DOI:10.1145/2398356.2398359

Computer Science Is Not a Science

To the question Vinton G. Cerf addressed in his President's Letter "Where Is the Science in Computer Science?" (Oct. 2012), my first answer would be that there isn't any. Max Goldstein, a mentor of mine at New York University, once observed that anything with "science" in its name isn't really a science, whether social, political, or computer. A true science like physics or chemistry studies some aspect of physical reality. It is not concerned with how to build things; that is the province of engineering. Some parts of computer science lie within mathematics, but mathematics is not a science and is rarely claimed to be one.

What we mislabel as computer science would more aptly be named "computology"—the study of computational processes and the means by which they can be realized. Its components can broadly be grouped into three interdependent areas: software engineering, hardware engineering, and the mathematics of computation. Just as the underlying discipline of chemical engineering is chemistry, the underlying discipline of software engineering is mathematics.

But not so fast. To qualify as a subject of science, a domain of inquiry needs two qualities: regularity and physicality. Reproducible experiments are at the heart of the scientific method. Without regularity they are impossible; without physicality they are meaningless. Digital computers, which are really just very large and complicated finite-state machines, have both these qualities. But digital computers are artifacts, not part of the natural world. One could argue either way whether that should disqualify them as a subject of science. Quantum computing and race conditions complicate the picture but not in a fundamental way.

None of this detracts from Cerf's essential point—that when we design software we rarely understand the full implications of our designs. As he said, it is the responsibility of the computing community, of which ACM is a vital part, to develop tools and explore principles that further that understanding and enhance our ability to predict the behavior of the systems we build.

Paul W. Abrahams, Deerfield, MA

In his President's Letter (Oct. 2012), Vinton G. Cerf wrote: "We have a responsibility to pursue the science in computer science [...and to develop] a far greater ability to make predictions about the behavior of these complex, connected, and interacting systems." This is indeed a worthwhile cause that would likely increase the reliability and trustworthiness of the whole field of computing. But, having picked up the gauntlet Cerf threw down, how do I make that cause fit the aspects of computer science I pursue every day?

Cerf discussed the problems software developers confront predicting the behavior of both software systems and the system of people developing them. As a professional developer, I have firsthand experience. Publishing a catalog of the issues I find might lead analysts to identify general problems and suggest mitigations would be subject to two limitations: probably not interesting enough for journal editors to want to publish and my employers likely viewing its content as commercially sensitive.

I could instead turn to the ACM Digital Library and similar resources, looking for ways to apply it to my professional work. However, this also has limitations; reading journal articles is a specialized, time-consuming art, and the guidance I would need to understand what and how results are relevant is often not available. Many of the "classic results" quoted by professionals turn out to be as verifiable as leprechaun sightings.[1]

To the extent the creation of software can be seen as "computer science," such creation is today two distinct fields: creating software and researching ways software can be created. If we would accept the responsibility Cerf has bestowed upon us, we would have to create an interface discipline—call it "computer science communication"—between these fields.

Graham Lee, Leicester, U.K.

Reference

1. Bossavit, L. *The Leprechauns of Software Engineering.* Leanpub, Vancouver, B.C., 2012; https://leanpub.com/leprechauns

Only Portfolios Mitigate Risk Well

Peter G. Neumann's "Inside Risks" Viewpoint "The Foresight Saga, Redux" (Oct. 2012) addressed how to provide security but fell short. Though security requires long-term approaches and research advances, traditional incentives target quick rewards. I teach a graduate course on IT strategy and policy largely focused on this dilemma. When technology moved slowly, slow acquisition and delayed delivery caused minor losses. Now, however, along with improvement due to technology innovation, delays in exploiting advanced technology incur exponentially increased opportunity costs. Most businesses cannot wait for high-trust solutions or systems that significantly surpass state-of-the-art quality. Likewise, most government systems are already too costly and too late, in part because they try to address an unreasonably large number of requirements.

The risk-management problem necessitates a portfolio-management approach. In the context of IT systems for business or government, it would be more affordable and practical to create multiple alternatives and fallback options and not depend on a single system where failure would be devastating. In addition, applications should be separated from research and funded appropriately. It would be great to have a secure Internet, unbreakable systems, and uniformly trained people, but such goals are not practical today. The focus should instead be on risk mitigation, resilience, and adaptation, even though the incentives for moving quickly are often irresistible. "Ideal" systems are indeed the enemy of practical portfolios built to withstand a range of risks.

Rick Hayes-Roth, Monterey, CA

Clock-Free Computing

As an undergrad at MIT in 1972, I took a course in asynchronous design from Prof. Jonathan Allen. Having some background at the time in digital circuitry, it was exciting to see this latest work as presented by Allen, and it was easy to imagine that in a few years most computers and other digital systems would operate this way. The reasoning was much like what Ivan Sutherland advocated in his Viewpoint "The Tyranny of the Clock" (Oct. 2012). Following graduation I started out in the working world designing digital hardware. Industry opens a student's eyes to the real world, and it was clear rather quickly that the synchronous world would not in fact budge for a long time. Though my work today involves mostly software, I still see the appeal of asynchronous logic and hope the vision of asynchronous computing finally takes hold. We could use more calls-to-arms like Sutherland's: "The clock-free design paradigm must eventually prevail." I look forward to that day, just as I look forward to another paradigm that should eventually prevail—parallel processing.

Larry Stabile, Cambridge, MA

Relational Model Obsolete

I write to support and expand on Erik Meijer's article "All Your Database Are Belong to Us" (Sept. 2012). Relational databases have been very useful in practice but are increasingly an obstacle to progress due to several limitations:

Inexpressiveness. Relational algebra cannot conveniently express negation or disjunction, much less the generalization/specialization connective required for ontologies;

Inconsistency non-robustness. Inconsistency robustness is information-system performance in the face of continually pervasive inconsistencies, a shift from the once-dominant paradigms of inconsistency denial and inconsistency elimination attempting to sweep inconsistencies under the rug. In practice, it is impossible to meet the requirement of the Relational Model that all information be consistent, but the Relational Model does not process inconsistent information correctly. Attempting to use transactions to remove contradictions from, say, relational medical information is tantamount to a distributed-denial-of-service attack due to the locking required to prevent new inconsistencies even as contradictions are being removed in the presence of interdependencies;

Information loss and lack of provenance. Once information is known, it should be known thereafter. All information stored or derived should have provenance; and

Inadequate performance and modularity. SQL lacks performance because it has parallelism but no concurrency abstraction. Needed are languages based on the Actor Model (http://www.robust11.org) to achieve performance, operational expressiveness, and inconsistency robustness. To promote modularity, a programming language type should be an interface that does not name its implementations contra to SQL, which requires taking dependencies on internals.

There is no practical way to repair the Relational Model to remove these limitations. Information processing and storage in computers should apply inconsistency-robust theories[1] processed using the Actor Model[2] in order to use argumentation about known contradictions using inconsistency-robust reasoning that does not make mistakes due to the assumption of consistency.

This way, expressivity, modularity, robustness, reliability, and performance beyond that of the obsolete Relational Model can be achieved because computing has changed dramatically both in scale and form in the four decades since its development. As a first step, a vibrant community, with its own international scientific society, the International Society for Inconsistency Robustness (http://www.isir.ws), conducted a refereed international symposium at Stanford University in 2011 (http://www.robust11.org); a call for participation is open for the next symposium in the summer of 2014 (http://www.ir14.org).

Carl Hewitt, Palo Alto, CA

References

1. Hewitt, C. Health information systems technologies. Stanford University CS Colloquium EE380, June 6, 2012; http://HIST.carlhewitt.info
2. Hewitt, C., Meijer, E., and Szyperski, C. *The Actor Model.* Microsoft Channel 9 Videos; http://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask

Design Software for the Unknown

In his article "Software Needs Seatbelts and Airbags" (Sept. 2012), Emery D. Berger identified typical flaws in coding, as well as techniques that might help prevent them, addressing a major conundrum taking much of a typical programmer's time: Much more time goes for tracking bugs than for writing useful programs.

Berger's analogy of software techniques and automobile accessories was illuminating, though computer technology has generally outpaced the automobile by orders of magnitude. Some have said automobiles could go one million miles on a single gallon of fuel, reaching its destination in one second, if automobile engineers were only as bright as computer scientists. Others have said we would be driving $25 cars that get 1,000 miles to a gallon if they were only designed by computer scientists instead of by automobile engineers. But the analogy should not be stretched too far. An advocate of software reliability might say seatbelts and bumpers are not intended to protect drivers from design errors but from their own errors, or bad driving; in software the analogous problem is designer error. If defects are discovered, cars are recalled and defective parts replaced. Some software products that update themselves multiple times a day crash anyway because analogous seatbelts and airbags in software are a luxury.

The defects Berger covered are more analogous to bad plumbing and crossed wires. Moreover, software developers may not even know all the components and functions in the software they deliver. Though perfect in terms of memory handling and buffer-overflow management, software can become a work of art during development, with no way to completely anticipate how it will perform under unknown circumstances.

I would like to see researchers of software code take a look at something I call "mind of software," aiming for ways to make software more safe and predictable for common use.

Basudeb Gupta, Kolkata, India

Communications welcomes your opinion. To submit a Letter to the Editor, please limit yourself to 500 words or less, and send to letters@cacm.acm.org.