The Concorde Doesn't Fly Anymore

Moti Ben-Ari

Keynote Talk, SIGCSE 2005, St. Louis, MO

I am genuinely excited and grateful that the SIGCSE Board has seen fit to give me this award. When you work in education, you don't often see spectacular results like you do when a startup company succeeds in an IPO. It is said that an educator never knows the true extent of his or her influence, which can continue for years or even generations, but it is gratifying to receive some recognition from one's own generation. I would also like to give my heartfelt thanks to my students, both in my department at the Weizmann Institute of Science and at the University of Joensuu in Finland, where I spent a sabbatical. The students that I have been fortunate to supervise have immensely amplified the scope of the work that I have been able to do.

I'm sure you're all wondering what the relevance of the Concorde could be for CS education; I promise to talk about that, but first I would like to reminisce about my involvement in my favorite CS topic, concurrent programming. Once upon a time, the employment outlook for all but the best mathematicians was bleak, but by 1970 when I graduated, the field of software development offered tempting opportunities. Several years later, while working as a systems programmer at the Tel Aviv University, I decided to return to graduate school for a master's degree; although a Department of Computer Science did not yet exist at the university, you could get advanced degrees in mathematics by studying computer science. Since my bachelor's degree was in mathematics, I was duly served up with a list of about six courses that I would have to pass in order to officially become a student of computer science. Shortly before the start of the school year, however, I was approached and asked—on the basis of my experience as a systems programmer—to teach the course on operating systems. I don't recall the details, but I'm sure that they agreed that if I taught the course, I could be excused from studying it.

So how was I supposed to teach a course in operating systems? The theoretical part was not too problematical. Fortunately, Per Brinch Hansen had already published his classic *Operating Systems Principles*, and even more fortunately, I found Edsger Dijkstra's paper *Cooperating Sequential Processes*. But I believe that in computer science, theory and practice must go together. As a systems programmer, I had developed a reasonably good knowledge of the structure of the specific OS for the university's only computer, a Control Data 6600. You could literally flush a buffer, because it had a fluid cooling system. The main hard disk had a hydraulic drive, so the operators checked the oil and water at regular intervals. I knew from experience that it was totally impossible to let students experiment with the system.

At that time I had managed to get the computer center to cough up 160 Swiss francs to order the magnetic tape containing the first Pascal compiler which fortuitously had been developed for the CDC computer. After years of writing Fortran and assembly language, my mathematical soul excitedly welcomed the Pascal language as a breath of fresh air. The tape also included the Pascal-S interpreter: a short program written in Pascal that was composed of a recursive descent compiler from a subset of Pascal into P-Code, a byte code for a virtual machine, and an interpreter for that P-Code. I began to see that a few "small" modifications of this well-written program could turn it into an interpreter for concurrent programs. My students could now freely crash the interpreter without bringing upon my head the wrath of the

university officials, especially, the all-important administrative data processing department who used the same mainframe as the faculty and students.

The second serendipitous moment came in the late 1970's when my thesis advisor went overseas for the summer, leaving me alone with the oppressive heat in those days before ubiquitous air conditioners, and alone with the weight of the theory of my thesis research. The Department of Mathematics had just received its own computer, a PDP-11! The department head, Dan Amir, a professor of pure mathematics, remarked: "I fought to obtain it, but I don't know what to do with it." I decided that this would be a good opportunity to type up the notes from my course, including the code for the concurrency simulator. Things went smoothly for a couple of weeks, until the brilliant student employed as the systems programmer approached me with some trepidation and told me that the system had asked if he would like to "mkfs," make a new file system, and he had answered "yes." Since then, I keep five to six backup copies of all my manuscripts!

Eventually, I completed the manuscript and although I was a mere graduate student, I send it off to several publishers. I was more than surprised and more than pleased to receive a letter from Prentice-Hall International saying that Tony Hoare liked it. The igloo book appeared shortly after I received my doctorate. Since imitation is the sincerest form of flattery, I'm quite flattered by the stream of concurrency books and simulators that have appeared since then.

I came back to the world of concurrency six years ago, when, together with my student (now Dr.) Yifat Ben-David Kolikant, we developed a course on concurrent and distributed computation for high school students. It has been nearly 40 years since I wrote my first computer program as a high school graduate, and concurrency is one of the few CS topics that I can still get excited about.

Recently I have been writing educational software tools in Java. Have you ever written GUI software? Of course you have dig through mountains of API documentation, helped perhaps by some useful tutorials and textbooks, in order to learn how to do this, but there is truly no intellectual challenge. It is boring, even though students like doing it because they immediately see a flashy display, and even though employers like you to teach it because they get trained technicians.

Let me compare this with an episode from daily life in a class on concurrency. I'm sure that you know that if you execute the following concurrent program, you need not get the answer 20 because of interleaving of the machine language instructions.

<table>
<tr><td colspan="2">var N: Integer := 0;</td></tr>
<tr><td>process P1;
var I: Integer;
begin
    for I := 1 to 10 do
        N := N + 1
end;</td><td>process P2;
var I: Integer;
begin
    for I := 1 to 10 do
        N := N + 1
end;</td></tr>
</table>

In fact with "perfect" interleaving—P1 loads N, P2 loads N, P1 adds 1, P2 adds 1, P1 stores N, P2 stores N—you get the answer 10.

Process	Instruction	Register P1	Register P2	N
P1	Load N	0	0	0
P2	Load N	0	0	0
P1	Increment	0	0	0
P2	Increment	1	0	0
P1	Store N	1	1	0
P2	Store N	1	1	1
		1	1	1

If, several years ago, you had asked me what values could possibly be computed by the program, I would have very confidently replied: "between 10 and 20." Well, one day I was present at a lab session in a high school class, where the students were studying this program. The young students were hitting the Reset and Go buttons of the concurrency simulator with the speed and agility of arcade game players. Suddenly, one of them called us over: the program printed the value 9. Now this was highly embarrassing, as the concurrency simulator had been successfully used for many years, and though I had in the meantime destroyed its structure and maintainability through incessant modification, these modifications were never made to the basic compilation or interpretation of existing statements and expressions. Several hours of feverish work enabled me to see that there were obscure interleavings that could produce the answer 9, and even, the answer 2! All my dogmatism that the answer must be between 10 and 20 went down the drain. As a curiosity, I checked with Alan Burns of the University of York, who had written a competing textbook and concurrency simulator; he mentioned that a similar episode had happened to him. As a homework assignment, I leave it to you to find the extreme scenario by yourselves. OK, I really shouldn't assign homework on this festive occasion, so you can look up the answer in our published note: M. Ben-Ari and A. Burns. Extreme interleavings. *IEEE Concurrency* 6(3), 1998, 90.

Better yet, use Gerard Holzmann's Spin Model Checker to find a scenario. (Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004. `http://spinroot.com`.) I've become quite excited recently about Spin. On the one hand it is a truly professional tool used for verifying industrial software and for academic research in program verification. Spin received the 2001 ACM System Software Award. On the other hand it is simple enough for students to use, and I have developed a simple graphical front end called jSpin (`http://stwww/weizmann.ac.il/g-cs/benari/jspin`). If we execute a verification run with the assertion that N is greater than 2 at the end of the program, Spin notifies us that the assertion does not necessarily hold, and it also produces a counterexample. The scenario is very hard to find by hand, but easy to study using the output of Spin.

I promised to talk about the Concorde and I'm sure that you've been eagerly trying to find some connection with computer science education. While I am extremely pleased to have received the SIGCSE Award, let's face it, this is not the Nobel Prize and I'm not receiving a one-million dollar prize. And all of you know that an academic salary is hardly sufficient to allow me purchase a ticket to fly on the Concorde. However, even if I could afford it, I could not take a flight on the Concorde, for the simple reason that its last commercial flight took place on the 24th of October of 2003, just a few months before the Norfolk conference. The Concorde had flown experimentally since 1969 and commercially since

1976. The best you can do nowadays is to buy a first-class ticket on a boring old 747; coincidentally the the first flight of the 747 was also in 1969.

The Concorde and the Boeing 747 have two things in common. The first, and most obvious, is that these symbols of hi-tech were developed in the 1960s and 70s. The second, and less obvious, is that should you decide to achieve the same goals today, you would do so in almost precisely the same way that was done thirty or forty years ago. There is no really revolutionary technology in aerospace, not in the sense that there aren't any new ideas, but that for more than 30 years, there is no new technology that has radically changed the way we accomplish goals in this field.

I would like to speculate that the situation is now similar in computer science and computing technology. From reading press releases, you would think that we experience true revolutions every day. I used to read profiles of startup firms in the newspaper, and invariably, the description of the activities of the startups included the phrase: "X has developed a technology to Y." As far as I could tell, what they did was to write a program, which is hardly a new technology or a true revolution. One gets the impression that before the mid 1990s, nobody wrote any programs, or at least that they wrote only simple low-tech programs.

Is that all we did? Did we only write low-tech programs?

Here is a quote from NASA's web site

> On July 20, 1969, the human race accomplished its single greatest technological achievement of all time when a human first set foot on another celestial body. Six hours after landing at 4:17 p.m. Eastern Daylight Time (with less than 30 seconds of fuel remaining), Neil A. Armstrong took the "Small Step" into our greater future when he stepped off the Lunar Module, named "Eagle," onto the surface of the Moon, from which he could look up and see Earth in the heavens as no one had done before him. (`http://history.nasa.gov/ap11ann/introduction.htm`.)

Even if NASA's prose is a bit florid, I agree that this was the "single greatest technological achievement of all time." You need only consider what it must have been like to sit atop massive rockets and to entrust yourself to the intricate calculation of the celestial mechanics involved in performing the various rendezvous. The fact that six out of seven attempts to land on the moon were totally successful is nothing short of amazing.

So what did the Apollo computer system look like? (Source: James E. Tomayko, Computers in Spaceflight: The NASA Experience, 1988, `http://www.hq.nasa.gov/office/pao/History/computers/Compspace.html`.) The Apollo Guidance Computer was "fairly compact" and weighed only 70.1 pounds (32 kilograms)! The execution of an instruction required two machine cycles—about 24 milliseconds or 83kHz, and the computer had 36K 16-bit words of fixed memory and 2K words of erasable memory. That is not a mistake: K as in kilowords not M as in megawords. Just for comparison: three years ago when I returned from my sabbatical, I insisted that my boss buy me a Pentium 4 computer with 256 megabytes of memory; otherwise, it would have been impossible to run modern educational software. The fixed memory (core rope) was unchangeable once it left the factory, so the software had to be delivered months before a mission for manufacturing and electrical testing, placing a premium on

the production of quality software without hacking. And yet with this computer system, NASA carried out the "single greatest technological achievement of all time!" And this achievement terminated with the flight of Apollo 17 to the moon in 1972.

Let us now turn to computer science and attempt to inject a bit of historical perspective into our field, which has been deluged with hype during the past decade. It goes without saying that the Internet is the most popular subject in computer science, the collapse of the dot-com bubble notwithstanding. Therefore, I think it is worthwhile to review some of the technical milestones, limiting ourselves to the period more than two decades ago. (Source: Hobbes' Internet Timeline. © 1993-2005 by Robert H Zakon. `http://www.zakon.org/robert/internet/timeline`.)

- 1961 – The first paper on packet switching by Leonard Kleinrock.
- 1968 - BBN was awarded a contract to build Interface Message Processors. (Senator Edward Kennedy congratulated BBN on the ecumenical *Interfaith* Message Processor, which shows that you can't trust politicians to understand technology.)
- 1971 - Ray Tomlinson of BBN invented an email program.
- 1971 - Project Gutenberg was started by Michael Hart.
- 1972 - First computer-to-computer chat.
- 1973 - Bob Metcalfe outlined the basic idea for Ethernet.
- 1973 - The File Transfer Protocol and Network Voice Protocol were published.
- 1974 - Vint Cerf and Bob Kahn published the specification for the Transmission Control Protocol.
- 1975 - First ARPANET mailing list.
- 1976 - Queen Elizabeth sent an email.
- 1980 - ARPANET crashed because of a virus.

The point I am trying to make is that the Internet is not some revolutionary new technology. It was new and revolutionary, but that was before our students were born, and we should be very careful not to ignore the historical perspective.

To be fair, there is some justification for looking upon the Internet as new. Here is one personal example. In 1993-4, I tried to market a program that I had written; excuse me, I should say, a new technology that I had developed. But, in order to obtain a connection to the Internet, I was required to receive permission from the Ministry of Communications, because at that time the only connections were through an inter-university hub, and I had to prove that I was engaged in R&D! Of course, this anecdote merely shows that the obstacles to the spread of the Internet were political and economical, not technological.

Let me talk a bit more about visionaries and purveyors of hype. Here are some quotes:

> It is impossible that old prejudices and hostilities should longer exist, while such an instrument has been created for the exchange of thought between all the nations of the earth.
>
> [It] may not affect magazine literature, but the mere newspapers must submit to destiny, and go out of existence.

Do you remember the good old days when people worked nine-to-five, just five days a week?

> The merchant goes home after a day of hard work and excitement to a late dinner, trying amid the family circle to forget business, when he is interrupted by a [message] from London,..., and the poor man must dispatch his dinner as hurriedly as possible in order to send off his message to California. The business man of the present day must be continually on the jump[.]

These are clearly utopian visions that high-technology will bring us peace, brotherhood and the paperless office, though instant communication throughout time zones will cause extreme stress in the work environment. And here is a characterization of the monopolistic practices of a leading company:

> Not surprisingly, the company regarded its near monopoly as a good thing. Far from encouraging progress [the CEO] claimed, competition between rival companies had actively hindered it. [The company] insisted that its monopoly was in everyone's interest, even if it was unpopular, because it would encourage standardization.

Is the author talking about Bill Gates and Microsoft?

In fact, all the quotes are authentic mid-19th century commentary on the use of the telegraph taken from Thomas Standage. *The Victorian Internet: The Remarkable Story of the Telegraph and the Nineteenth Century's On-line Pioneers*. New York: Walker Publishing Company, 1998. The monopolist CEO was William Orton of Western Union. I highly recommend Standage's book as an antidote to rampant futurism. Here is how Standage characterizes the introduction of the telegraph:

> During Queen Victoria's reign, a new communications technology allowed people to communicate almost instantly across great distances. It revolutionized business practice, gave rise to new forms of crime, and inundated its users with a deluge of information. Romances blossomed, secret codes were devised and cracked. The benefits of the network were relentlessly hyped by its advocates and dismissed by the skeptics. Governments and regulators tried and failed to control the new medium. A technological subculture with its own customs and vocabulary was establishing itself. (p. vii)

Exactly the same paragraph could be written today, just changing Victoria to Elizabeth.

I believe that we have lost perspective in computer science. Each new system is trumpeted as a new technology, even though it may just be a repackaging of well-known technology. I feel quite certain that

from a historical perspective, the invention of the telegraph will come to be regarded as more significant than the invention of the Internet.

Let me give another example of the lack of historical perspective in computer technology. I believe that the design of Java is relatively good, and I use it extensively in teaching and for developing software tools. But I cannot join in the hype of Java as something new; it is simply a successful packaging of technologies that have been with us, not for years, but for decades:

- Compilation to a virtual machine goes back to Pascal-P in 1974.
- Object-oriented programming was invented for Simula in 1964.
- Support for concurrency within a programming language appeared in Concurrent Pascal in 1974.
- Smalltalk, first developed in 1969, included a large API.
- Even the cryptic syntax is taken from C which was developed in 1971.

Now I don't want you to become depressed by my litany of claims that there is not all that much that is revolutionary in computer science. This is not cause for pessimism, but rather a plea for maturity. I believe that as computer science educators we need to do two things: First, we need to structure education in our discipline on the model of other mature fields of science and engineering, and second, we must accept the responsibility that goes with maturity, and not offer excuses like juvenile delinquents. You know: "Computer science is a young field, so we cannot be expected to (whatever)."

So what exactly would education be like in the mature fields of computer science and computational engineering. (I have invented a new term for the discipline of designing, developing and maintaining computational systems, because the term software engineering is used for certain limited aspects of that engineering activity.)

Well, the Concorde may not fly anymore, but, believe it or not, people actually study to become mechanical and aeronautical engineers, even though they are limited to building 747 clones. My son Hezi recently completed his studies in mechanical engineering at the Technion, so I decided to investigate what he studied in that old-fashioned, low-tech discipline, and how it compares with what we teach in our revolutionary, hi-tech discipline.

First let us look at what we teach in computer science. Let us examine the new ACM/IEEE curriculum. I don't want to criticize CC2001 in and of itself; it is a massive and well-thought-out effort, to which I even contributed a few sentences. But CC2001 is a reflection of what CS education is, not necessarily what I think it should be. CC2001 divides the quote-computer science body of knowledge-unquote into fourteen fields, which I have divided into two groups, one of which I call the principles of computer science:

> Discrete structures; Programming fundamentals; Algorithms, complexity; Architecture, organization; Operating systems; Programming languages; Computational science, numerical methods.

The other I call specializations and artifacts:

> Net-centric computing; Human-computer interaction; Graphics, visual computing; Intelligent systems; Information management; Social, professional issues; Software engineering.

The required core is 280 hours (or about seven semester courses) spread out over all of these fourteen fields. But of these 280 hours, only 187, the equivalent of less than five semester courses, are in what I would call the principles of computer science: 43 hours in Discrete Structures, 38 in Programming Fundamentals, 31 in Algorithms and Complexity, 36 in Architecture and Organization, 18 in Operating Systems, and 21 hours in Programming Languages. In addition, the suggested curriculum for a research university requires four courses in mathematics and two in science. The rest of the curriculum consists of elective specialization, applications, and management and social issues.

(A somewhat different approach was taken by Philip Machanick, who divided the Core into 143 hours of "principles" and 137 hours of "artifacts." Philip Machanick, Principles versus artifacts in computer science curriculum design. *Computers & Education* 41 (2003), 191–201.)

Now let us look at the curriculum in low-tech mechanical engineering. Here is how the web page of the ME department at the Technion describes the course of study (`http://meeng.technion.ac.il/Studies/`):

> The student receives a strong foundation in basic subjects such as: mathematics, physics, computers, dynamics, thermodynamics, flow theory, strength and control theory. Subsequently, the student may specialize in one or more of the diverse areas of the field[:] ... advanced design and manufacturing, CAD-CAM, computer systems, control and automation, energy, mechatronics, nuclear energy, optics, and robotics. ... The first five semesters of the undergraduate studies are devoted mostly to basic sciences and engineering sciences. In the last three semesters the student takes elective courses in accordance with his chosen area of specialization. Every undergraduate student, in the last semesters of his studies, carries out a number of projects, under supervision of experienced engineers, which illustrate applications to advanced engineering problems.

Here are the courses studied in the first five-plus semesters, before the students begin to study the trendy specializations:

Mathematics Differential & integral calculus 1; Differential & integral calculus 2; Algebra; Ordinary differential equations; Numerical analysis; Partial differential equations; Statistical methods in engineering.

Science Chemistry; Physics 1; Physics 2; Physics lab 1; Physics lab 2.

Theory of mechanical engineering Solid mechanics 1; Solid mechanics 2; Introduction to materials engineering; Manufacturing processes; Thermodynamics; Dynamics; Fluid mechanics; Mechanical engineering design; Linear systems; Heat transfer; Introduction to control & automation; Introduction to mechatronics; Electrical actuators; Finite elements for engineering analysis.

Process ("Software Engineering") Design for manufacturing; Experimental methods.

Skills and Allied Subjects Computerized engineering drafting; Technical English; Introduction to computers; Engineering economy.

To summarize, here is a comparison of "hi-tech" CS vs. "low-tech" ME:

	CS	ME
Mathematics	4	7
Science	2	5
Theory and concepts	5	14
Total	11	26

Surprise, surprise! Mechanical engineering is surely one of the most mature and concrete engineering disciplines. Many computer science students probably look down on ME students as being not bright enough to study new hi-tech stuff. Yet ME students study twelve courses in science and mathematics and fourteen courses in the theoretical basis of the subject before they are even allowed to choose electives in areas of specialization that constitute actual engineering practice. This is more than twice as much as the CS requirements according to the CC2001.

I would like to offer an anecdote to emphasize this disparity between what we teach in CS and what is taught in the mature disciplines of science and engineering. A project leader at an aerospace company once told me that in his experience it is easier to teach computing to a physics major than it is to teach physics to a computer science major. Given the firm theoretical foundation of students of even a concrete subject like mechanical engineering compared with the artifact-laden education of a computer science student, this statement is not at all surprising.

The comparison with physics is also important from the point of view of the *legitimacy* of the curriculum and the teacher. Physics students in both high schools and in their first-year at the university spend large amounts of time studying the 350-year-old theory of Newtonian mechanics, and solving problems concerning inclined planes and pulleys. This material is hardly relevant to the current activities of practicing physicists, but it would be highly unusual for a student, parent or prospective employer to complain and to demand that introductory students *begin* by studying relevant subjects like the quantum mechanics of semiconductors or black holes. Physics educators have achieved a legitimacy that enables them to dictate a learning sequence that is not affected by trends and fashions.

I believe that CS education must fundamentally change in order to equip the student with a firm, deep and broad theoretical background, long before specialization is undertaken. We are grown up now and with growing up comes the responsibility to build a mature system of education.

Thank you again for granting me this award. I hope that ten or twenty years from now, the recipient of this award will be able to report on a convergence of the educational practices in computer science with those in other scientific and engineering disciplines. And maybe by then, the Concorde, or one of its descendents, will by flying again.