Grand Challenges Conference Application

Jim Horning (work@horning.net)
NAI Labs and CRA board member

Proposal:

Why are computers so interesting and so important? Because they make it possible to do interesting and important stuff that we couldn't do otherwise. It's not so much that they make things cheaper, it's that they make things possible. And because of Moore's Law more things are becoming possible all the time.

I still vividly remember writing my first computer program, 42 years ago. I wrote it to do a computation that I never would have contemplated doing by hand. What immediately hooked me on computers was the feeling of absolute control--here was a device that would do exactly what I told it! (Unlike people, and unlike the machines I was used to.) Of course, I quickly learned that it wasn't always easy to tell the computer exactly what I really wanted it to do. I remember a jingle from that time:

I really hate this damn machine,
I wish that they would sell it.
It never does what I mean,
But only what I tell it.

But we computerniks reveled in our ability to get the computer to do what we wanted--if not on first try, then after a moderate amount of debugging. Crashes and other deviations from desired behavior were individually investigated. Sometimes it was a hardware problem that would be identified using machine diagnostics and then repaired--vacuum tubes weren't as reliable as integrated circuits are. But more often it was an identifiable bug in the program, something that we would fix--and then test the fix. This did not shake our confidence that eventually we would get the program right, and then the machine would reliably do what we wanted.

In many ways, we've made enormous progress since those days. Machines are vastly faster, more powerful, smaller, and more plentiful--and cheaper, too. Millions of people with little or no training or understanding of how computers work use computers to do things we didn't even dream of back then. High-level languages, database systems, graphical user interfaces, visual application builders, and many other intellectual amplifiers enable us to build systems of ever-greater functionality and scale.

But along the way, we have lost our confidence that any one of us can surely get a computer to always Do the Right Thing, if we just work hard enough at it.

I no longer have confidence in the trustworthiness of the computers I rely on daily. I no longer expect every crash to be investigated. It no longer surprises me that customer support's first suggestion is to reboot the machine--or that rebooting without investigation is frequently the most cost-effective thing to do. I don't blame any single supplier for this loss of confidence, because every vendor in the industry leans over backward not to give

any warrantee or assurance that what they sell will operate correctly and not cause damage.

What has happened? At least two things:

1. The exponential increase in hardware power described by Moore's Law has led to a corresponding increase in the complexity of what we do with computers. However, there has not been a corresponding increase in our ability to master that complexity. No one understands the whole of any system anymore--not the hardware, not the operating system, not the network, not even individual applications. When something goes wrong, we can often only speculate which component--or interaction of components--caused the problem.

2. We have let software vendors get away with it. Market forces drove Intel to recall Pentium chips that occasionally made small errors in division. When was the last similar software recall? And why should vendors invest in quality that customers don't demand? It's features that sell software! As I read in ACM's "Ubiquity" last summer, "Software managers and developers give lip service to quality while customers grow accustomed to buggy software."

Here is a conundrum: Hardware reliability seems to be increasing, despite the exponential growth in chip complexity, and the increasing size of the teams designing new chips. Yet software reliability seems to be decreasing. What is the difference? Do hardware engineers do a better job:

- of partitioning their systems?
- of specifying their interfaces?
- of designing for reliability and testability?
- of verifying and testing their designs?

Or does the extra expense of dealing with hardware errors just make everybody more cautious?

I believe that one important difference lies in interface specification. Generally, only hardware interfaces are sufficiently well documented that components can be rigorously tested against specifications. Hardware engineers have a long history of careful interface specification, dating back to the time of discrete components, based on the notion that components should be interchangeable. The interface that the hardware presents to the software is carefully specified and slow to change. Put a new CPU chip in your system, and all the legacy software had better still run; install new software and all bets are off. There's too much truth in the quip that "Hardware is the part of the system that you can change easily, software is the part that you can't."

I see three major challenges in the area of software interface specification:

1. Writing specifications that software developers can use--specifications that:
   - precisely define all the necessary properties of the software,
   - do not unnecessarily constrain the implementation,

- can be used to verify the implementation, and
- are readily understood by implementers.

2. Writing specifications that capture what it is that users want, and do so in a way that they can understand them:
   - to verify in advance that the system being designed is the system wanted, and
   - to precisely understand the behavior of the specified system--to know what to expect, and how to accomplish what they want to do.

3. Persuading the parties involved that precise interface specification is worth the cost. This may well be the hardest problem of the three.

I'm not aware of any attempt to quantify the economic costs of software unreliability and incomprehensibility. But they must be enormous. The massive efforts invested in checking software for the well-publicized Y2K problem and fixing it as necessary were just the tip of a tip of this iceberg. The time lost to rebooting, to recovering lost data, to repairing hardware that falls victim to computer rage, etc., tends not to show up in the accounts anywhere--let alone the costs of recovering from incorrect results produced by computers. But these are real costs, distributed throughout most of the economy. I've heard it seriously suggested that computerization has been a net drain on the American economy over the last half-century. But computers enable us to do interesting and important stuff that we couldn't do otherwise, so refusing to use them simply isn't an option.

The most important challenge for Computer Science and Engineering is figuring out how we can make computers do what we want them to do--and then actually making them do it.

BIO

Jim Horning wrote his first computer program in 1959 and was immediately hooked by its power and by a feeling of absolute control (which has steadily faded in the last few decades). Jim has been working with computing ever since, mostly in research settings. His research has ranged from operating systems to formal specification, from AI to programming languages. His long-term research interest is "the mastery of complexity"--attempts to bring computing back under human comprehension and control.

Jim received his Ph.D. in Computer Science from Stanford University in 1969, following a M.S. in Physics from UCLA and a B.A. in Physics and Mathematics from Pacific Union College.

Jim was a founding member and Chairman of the University of Toronto's Computer Systems Research Group (CSRG), a Research Fellow at Xerox's Palo Alto Research Center (PARC), a founding member and Senior Consultant at Digital's Systems Research Center (DEC/SRC), and the founder and Director of InterTrust's Strategic Technologies and Architectural Research Laboratory (STAR Lab). He is currently manager of the Security Architecture andModeling group at Network Associates Laboratories.

Jim is a co-author of two books, "Larch: Languages and Tools for Formal Specification" (1993) and "A Compiler Generator" (1970), and author or co-author of chapters in nine other books. He is co-inventor on two U.S. patents (4,164,107 and 5,940,619), with other patents pending.

Jim has been a member IFIP's Working Group 2.3 (Programming Methodology) since 1974 and is a past Chairman. He is currently participating in the NRC/CSTB study committee on Privacy in the Information Age. He is a member of the Board of CRA and a Fellow of the ACM.