

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 Holz-

mann 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 because 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

“How can you ever hope to build a dependable system if you don't know what ‘dependable’ means?” asks MIT's Daniel Jackson.

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

Awards

2007 ACM Award Winners

Outstanding Contribution to ACM Award
Robert A. Walker,
Kent State University

Distinguished Service Award
David A. Patterson, University
of California at Berkeley

Eugene L. Lawler Award for Humanitarian Contributions within Computer Science and Informatics
Randy Wang,
Microsoft Research India

Paris Kanellakis Theory and Practice Award
Bruno Buchberger,
Johannes Kepler University

Karl V. Karlstrom Outstanding Educator Award
Randy Pausch,
Carnegie Mellon University

Grace Murray Hopper Award
Vern Paxson, International
Computer Science Institute
and University of California at
Berkeley/Lawrence Berkeley
National Laboratory

A.M. Turing Award
Edmund M. Clarke,
Carnegie Mellon University
E. Allen Emerson,
University of Texas at Austin
Joseph Sifakis, Centre National
de la Recherche Scientifique
and Verimag Laboratory

Software System Award
David Harel, The Weizmann
Institute of Science
Hagi Lachover
Amnon Naamad, EMC Corporation
Amir Pnueli, NYU Courant
Institute of Mathematical Sciences
Michal Politi,
Tadiran Electronic Systems
Rivi Sherman, Negevtech
Mark Trakhtenbrot,
Holon Academic Institute of
Technology and The Open
University of Israel
Aron Trauring, Zotecabv

ACM – Infosys Foundation Award
Daphne Koller, Stanford University

Doctoral Dissertation Award
Sergey Yekhanin, Princeton University

Honorable Mentions:
Benny Applebaum,
Princeton University
Vincent Conitzer, Duke University
Yan Liu, IBM

ACM-W Athena Lecturer Award
Shafi Goldwasser, MIT and The
Weizmann Institute of Science
ACM – AAAI Allen Newell Award
Leonidas J. Guibas, Stanford University

Several award winners had yet to be announced at press time. We'll have more news about the 2007 ACM award winners in next month's issue.

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,

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

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, setjmp or longjmp 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.” ■

Leah Hoffman is a Brooklyn-based freelance writer.

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 Branff, 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.”

Coming Next Month in COMMUNICATIONS

Scaling Massive Multiplayer Online Game Infrastructure

Techniques for Designing Games with a Purpose

Computer Science and Game Theory

The Rise and Fall of CORBA

Evaluating Methodology for the 21st Century

Composable Memory Transactions

Envisioning the Future of Computing Research

CTO Roundtable

Part II of an interview with Donald Knuth

And the latest news about game theory, assistive technologies, and computing and the developing world.