematical model and using an algorithm to make sure it satisfies certain requirements. Though the task can be time consuming, it forces developers to articulate their requirements in a systematic, mathematical way, thereby minimizing ambiguity. More importantly, however, model checkers automatically give diagnostic counterexamples when mistakes are found, helping developers pinpoint what went wrong and catch flaws before they are coded.

“When people use the term ‘reliability,’ they might have some probabilistic notion that ‘only rarely’ do errors crop up, whereas people in the formal verification community mean that all behaviors are correct against all specified criteria,” explains Allen Emerson of the University of Texas at Austin. (In recognition of the importance of formal verification techniques, the 2007 ACM A.M. Turing Award was given to Edmund Clarke, Allen Emerson, and Joseph Sifakis for their pioneering work in model checking. A Q&A with...
the three Turing recipients can be found on page 112.)

Model checking has proven extremely successful at verifying hardware designs. In fact, Xudong Zhao, a graduate student of Clarke’s, showed that model checking could have found Intel’s floating-point division error—and that the company’s fix did indeed correct the problem. Since then, Intel has been a leading user of the technique.

But even small programs can have millions of different states (a dilemma known to the discipline as the “state explosion problem”), there are limits to the size and complexity of designs that model checking can verify, and it’s been less immediately successful for software. The verification of reactive systems—the combination of hardware and software interacting with an external environment—also remains problematic, due mainly to the difficulty of constructing faithful models.

“We’ve come a long way in the last 28 years, and there’s a huge, huge difference in the scale of problems we can address now as opposed to 1980,” says Holzmann. “But of course we are more ambitious and our applications have gotten more complex, so there is a lot more to be done.”

Other techniques include specialized programming languages and environments that facilitate the creation of reliable, reusable software modules. Eiffel, developed by the Swiss Federal Institute of Technology’s Bertrand Meyer and recipient of ACM’s 2006 Software System Award, is one well-known example; Alloy, a tool developed by Daniel Jackson and the MIT Software Design Group, has also shown great promise.

To supplement the new languages and techniques, other researchers have focused on outlining more effective procedures and methodologies for developers to follow as they work.

“I’m not a great believer in formal analysis,” says Grady Booch of IBM Research. “Problems tend to appear at this curious intersection of the technological and the social.” After monitoring 50 developers for 24 hours, for example, Booch found that only 30% of their time was spent coding—the rest was spent talking to other members of their team. Avoiding miscommunication, he believes, is therefore critical. Booch is perhaps best known for developing (with Ivar Jacobson and James Rumbaugh) the Unified Markup Language, or UML, a language that uses graphical notations to create an abstract model of a software or hardware system and helps teams communicate, explore, and validate potential designs. More recently, he has continued to focus on the big picture of development with the online Handbook of Software Architecture, which brings together a large collection of software-intensive systems and presents them in a manner that “exposes their essential patterns and that permits comparisons across domains and architectural styles.” The ultimate goal, of course, is to help developers apply that time-tested knowledge to their own programming projects.

“Reuse is easier at a higher level of abstraction,” explains Booch. “So we can reuse patterns, if not necessarily code.”

MIT’s Daniel Jackson is another strong believer in the “big picture” approach. “The first thing we need to do is be honest about the level of reliability that we need,” he asserts. “The second thing is to think about what really cannot go wrong—about what’s mission critical and what’s not.”

Rather than starting with a typical requirements document that outlines...
tasks in a procedural way, says Jackson, developers must first make sure they understand what the system is really about. What are its essential properties? Who are its stakeholders? What level of dependability does it need?

“How can you ever hope to build a dependable system if you don’t know what ‘dependable’ means?” he asks. The task itself is abstract, but Jackson believes that articulating all requirements and assumptions is crucial to tackling it—ideally in a formal, methodological way. The most important thing, according to Jackson, is the act of articulation itself. “When you write things down, you often find that you didn’t understand them nearly as well as you thought you did.” And there’s always a temptation to jump to the solution before you’ve fully understood the problem. “That’s not to say that automated tools and techniques like model checking aren’t useful, of course. Tools are an important support, but they’re secondary,” says Jackson.

And the more safety-critical the application, the more rigorous developers must be. “If your computer crashes, it’s inconvenient, but it’s not a threat to anyone’s life,” says Holzmann. Among the approaches he and his lab—who work to guarantee the safety of the computer systems that run spacecraft—are currently looking into is the development of simple, yet effective, coding standards. His recommendations may seem somewhat draconian (in safety-critical applications, they forbid the use of goto statements, set jmp or long jmp constructs, and direct or indirect recursion, for example), but they are intended to increase simplicity, prevent common coding mistakes, and force developers to create more logical architectures. Simpler programs are also easier to verify with tools like model checkers. After overcoming their initial reluctance, Holzmann says, developers often find that the restrictions are a worthwhile trade-off for increased safety.

A rigorous focus on simplicity can be costly, of course, especially for complex legacy systems that would be prohibitively expensive to replace but that need, nonetheless, to be updated or further developed. So can taking the time out to formally articulate all requirements and assumptions, or to verify software designs. Yet the cost of fixing an error in the initial stages of development is far less than fixing it at the end—a lesson that Intel, for one, now knows well.

“Computer science is a very young discipline,” explains Joseph Sifakis, research director at CNRS. “We don’t have a theory that can guarantee system reliability, that can tell us how to build systems that are correct by construction. We only have some recipes about how to write good programs and how to design good hardware. We’re learning by a trial-and-error process.”

Simpler programs are easier to verify with tools like model checkers.

Computer Science Winning Strategy
St. Petersburg University of Information Technology, Mechanics and Optics recently won the 32nd annual ACM International Collegiate Programming Contest (ICPC) World Finals, held in Braniff, Canada. It was the university’s second ACM-ICPC world championship in four years.

The annual programming contest started with 6,700 teams from 1,821 universities in 83 countries, competing at 213 sites around the world. Through a series of regional competitions, the field narrowed to 100 teams. At the World Finals, each three-person team had one computer and five hours to solve 11 programming problems.

“The main goal at the World Finals is to solve problems,” says Andrey Stankevich, coach of the St. Petersburg University of Information Technology, Mechanics and Optics team, who was interviewed via email. “If you use your time to solve problems (and not to look for bugs in the problems already solved, but not accepted by the judges) you have time to solve more. So, the way to win the World Finals is to solve problems in such way that you don’t make bugs, and if the problem is accepted, you can immediately start solving another one. This requires cooperation in both thinking about problems and writing code.”

The winning team solved eight problems, followed by second-place Massachusetts Institute of Technology, third-place Izhevsk State Technical University, fourth-place Lviv National University and fifth-place Moscow State University, each of which solved seven problems.

The competition at each ACM-ICPC World Finals appears to be stronger than the previous one, and longtime contest sponsor IBM believes the global contest is good for the IT industry. “The value proposition for IBM is not only about the students who go on to work for IBM, but who go on to work for our clients and our business partners, or who become faculty members,” says IBM director of talent Margaret Ashida. “It’s a win for everyone.”
It's no secret that engineers and designers constantly seek to build safer and more convenient systems. And, over the last century, planes, trains, automobiles, and industrial machines have become far more automated and efficient. However, when a Metro subway train rammed into another train in Washington, D.C. last June, designers had to confront the unpleasant reality that automation may have been the cause. The accident, which killed nine people and injured 80, may have been rooted in a computer malfunction and the operator's inability to manually apply the brakes quickly enough.

The Metro train accident lies at the heart of what human factors experts refer to as the “automation paradox.” As automated systems become increasingly reliable and efficient, the more likely it is that human operators will mentally “switch off” and rely upon the automated system. And as the automated system becomes more complex, the odds of an accident or mishap may diminish, but the severity of a failure is often amplified.

As John D. Lee, a professor of industrial and systems engineering at the University of Wisconsin at Madison told the Washington Post: “The better you make the automation, the more difficult it is to guard against these catastrophic failures....”

Understanding how people and machines interact is infinitely complex. Programming all the various possibilities and scenarios into a system can tax even the best design and engineering experts. What's more, as technology evolves, the entire process grows more convoluted and iterative. In some cases, experts say, it's wise to ask what purpose automation serves and when it's best to use it and eschew it.

What is the fallout from automation glitches? Where do programmers, designers, and engineers typically fall short? And what can technologists do to build better systems? There are no simple solutions. But as Donald Norman, professor of computer science at Northwestern University, co-founder of Neilsen Norman Group, and author of The Design of Future Things, says, “Designers often make assumptions or act on incomplete information. They simply don't anticipate how systems will be used and how unanticipated events and consequences will occur.”

**Human-Machine Interface**

It's clear that automation has provided enormous gains to society. Safer and more efficient factories; faster police, emergency, and fire response; and more user-friendly and safer automobiles are only a few of the benefits. Yet, at the same time, it takes little effort to find evidence of breakdowns between human and machine.

The crash of Air France Flight 447 that occurred over the Atlantic Ocean last June—killing all 228 people aboard—may have been caused by a malfunction in a speed sensor. The plane's Pitot tubes, a pressure measurement instrument used to track fluid flow velocity, may have became blocked by ice. At that point, they may have stopped emitting signals, and experts say that the pilots could have encountered false speed readings. In fact, the jet—which was coping with a series of storms, including a severe thunderstorm—reportedly relayed a signal that its computer system no longer knew the speed of the aircraft, and that automatic pilot and thrust functions were switched off. This may have forced the pilots to take over manual control during chaotic, if not impossible, flying conditions.

There are also plenty of examples of humans having trouble with automation systems in everyday life. As automobiles become more automated, new problems crop up. For instance, motorists blindly follow the incorrect directions provided by a navigation system, even though a glance at the road would indicate there's an obvious error. A few motorists have even driven off a cliff or into oncoming traffic after following directions explicitly. What's more, studies show that many motorists use automation features, such as...
adaptive cruise control, incorrectly. In some cases, Norman says, these automated systems cause the car to speed up as motorists exit a highway because there’s suddenly no car in front. If a driver isn’t paying attention, an accident can occur.

In the case of airplane pilots and train operators, one solution is regular training sessions in which the pilot or operator is required to turn off their automated system and operate everything manually. This can help them retain their skills and alertness.

But even this is not likely to eliminate breakdowns. Human-machine interface failures occur for a number of reasons, experts say. Sometimes, designers rely on a wrong set of assumptions to build a system. They simply don’t understand the way people use technology or the cultural differences that occur. In some instances, thousands and sometimes millions of variables exist and capturing everything in a single algorithm is exceedingly difficult. In fact, Norman argues that machine logic doesn’t necessarily jibe with the human brain. “If you look at ‘human error’ it almost always occurs when people are forced to think and act like machines,” he says.

Worse, complex algorithms often prompt humans to relate to devices as if they were fellow human beings. As a result, the autopilot on a plane, the cruise control on a car, and automated speed-control systems in mass transit become either aids or crutches, depending the situation.

Too often, the sum of a system is not equal to the individual parts, says Sidney W. A. Dekker, director of research at the Leonardo da Vinci Center for Complexity and Systems Thinking at Lund University in Sweden. “There is often a great deal of human intuition involved in a process or activity and that’s not something a machine can easily duplicate,” says Dekker. “If you look at delivering babies, there’s a reason we have midwives and nurses. Machines can monitor and help, but they can’t detect subtle signs and they’re unable to adapt to situations as seamlessly.”

David D. Woods, professor of cognitive engineering at Ohio State University, says that designers can easily succumb to the trap of thinking “a little more technology will solve the problem.” However, understanding variables and identifying possible exceptions and disruptions is paramount. For example, when the Metro D.C. train crashed, it may have been due to wet leaves on the tracks and a computerized system that wasn’t programmed for such a scenario. “The automation system functioned as it was designed,” Woods says. “The situation simply fell outside the model of what engineers envisioned.”

Beyond Failure
Make no mistake, human factors experts constantly scrutinize automation. Many believe that if human error exists, it falls on the shoulders of those engineering, designing, and programming technology. “In reality, there is no such thing as operator error. Too often, systems aren’t designed as whole and those creating them overlook important factors,” argues Nancy Leveson, professor of aeronautics and astronautics at Massachusetts Institute of Technology and author of the forthcoming book Engineering a Safer World.

Yet, progress is taking place. Consider the airline industry: In 1989, 1.4 crashes per 1 million departures occurred. By 2008, the number had dropped to 0.2 fatal accidents per 1 million departures. In fact, crashes have steadily dropped over the decades while survivability has increased.

Dekker, who is a pilot and has flown various aircraft, including a Boeing 737, says that the industry has gotten serious about stamping out flaws, bugs, and oversights.

These improvements have taken place because the airline industry has moved beyond studying ergonomics and discreet processes. In fact, Leveson says that researchers have put a microscope to cognitive functions, psychology, cultural issues, and a variety of other components that comprise human factors. “They have evolved toward a system view and worked to understand how everything—hardware, software, procedures, and humans—interact. It’s a model that other industries must embrace,” she says.

One thing is certain: Automation disconnects won’t disappear anytime soon. Leveson believes that, ultimately, the people designing systems must take a more holistic view and get past the notion that when a problem or breakdown occurs it’s a result of “human error.” She believes that universities must place a greater focus on human factors and that programmers and others must understand that, without a big picture view of what they are building, the end result will continually fall short.

Others, such as Dekker, argue that society must examine larger issues, including whether automation automatically translates into progress. “In reality, not every function or process is best automated,” he says. “In some cases, automation simply creates new or different tasks and doesn’t provide any real benefit.” Automation may also change processes to the point where people are more confused and entirely new social dynamics take place. At that point, he says, designers may attempt to add new features, which only ratchet up confusion and complexity further.

To be sure, imperfect people continue to build imperfect systems. The need to focus on human-machine interfaces has never been greater. “Designers, engineers, programmers, and others must take an expansive view of automation and understand all the possibilities and variables,” concludes Norman. “Only then can we build systems that improve performance and solve real-world problems.”

Further Reading


Samuel Greengard is an author and freelance writer based in West Line, OR.

© 2009 ACM 0001-0782/09/1200 510.00
Inside Risks
Risks of Undisciplined Development

An illustration of the problems caused by a lack of discipline in software development and our failure to apply what is known in the field.

The branches of engineering (such as civil, electrical, and mechanical), are often referred to as disciplines for good reason. Associated with each specialty is a set of rules that specify:

- checks that must be made;
- properties that must be measured, calculated, or specified;
- documentation that must be provided;
- design review procedures;
- tests that must be carried out on the product; and
- product inspection and maintenance procedures.

Like all professional education, engineering education is designed to prepare students to meet the requirements of the authorities that regulate their chosen profession. Consequently, most graduates are taught they must carry out these procedures diligently and are warned they can be deemed guilty of negligence and lose the right to practice their profession if they do not.

Because they are preparing students for a career that can last many decades, good engineering programs teach fundamental principles that will be valid and useful at the end of the graduate’s career. Engineering procedures are based on science and mathematics; and graduates are expected to understand the reasons for the rules, not just blindly apply them.

These procedures are intended to assure that the engineer’s product:

- will be fit for the use for which it is intended;
- will conform to precise stable standards;
- is robust enough to survive all foreseeable circumstances (including incorrect input); and
- is conservatively designed with appropriate allowance for a margin of error.

In some areas, for example building and road construction, the procedures are enforced by law. In other areas, and when engineers work in industry rather than selling their services directly to the public, employers rely on the professionalism of their employees. Professional engineers are expected to know what must be done and to follow the rules even when their employer wants them to take inappropriate shortcuts.

Anyone who observes engineers at work knows that exercising due diligence requires a lot of “dog work.” The dull, but essential, work begins in the
design phase and continues through construction, testing, inspection, commissioning, and maintenance. Licensed engineers are given a unique seal and instructed to use it to signify the acceptability of design documents only after they are sure the required analysis has been completed by qualified persons.

Real-World Experience
Recent experiences reminded me that the activity we (euphemistically) call software engineering does not come close to deserving a place among the traditional engineering disciplines. Replacing an old computer with a newer model of the same brand revealed many careless design errors—errors that in all likelihood could have been avoided if the developers had followed a disciplined design process. None of the problems was safety critical, but the trouble caused was expensive and annoying for all parties.

My “adventure” began when the sales clerk scanned a bar code to initiate the process of creating a receipt and registering my extended warranty. There were three codes on the box; not surprisingly, the sales clerk scanned the wrong one. This is a common occurrence. The number scanned bore no resemblance to a computer serial number but was accepted by the software without any warning to the clerk. The nonsense number was duly printed as the serial number on my receipt. My extended warranty was registered to a nonexistent product. I was billed, and no problem was noted until I phoned the customer care line with a question. When I read the serial number from the receipt, I was told that I had purchased nothing and was not entitled to ask questions. After I found the correct number on the box, I was told that my computer was not yet in their system although a week had passed since the sale.

Correcting the problem required a trip back to the store and tricking the company computer by returning the nonexistent machine and buying it again. In the process, my name was entered incorrectly and I was unable to access the warranty information online. After repeatedly trying to correct their records, the help staff told me it could not be done.

A different problem arose when I used the migration assistant supplied with the new computer to transfer my data and programs to the new machine. Although the description of the migration assistant clearly states that incompatible applications will be moved to a special directory rather than installed, a common software package on the old machine, one that was not usable or needed on the new one, was installed anyway. A process began to consume CPU time at a high rate. Stopping that process required searching the Internet to find an installer for the obsolete product.

The next problem was an error message informing me that a device was connected to a USB 1.1 port and advising me to move it to a USB 2.0 port. My new computer did not have any 1.1 ports so I called the “care” line for advice. They had no list of error messages and could not guess, or find out, which application or component of their software would issue such a message or under what conditions it should be issued. They referred the problem to developers; I am still waiting for a return call.

These incidents are so petty and so commonplace that readers must wonder why I write about them. It is precisely because such events are commonplace, and so indicative of lack of discipline, that such stories should concern anyone who uses or creates software.

As early as the late 1950s, some compilers came with a complete list of error messages and descriptions of the conditions that caused them. Today, such lists cannot be found. Often,
Even sophisticated and experienced purchasers do not demand the documentation that would be evidence of disciplined design and testing.

lists more than they need to hear about the importance of creativity.

It is obviously important to give courses on picking the most efficient algorithms and to make sure that students graduate prepared to understand current technology and use new technology as it comes along, but neither substitutes for teaching them to be disciplined developers.

Disciplined design is both teachable and doable. It requires the use of the most basic logic, nothing as fancy as temporal logic or any of the best-known formal methods. Simple procedures can be remarkably effective at finding flaws and improving trustworthiness. Unfortunately, they are time-consuming and most decidedly not done by senior colleagues and competitors.

Disciplined software design requires three steps:
1. Determine and describe the set of possible inputs to the software.
2. Partition the input set in such a way that the inputs within each partition are all handled according to a simple rule.
3. State that rule.

Each of these steps requires careful review:
1. Those who know the application must confirm that no other inputs can ever occur.
2. Use basic logic to confirm that every input is in one—and only one—of the partitions.
3. Those who know the application, for example, those who will use the program, must confirm the stated rule is correct for every element of the partition.

These rules seem simple, but reality complicates them:

1. If the software has internal memory, the input space will comprise event sequences, not just current values. Characterizing the set of possible input sequences, including those that should not, but could, happen is difficult. It is very easy to overlook sequences that should not happen.
2. Function names may appear in the characterization of the input set. Verifying the correctness of the proposed partitioning requires knowing the properties of the functions named.
3. The rule describing the output value for some of the partitions may turn out to be complex. This is generally a sign that the partitioning must be revised, usually by refining a partition into two or more smaller partitions. The description of the required behavior for a partition should always be simple but this may imply having more partitions.

Similar “divide and conquer” approaches are available for inspection and testing.

While our failure to teach students to work in disciplined ways is the primary problem, the low standards of purchasers are also a contributing factor. We accept the many bugs we find when a product is first delivered, and the need for frequent error-correcting updates, as inevitable. Even sophisticated and experienced purchasers do not demand the documentation that would be evidence of disciplined design and testing.

We are caught in a catch-22 situation:

- Until customers demand evidence that the designers were qualified and disciplined, they will continue to get sloppy software.
- As long as there is no better software, we will buy sloppy software.
- As long as we buy sloppy software, developers will continue to use undisciplined development methods.
- As long as we fail to demand that developers use disciplined methods, we run the risk—nay, certainty—that we will continue to encounter software full of bugs.

David L. Parnas (parnas@mcmaster.ca) is Professor Emeritus at McMaster University and the University of Limerick as well as President of Middle Road Software. He has been looking for, and teaching, better software development methods for more than 40 years. He is still looking!

Copyright held by author.
THE SO-CALLED LAMP STACK (Linux OS/Apache Internet server/MySQL DBMS/Perl) consists of 10 million lines of code, interacting in myriad ways to achieve impressive functionality and performance. This approaches the intellectual complexity of the Saturn V rocket (with three million parts) that took humans to the moon. The similarities between these two enormous engineering feats are important: good engineering design practices have been followed; requirements were defined and met; rigorous testing and debugging has taken place. Yet, to date, the LAMP stack is much less well understood than the Saturn V rocket. It is much more difficult to predict how the LAMP stack will perform under varying conditions and where things might go wrong than it is to consider how the Saturn V may behave in different operating environments.

What the engineers at NASA have that the developers and users of the LAMP stack do not is an understanding of how configurations of system components (in NASA’s case, physical materials) will behave in a variety of (physical) contexts and an idea of where the boundary conditions of those behaviors lie. This understanding is the product of physical theories about universal laws of nature—laws that have been identified through a tradition of model construction and empirical testing to produce general principles.

The Saturn V rocket that launched humans to the moon relied on an IBM System/360 for data processing.
Developers of the LAMP stack do not approach this level of understanding. If computer science can achieve this level, that is, uncover the kinds of scientific theories and laws that physics provides, software developers building artifacts could do what NASA engineers do now: analyze their designs according to underlying theories and predict how a working system will behave. Extracting such laws and principles is the goal of science.

Whether and how computer science is a science has been a topic of discussion since the beginnings of the field. Thinkers including Herbert Simon have offered deep insight into the special nature of computation and how it exists as a phenomenon—in part, it’s artificial, something we create. However, the nature of what computer science is studying, or how best to study it, is by no means a settled topic. Just because we design and build computational systems does not mean we understand them; special care and much more work is needed to correctly characterize computation as a scientific endeavor.

The early challenges of the field were engineering in nature, leading to a de-emphasis of empirical methods. In recent years, with increasing complexity of computational systems, empirical methods are now needed to discover system limits and predict future behavior.

**The Methodology of Debugging**

Let’s start with the parts of the scientific method we do well. Programmers engage naturally in one of the purest forms of empirical investigation: debugging. Whenever a program exhibits a fault, in which a run over particular input data yields an exception or incorrect output, the programmer generally follows a series of steps. First, the programmer develops a hypothesis of where the fault lies (whether in a particular statement or more generally with the logic of the design) and considers how to correct it. This hypothesis is based on the programmer’s mental model of how the programming environment, referenced libraries, operating system, and environment (such as input data, system events, and so forth) function and interact. This model is generated and refined through additional runs of the program with print statements or through interaction with a debugger in which the program state can be examined as the program is single-stepped. The programmer then tests this hypothesis by making changes believed to fix the problem and rerunning the program, predicting that the change will result in the correct output for that particular test input. If the program still faults, the programmer refines the hypothesis, updating their mental model of how they believe the underlying system is behaving, and further investigates possible causes. Programmers frequently demonstrate they are highly skilled at understanding complex interactions by forming models and testing and revising them.

**Toward General Predictive Models**

Why, then, do we not have the same depth of understanding of the LAMP stack as the Saturn V rocket? Formulating hypotheses, devising tests and carrying them out, and then revising hypotheses based on test outcomes are critical components of the basic methodology of empirical science. But these activities are not all that science does. What is missing from the debugging picture is searching for generalizations: attempting to extract general patterns and identify the factors that govern their behavior. This requires looking past individual systems, past simply getting the program to run in a particular context, to characterizing general principles of system behavior. The predictive model underlying the hypothesis testing undertaken by our programmers is specific to the extreme.

Empirical generalization progresses from description of particular instances to prediction (via models) of classes of systems. This progression is advanced to an explanation that connects what is going on in particular observed cases to a general class, in this case of certain kinds of software systems in certain kinds of configurations. The models and the kinds of systems they describe should both be broadly construed to render a true understanding of these systems. Are such general predictive models even possible for software systems? Experience indicates that when such systems are examined experimentally, with an explicit goal of discovering causal models, such models can in fact be found and highlight deep structure.

The articulation of scientific theories as predictive causal models, the methodology of empirical generalization, and the evaluation of such theories via hypothesis testing is prevalent in isolated sub-disciplines of computer science, including human-computer interaction, empirical software engineering, Web science, and data mining. It is, however, rarely used in other sub-disciplines of computer science, especially those concerned with software systems artifacts, such as compilers, databases, networks, operating systems, and programming languages.

**Examples of Computational Models**

One example of a computational model that is the product of empirical generalization is the *Theory of Locality.* This model arose from a study of the cost of managing page transfers between main memory and a much slower disk drive, within the general area of operating systems. What Dennett found was that data relevant to the current context of a running program tended to be grouped in space and time to local chunks, and this was the product of the way programs are written by humans, rather than due to any underlying constraints of the computing system. When our memory management algorithms respect this general pattern, performance improves dramatically. The resulting theory—that human-constructed information processing systems will exhibit locality—is inherently predictive and has been tested many, many times. This theory has subsequently been generalized to apply to computational systems of all kinds: “in virtual memory to organize caches for address translation and to design the replace-
Good scientific models will generate new questions to be answered and will drive our field to yet deeper understanding.

a threshold value of certain parameters. Subsequent studies have since systematically explored this phenomenon, and have discovered that where these transitions occur may depend on a variety of additional factors, including the kind of problem-solving method used. Different problem-solving methods provide us with different tools with which to study the complexity of these problems; an apt analogy has been made to astronomers using telescopes that operate on different light wavelengths to provide different perspectives on the structure of the cosmos. The jury is still out on how to best characterize the transition behavior, but the key point for our purposes is to again focus on the methodology employed: these researchers conduct experiments, take measurements, and refine their models, while striving for generality.

Realizing the Benefits of More Science in CS

These examples and other extant computer science theories emphasize that by embracing the methodology of developing and evaluating predictive models through experimentation over multiple members of a class of software systems, a more complete understanding of such artifacts will emerge. In addition, this observational and experimental scientific perspective encourages the computer scientist to actively look for relevant phenomena that may have been missed because they aren’t currently described by existing, closed-form analytical descriptions. The developed explanatory model provides better prediction and ultimately more dependable products built on those principles. Finally, good scientific models will generate new questions to be answered and will drive our field to yet deeper understanding.

How can these benefits be realized? How might we change what we do? We can adapt our already very skilled hypothesis testing in debugging and broaden it by asking more general questions, identifying the classes of system properties that contribute to behavior, identifying their boundary conditions, and working to fit them into a unified picture. We can also more broadly adopt additional methodological tools, such as from statistics and dynamic systems; a number of computer science subfields already profitably do so.

The pristine presentations of scientific reasoning and the tremendous successes of such reasoning in other fields may appear to the practicing computer scientist as out of reach. But many of our colleagues have started down this path, the tools are accessible, and the promise is great.

Acknowledgments

We thank the reviewers and editors for an enjoyable and vigorous conversation around the topic of this column. The reviewers will not agree with every detail, but they have certainly helped us refine the presentation.

References


Clayton T. Morrison (clayton@cs.arizona.edu) is a research assistant professor of computer science at the University of Arizona. He has more than a decade of experience building artificial intelligence systems for DARPA, AFOSR and NSF projects.

Richard T. Snodgrass (rts@cs.arizona.edu) is a professor of computer science at the University of Arizona. He is an ACM Fellow, has chaired the ACM Publications Board and SIGMOD, and has served as the editor-in-chief of ACM Transactions on Database Systems.

Copyright held by author.
The Software Industry is the Problem

ONE SCORE AND SEVEN YEARS AGO, Ken Thompson brought forth a new problem, conceived by thinking, and dedicated to the proposition that those who trusted computers were in deep trouble.

I am, of course, talking about Thompson’s 1984 ACM A.M. Turing Award Lecture—“Reflections on Trusting Trust.” Unless you remember this piece by heart, you might want to take a moment to read it if at all possible (http://bit.ly/nNGh5b).

The one sentence in Thompson’s lecture that really, really matters is: “You can’t trust code that you did not totally create yourself.”

This statement is not a matter of politics, opinion, taste, or in any other way a value judgment; it is a fundamental law of nature, which follows directly from pure mathematics in the general vicinity of the works of Turing and Gödel. If you doubt this, please (at your convenience) read Douglas Hofstadter’s classic Gödel, Escher, Bach, and when you get to the part about “Mr. Crab’s record player,” substitute “Mr. Crab’s laptop.”

Gödel, Escher, Bach

Hofstadter’s book, originally published in 1979, does not in any way detract from Ken Thompson’s fame, if, indeed, his lecture was inspired by it;
1979 was a long time ago, and it’s possible that not every reader may know of—much less have read—this book. My editor proposed that I summarize or quote from it to make things clearer for such readers.

Considering that Gödel, Escher, and Bach are all known for their intricate multilayered works and that Hofstadter’s book is a well-mixed stew not only of their works, but also of the works of Cantor, Church, Gantôr, Turing, and pretty much any other mathematician or philosopher you care to mention, I will not attempt a summary beyond: “It’s a book about how we think.”

The relevant aspect of the book here is Gödel’s incompleteness theorem, which, broadly speaking, says that no finite mathematical system can resolve, definitively, the truth value of all possible mathematical conjectures expressible in that same
In the book this is illustrated with a fable about Mr. Crab’s “perfect record player,” which, because it can play any and all sounds, can also play sounds that make it resonate and self-destroy—a vulnerability exploited on the carefully constructed records of Mr. Crab’s adversary, Mr. Tortoise.

Mr. Crab tries to protect against this attack by preanalyzing records and rearranging the record player to avoid any vulnerable resonance frequencies, but Mr. Tortoise just crafts the sounds on his records to the resonance frequencies of the part of the record player responsible for the rearrangement. This leaves Mr. Crab no alternative but to restrict his record playing to only his own, preapproved records, thereby severely limiting the utility of his record player.

Malware-scanning programs try to classify executable code into “safe” and “unsafe,” instead of mathematical conjectures into “true” and “false,” but the situation and result are the same: there invariably is a third pile called “cannot decide either way,” and whatever ends up in that pile is either a security or a productivity risk for the computer user.

Amusingly, malware scanners almost unfailingly classify malware-scanner programs, including themselves, as malware, and therefore contain explicit exemptions to suppress these “false” positives. These exemptions are of course exploitable by malware—which means the classification of malware scanners as malware was correct to begin with. “Quis custodiet ipsos custodes?” (Who will guard the guards themselves?)

**Back to Thompson**

In 1984, the Thompson lecture evoked wry grins and minor sweating for Unix system administrators at universities, because those were the only places where computers were exposed to hostile users who were allowed to compile their own programs. Apart from sporadic and mostly humorous implementations, however, no apocalyptic horsemen materialized in the sky.

In recent years, there have been a number of documented instances where open source projects were broken into and their source code modified to add backdoors. As far as I am aware, none of these attacks so far has reached further than the lowest rung on Ken Thompson’s attack ladder in the form of a hardcoded backdoor, clearly visible in the source code. Considering the value to criminals, however, it is only a matter of time before more advanced attacks, along the line Thompson laid out, will be attempted.

The security situation with commercial closed-source software is anyone’s guess, but there is no reason to think—and no credible factual basis for a claim—that the situation is any different or any better than it is for open source projects.

The now-legendary Stuxnet malware incident has seriously raised the bar for just how sophisticated attacks can be. The idea that a widely deployed implementation of Java is compiled with a compromised compiler is perfectly reasonable. Outsourced software development does not make that scenario any less realistic, likely, or scary.

**We Have to Do Something, But What?**

We have to do something that actually works, as opposed to accepting a security circus in the form of virus or malware scanners and other mathematically proven insufficient and inefficient efforts. We are approaching the point where people and organizations are falling back to pen and paper for keeping important secrets, because they no longer trust their computers to keep them safe.

Ken Thompson’s statement—“You can’t trust code that you did not totally create yourself”—points out a harsh and inescapable reality. Just as we don’t expect people to build their own cars, mobile phones, or homes, we cannot expect secretaries to create their own text-processing programs nor accountants to create their own accounting systems and spreadsheet software. In strict mathematical terms, you cannot trust a house you did not totally create yourself—but in reality, most of us will trust a house built by a suitably skilled professional. Usually we trust it more than the one we might have built ourselves—even when we may have never met the builder and/or when the builder is
dead. The reason for this trust is that shoddy construction has had negative consequences for builders for more than 3,700 years. “If a builder builds a house for someone, and does not construct it properly, and the house which he built falls in and kills its owner, then the builder shall be put to death.” (Hammurabi’s Code, approx. 1700 BC)

Today the operant legal concept is “product liability,” and the fundamental formula is “if you make money selling something, you’d better do it properly, or you will be held responsible for the trouble it causes.” I want to point out, however, that there are implementations of product liability other than those in force in the U.S. For example, if you burn yourself on hot coffee in Denmark, you burn yourself on hot coffee. You do not become a millionaire or necessitate signs pointing out that the coffee is hot.

Some say the only two products not covered by product liability today are religion and software. For software that has to end; otherwise, we will never get a handle on the security madness unfolding before our eyes almost daily in increasingly dramatic headlines. The question is how to introduce product liability, because just imposing it would instantly shut down any and all software houses with just a hint of a risk management function on their organizational charts.

**A Software Liability Law**

My straw-man proposal for a software liability law has three clauses:

**Clause 0. Consult criminal code to see if any intentionally caused damage is already covered.** I am trying to impose a civil liability only for unintentionally caused damage, whether a result of sloppy coding, insufficient testing, cost cutting, incomplete documentation, or just plain incompetence. Intentionally inflicted damage is a criminal matter, and most countries already have laws on the books for this.

**Clause 1. If you deliver software with complete and buildable source code and a license that allows disabling any functionality or code by the licensee, then your liability is limited to a refund.** This clause addresses how to avoid liability: license your users to inspect and chop off any and all bits of your software they do not trust or do not want to run, and make it practical for them to do so.

The word **disabling** is chosen very carefully. This clause grants no permission to change or modify how the program works, only to disable the parts of it that the licensee does not want. There is also no requirement that the licensee actually look at the source code, only that it was received.

All other copyrights are still yours to control, and your license can contain any language and restriction you care to include, leaving the situation unchanged with respect to hardware locking, confidentiality, secrets, software piracy, magic numbers, and so on. Free and open source software is obviously covered by this clause, and it does not change its legal situation in any way.

**Clause 2. In any other case, you are liable for whatever damage your software causes when used normally.** If you do not want to accept the information sharing in Clause 1, you would fall under Clause 2 and have to live with normal product liability, just as manufacturers of cars, blenders, chainsaws, and hot coffee do. How dire the consequences and what constitutes “used normally” are for the legislature and courts to decide.

An example: A salesperson from one of your longtime vendors visits and delivers new product documentation on a USB key. You plug the USB key into your computer and copy the files onto the computer. This is “used normally” and should never cause your computer to become part of a botnet, transmit your credit card number to Elbonia, or send all your design documents to the vendor.

The majority of today’s commercial software would fall under Clause 2. To give software houses a reasonable chance to clean up their acts and/or to fall under Clause 1, a sunrise period would make sense, but it should be no longer than five years, as the laws would be aimed at solving a serious computer security problem.

And that is it, really. Software houses will deliver quality and back it up with product liability guarantees, or their customers will endeavor to protect themselves.

**Would it Work?**

There is little doubt that my proposal would increase software quality and computer security in the long run, which is exactly what the current situation calls for.

It is also pretty certain there will be some short-term nasty surprises when badly written source code gets a wider audience. When that happens, it is important to remember that today the good guys have neither the technical nor legal ability to know if they should even be worried, as the only people with source-code access are the software houses and the criminals.

The software houses would yell bloody murder if any legislator were to introduce a bill proposing these stipulations, and any pundit and lobbyist they could afford would spew their dire predictions that “this law will mean the end of computing as we all know it!”

To which my considered answer would be: “Yes, please! That was exactly the idea.”

---

**Related articles on queue.acm.org**

CTO Roundtable: Malware Defense
http://queue.acm.org/detail.cfm?id=1731902

All Things Being Equal?
Stan Kelly-Bootle
http://queue.acm.org/detail.cfm?id=1348596

B.Y.O.C. (1,342 Times and Counting)
Poul-Henning Kamp
http://queue.acm.org/detail.cfm?id=1944489

**References**


Poul-Henning Kamp (phk@FreeBSD.org) has programmed computers for 26 years and is the inspiration behind bikeshed.org. His software has been widely adopted as “under the hood” building blocks in both open source and commercial products. His most recent project is the Varnish HTTP accelerator, which is used to speed up large web sites such as Facebook.