# The Invisible Programmers

Keynote Speech at the

Conference on Methods, Materials and Tools for Programming Education

Tampere, Finland

4 May 2006


## Mordechai (Moti) Ben-Ari

Department of Science Teaching

Weizmann Institute of Science




Good morning. I would like to thank Hannu-Matti Järvinen for inviting me to speak at this conference. I have been to Finland many times, including two sabbaticals, but never to Tampere, and I am very pleased to be here.

The title of my talk is "The Invisible Programmers". The reason for this title is that most of the software that you actually see (and hence what you think programmers do) is software packages for personal computers and web pages on the internet. But many, perhaps most, software engineers work on other types of software, in particular software for systems that are not exclusively software systems or whose purpose is other than just having a program work. The reason it is important to talk about this is that I think that too much of computer science education is directed at the limited needs of the visible programmers. This has two disadvantages: first, students and prospective students do not see the full range of future employment opportunities and thus many may turn away from this profession for lack of interest; second, and perhaps more seriously, graduates in computer science are not prepared for working in many applications areas, and thus are at a serious disadvantage when competing with graduates in other fields of science and engineering. I feel that we will be letting them down if we do not prepare them for the invisible opportunities.

My next slide shows a very, very, very expensive personal computer. It is called a Mercedes-Benz S-Class car. I don't really know how much it costs, but probably too

much for a professor's salary. There is a story that says of such things: "If you have to ask the price, you can't afford it"! So why do I call it a personal computer? Well, obviously it is "personal", because usually one person buys it, but strangely enough, it can easily be called a computer, because it contains more software than many software packages that you run. I received this slide from Klaus Grimm who runs a software research unit at DaimlerChrysler. I'll translate the German: More than 50 controllers, more than 600,000 source code lines, hundreds of bus messages, thousands of signals and a network composed of three busses. Clearly, by any standard this is a very complex computer system. It is also a very interesting computer system, and much more challenging to work on than just another software package for an ordinary personal computer.

We often hear that the demand for programmers and software engineers is declining, due to the bust of the so-called dot.com bubble. This is actually not true, as employment opportunities in computing are steady or rising, especially if you know where to look for them. In the following slide, you can see that software development for automobiles is an expanding field. It might be exaggerated: I doubt that there will be a billion lines of code in a car within the next few years, but there is no question that cars of the future will contain hundreds of thousands, if not millions, of lines of code. The next slide is even more fascinating: a very large percentage of the price of a car—about 25% in the case of the Mercedes-Benz S-Class—is due to the electronics, and the percentage will go up! Since almost all modern electronic components contain programmable elements, the opportunities for good software engineers in this industry are expanding.

Some authorities have even more extreme predictions. Rainer Hartenstein, an IEEE Fellow and an expert on computer architecture, quotes the following prediction that by 2010, 90% of all code will be for embedded systems. (The quote comes from a British government source, but unfortunately Rainer has not been able to locate for me the exact source.) Even if this is exaggerated, it still points out that a large amount of programming and software engineering is going to done that is invisible to students who see only personal computers and the internet.

What are the characteristics of these embedded systems? First of all, they are long-lived. This is especially apparent in aircraft and spacecraft, where a computer system is used for decades without being replaced. But even in automobiles, if you have a company that successfully developed a computerized brake controller, you don't change it just because Intel or AMD has released a processor with a few more MHz! It costs so much to develop a car that a model is sold for at least five years with only minor modifications. Students who are used to demanding the very latest computers have to realize that most software development is done on stable hardware and software platforms that are not changed very often.

Next, there is the issue of reliability and responsibility. Have you ever read the so-called "end-use license agreements" that come with software? (As an aside, note that you don't actually buy a software package like you do a car, you just license its use.) Read this agreement from Microsoft and replace the names Microsoft and Software with DaimlerChrysler and Mercedes-Benz: "DaimlerChrysler warrants that the Mercedes-Benz car will perform substantially in accordance with the user manual for a period of ninety days; YOU ARE NOT ENTITLED TO ANY DAMAGES; DaimlerChrysler provide[s] the Mercedes-Benz car AS IS AND WITH ALL FAULTS, and hereby disclaim[s] all other implied warranties of merchantability, of fitness for a particular purpose, of reliability or availability, of accuracy or completeness of responses, of results, of workmanlike effort, of lack of viruses, and of lack of negligence"!! It doesn't work this way in the real world. You have give a two or three year warranty with a car, you have to fix all the faults, you are responsible for negligence and you have to pay large sums of money as damages if someone gets hurt. It is not an excuse if the brakes fail because of a software error, rather than a mechanical failure.

Another characteristic of embedded systems is that the software engineer has to understand the system. It is not like writing an editor, where the programmer knows what has to be done and can accommodate her own personal preferences. Instead, if you really want to be part of the development team, you have to obtain a deep knowledge of the engineering principles involved. I'll give an example later on.

Students are not given enough exposure to cross-platform development. They use their own personal computers to develop programs for their own personal computers. In

practice, you may be writing software for a controller that is still under development. When integration begins it can be quite a challenge to figure out where a fault resides. Similarly, it can be difficult to test embedded systems. I'm sure you've heard of the failure of the first launch of the Ariane 5 rocket due to a software error. One of the problems was that it is impossible to "debug" such a system; you can't launch a $300 million rocket ten times a day, just to check out the software. Similarly, your access to the prototype of a car is likely to be limited. This forces software engineers to devote much more time to develop techniques for testing and even better to use formal methods to reduce the amount of testing that must be done.

I once met an excellent programmer who said that before beginning to develop a program he writes the user manual; that way, he knows when he is finished the development. What do our students do? First the write a program and then they write a user manual that explains how to use the program they wrote. In embedded systems, this simply cannot work. No one team (and even no one company) does all the development. A car manufacturer might buy the brakes from one subcontractor, the climate control from another and the engine from a different division of the company located in a different country. Precise specification of requirements and interfaces is essential.

We are so used to upgrading our computers that we do not give a second thought to issues of efficiency. It is true that efficiency is the enemy of good software, but sometimes it cannot be avoided. Extra memory and faster processors cost more and a manufacturer will not necessarily raise the price of a car just because your algorithms need more CPU power or memory. Instead, you will be given a fixed set of resources to work with. Similarly, embedded systems often work to marketing schedules: new model years for cars, delivery contracts for airplanes and so on.

Finally, embedded systems often use specialized technology. Over the past few years, there has been an incredible trend to uniformity in computer science education: personal computers with Windows or some variety of Unix, and Java or one of the similar C-something languages. Here is a list of some of the technologies used in embedded systems. Students who demand to be taught a certain language because it is "needed now" may not know what is going on now in some of the "invisible" industries and certainly they don't know what they will be required to use in the future. This implies

that there be a lot more diversity in teaching computer science even at the introductory level.

Now that we know what software development for embedded systems is like, let us check what is required to work on them, and who is being trained for these positions. There are two comprehensive recent studies, but both of these are focused on what is called "IT" (information technology). The first by Gallivan, Truex and Kvasny checked ads in the trade newspaper *Computerworld*, as well as a local Atlanta newspaper. I searched the electronic text of their article: the words science, mathematics and electronics do not appear. That, of course, is not surprising, because a position working on the development of software for a brake controller is unlikely to appear in *Computerworld*! In a much more comprehensive study using a variety of sources (including ads from *Computerworld*), Sami Surakka of the Helsinki University of Technology concluded that physics and continuous mathematics were not important for software developers. Since Surakka's research was intended to guide the computer science curriculum, this brings up the danger that computer science graduates might eventually lack the preparation for the types of systems I have been talking about.

So if not *Computerworld*, where should we look? Positions with companies engaged in embedded systems development do not necessarily advertise in general newspapers. Recruitment will be done through specialized publications, visits to universities and word of mouth from current employees to faculty and students.

Still, we can find examples on the web. For example, Boeing has a website that describes open positions. Here is one: "Perform Avionics System and Software Requirements Analysis, Design, Development, Unit Test and Integration for Real Time Embedded Software for the International Space Station program in Houston." There are several things worth noting here. First, they are using the supposedly useless Ada programming language, and the VAX computer systems which ceased manufacture last year. Training in trendy hardware and software does not qualify you for this work. Second, note the last two sentences: "Debug problems encountered on-orbit." Clearly, you aren't going to be debugging problems by blasting off to visit the space station;

debugging embedded systems is very complex and challenging. “Assist with the resolution of complex programmatic and technical problems.” Well, you can’t assist in resolving problems if you don’t know any of the math and physics involved. The next slide shows the qualifications needed for the position. The first paragraph is more or less standard: some experience and the ability to communicate and work in teams. The second paragraph is somewhat upsetting. To develop software, there doesn’t seem to be any special requirement for a degree in computer science or software engineering. A degree in mathematics, physics or some other field of engineering is just as good. Apparently, there is no real added value to a degree in our field, even for a position in software development.

Here is another employment opportunity, this time from DaimlerChrysler in Germany for the development of software for engines and transmissions. If you ask me, this sounds like a fascinating and challenging position, much more than building a web site or writing a package for arranging MP3 files. Now let us look at the qualifications. The first, programming experience in C or C++, is OK, typical of what one can expect from a computer science graduate. The second, experience with embedded systems is also OK, but what about experience with Matlab. Matlab is a package for performing symbolic mathematics and mathematical calculations. But that is going to be a problem if we accept the research claiming that mathematics and physics are not “needed” for employment in software development. The third qualification is really shocking: a degree in Electrical or Computer Engineering. What they are saying is: if you want someone to develop software for an interesting embedded system, don’t bother with a graduate in computer science or software engineering. The next quote I heard personally and it hurts: It is easier to teach computing to a physics graduate than it is to teach physics to a computer science graduate.

Fortunately, I did my BSc degree in mathematics before there were computer science degrees, so it was always relatively easy for me to become a part of a team working on embedded systems because the mathematics and physics didn’t frighten me. Let me give you an example. This is taken from a time about twenty years ago when I worked on software for aircraft control systems, but when preparing this talk I found that

the concepts are also used in modern graphics software. The problem has to do with rotational transformations. Any object in three-dimensional space can rotate on three axes; these rotations are conventionally called pitch, roll, and yaw. Now after so many years, I don't remember the exact context, but the idea is that given the current coordinate system of the airplane, you want to compute a new coordinate system after the airplane moves in the three rotational dimensions. One way to do that is to multiply by a matrix whose elements are expressions of trigonometric functions of the three angles, denoted here by psi, phi and theta. Now already you can see that a software engineer needs to know at least high school trigonometry and a bit of linear algebra. But it gets better.

I'm sure you all studied complex numbers at some time. Although the concept of the square root of -1 is not intuitive, complex numbers enable the description of positions and transformations in a two dimensional plane, using what are called polar coordinates. In 1843, Sir William Rowan Hamilton (you probably know the name from the NP-Complete problem of finding a Hamiltonian cycle in a graph) was able to generalize complex numbers into a mathematical concept called quaternions. He is said to have made the breakthrough while walking along a canal in Dublin, and he carved the equations into a bridge. A plaque now commemorates the place. A position in three dimensional space is described by a normalized four-dimensional vector, where i, j, and k are all square roots of -1 and form what is called in abstract algebra a non-commutative ring. Now you would think that such a construct is useless pure mathematics, but it turns out that computing a rotational transformation can be done much more efficiently and with less round-off error by multiplying quaternions instead of multiplying the matrix of Euler angles.

When I first encountered this concept, I spent a few days in the library figuring out the mathematics, and I wrote a summary of the relation between coordinate transformations and quaternions so that programmers could understand what they were writing. The point of this example is again that if you claim that mathematics and physics are not needed by programmers, this might be true in certain fields, but there are lots of invisible programmers who really do need this knowledge. We must not uniformly

remove these subjects from the computer science curriculum, but instead encourage many students to learn them.

The next topic I want to talk about is how to achieve reliable software. Recall the contrast I made earlier between the "end-use license agreements" which disclaim all responsibility, and the very strict requirements for warranties and assumption of liability in industries such as transportation and even finance. It turns out that it *is* possible to develop reliable software. One example is the UK company Praxis High Integrity Systems. They describe in the literature a 100,000 source line project for a smart card with extreme requirements for security. Believe it or not, Praxis gives a one-year guarantee on its software, promising to fix any bug that arises. In this project, only four bugs were uncovered and they were fixed very quickly. Please compare this with any program you have worked on.

How did they do this? They used a combination of a very strict adherence to principles of software engineering, together with the use of advanced technology for reliable software. First, they used the Z language that can specify requirements very precisely. Z does require an elementary knowledge of set theory and functions, so people tend to avoid it.

Second, they used the Ada and Spark languages for developing reliable software. In my opinion, the world of software has suffered immensely from not using Ada. Most viruses and software crashes come from exceeding array bounds and errors with pointers, and these can be avoided at compile time, as was demonstrated in Pascal 35 years ago. Ada simply extended these concepts into a language with structures for large systems. 25 years ago it was claimed (and I believe that the claim was incorrect even then) that there was non-negligible runtime overhead to these checks. Certainly this is no longer significant now that the speed of CPUs has increased by three orders of magnitude from a megahertz to a gigahertz. The best textbook I ever wrote was on Ada, but unfortunately it didn't sell many copies, so I can now let you download it for free from my web site. If you are all interested reliable software, I really do suggest that you look into Ada. Once you try it, you will never want to go back to debugging programs in C and C++.

A newer technology is called Spark. It is a language based upon a subset of Ada, together with annotations that enable static analysis and formal verification of programs to be performed by the Spark toolset. To use Spark optimally, you do require elementary knowledge of mathematical logic, but the profit in terms of avoiding runtime bugs is significant. Here is a trivial example of a procedure to compute integer division by repeated subtraction. The annotations include pre- and post-conditions, as well as invariant assertions, which are elementary concepts from program verification. In addition, there are annotations like "derives" which enable static analysis. In this case, we are specifying that the output variables for the quotient and the remainder are derived from the input variables X1 and X2. Here you can see that running the Spark examiner tool, it determines that there are no problems with the flow analysis. Suppose now that I made a mistake and declared the parameter R as mode in out, meaning that its value is copied in at invocation and copied out at return. Now this is legal Ada although I'm ignoring the initial value by assigning 0 to R in the first line of the procedure. Spark will catch this problem, because the parameter signature and the annotations are not compatible. In a large program it is just such inconsistencies that cause so many bugs.

Now we can try to verify the correctness of the program. Those of you with a background in program verification will easily see that there will be verification conditions for each of three paths: from the precondition to the loop assertion, around the loop from the loop assertion back to the loop assertion, and from the loop assertion to the postcondition. Actually, there are four verification conditions, because there is an *if* statement within the loop, so there are two paths around the loop, depending on whether the *then* branch or the *else* branch is taken. Here are the four verification conditions written out in mathematics and you can try to prove them. It is really not that difficult. However, the Spark Examiner tool can automatically generate the verification conditions as shown in the next four slides. Note that in one of them we assume that r+1 is equal to x2 and in another we assume that r+1 is not equal to x2, these being the conditions for the branches of the if-statement. Not only can Spark generate the verification conditions, but the Spark Simplifier can prove three out of the four automatically. What is left is a simplified version of the third verification condition that can be easily proved mathematically in a few lines.

What I hope I have shown is that static analysis and program verification are not esoteric topics for graduate students in the theory of computer science, but practical tools that—with the support of modern toolsets like Spark—can be taught to undergraduates.

The last topic I want to talk about is concurrency. As many of you know, this is a topic that I have worked on for a long time, publishing several textbooks and pedagogical software tools. You will excuse me now for promoting the latest edition of my book that appeared just last year. I think that after writing it for the third time in twenty five years, I finally did it right! In a recent ACM President's letter David Patterson, an expert on computer architecture, forecast an important future for concurrency because of the limitations of current technology: von Neumann architectures and fabrication techniques. What I like about concurrency is that you can get a lot of very interesting and exciting computational behavior with very little syntax and semantics. It is sufficient to add a process construct and semaphores to a simple language like a subset of Pascal or C and students can study all the essential topics in concurrency. We have been successful in teaching concurrent and distributed computation to high school students.

What I want to show you today is the concept of model checking for verifying properties of concurrent programs. Consider the frog puzzle, where you click on a frog to get it to move or jump over an adjacent frog. The object of the puzzle is to exchange the positions of the male and female frogs. It is actually non-trivial to solve! Let us now look at a set of questions concerning the puzzle. Are there solutions to the puzzle? From any non-final state, is there always a path to a solution? Does the existence of a solution depend on which side makes the first move? Intuitively not, but it is worth giving an answer and justifying it. What is the minimum number of steps before deadlock? What is the minimum number of steps before reaching a non-final state from which a solution cannot be reached? How can we answer these questions?

They can be solved by building a state diagram containing all possible states and transitions. For example, here are two states. You can go from the left state to the right state by having the male frog move one stone to the right. Note that some of the states are final states, for example, the blue state in this slide, because by the rules of the puzzle no

more moves can be made. We can create a diagram of all accessible states by starting with the initial state and systematically extending the diagram with all possible transitions. The completed diagram (which unfortunately is quite unreadable when reduced to fit on a slide) can be used to answer all the questions I posed before; in particular, there are two solutions to the puzzle, one in which a male frog moves first and one in which a female frog moves first.

Now this looks like a game for young kids but it actually enables us to teach the concept of model checking. I have become a big fan of the Spin is a model checker developed by Gerard Holzmann. This is a real software tool that is widely used in industry, but it can be used very easily by students. Spin simulates and verifies models written in the Promela language with correctness specifications written in linear temporal logic (LTL). The installation and running of Spin is trival. The use of a programming language—Promela—means that programs are familiar and understandable. Correctness properties are easy to express.

Here is a program for the frog puzzle. There is an array of stones, whose values can be nothing, male frog or female frog. mF is a process type for male frogs that can be instantiated for several frogs. A guarded commands syntax easily encodes the rules of the puzzle: if the next stone is empty, the frog will move to it; on the next slide, we see that if the next stone is not empty but the one after it is, the frog can jump to the empty stone. The code for the female frog is similar. (You can download this code from the website associated with my concurrency book.)

You can run the program repeatedly and get different answers because of the inherent non-determinism of the program, since in any state there may be several possible moves and jumps of the various frogs. However, it is more interesting to ask Spin to search for a solution. First we define what success is: the three left stones contain female frogs, the middle stone is empty and the three right stones contain male frogs. Now we do something tricky: we claim (incorrectly, of course) that always the variable success is false. Spin now tries to prove that we are bad—that we made a false claim, so it searches for a counterexample, namely, a state in which success is true. Once this happens, we ask

Spin to display the sequence of transitions leading to the counterexample, thus obtaining the solution.

So here is another example, where a puzzle and a program that are readily accessible to students employ the most advanced techniques used to verify highly reliable systems. I am currently working on a set of tools to facilitate the use of Spin by students. (You can download all these tools from my web site.) I have developed the jSpin integrated development environment that enables you to edit a Promela program, run it in various modes and filter the output to make it easier to study. I also have a tool called SpinSpider that automatically generates a graphical presentation of the entire state diagram. Currently, Mikko Vinni, a student at the University of Joensuu here in Finland, is working on a tool to improve the presentation of these graphs.

Let me conclude my talk by describing how I would like to see computer science education. First, I believe that all students should have introductory courses in mathematics: a course in discrete mathematics (set theory and logic) that is needed for formal methods and a course in continuous mathematics (the very basic essentials of calculus, linear algebra and differential equations) that is needed in applications. All students should know something about architecture and assembly language programming, because deep down it is the computer that is executing the statements you write. And I really do believe that every student should minor in some applications area: my suggestions would be electronics or physics or economics or biology. This will enable the student to fit into some environment without appearing to be totally ignorant of what is going on.

At the introductory level of computer science education, I would like to see more diversity: not everyone has to learn object-oriented programming, certainly not at the introductory level. Students can just as easily start with mathematical software tools like Matlab, with languages and tools for reliable software like Ada and Spark, with hardware languages and systems, or with concurrent and distributed computation. The world of software is diverse and those of us engaged in computer science education must ensure that are students receive an education that is equally diverse.