The Future of Computing: Logic or Biology

Text of a talk given at
Christian Albrechts University, Kiel
on 11 July 2003

Leslie Lamport

21 July 2003

In the late 60s, Bob Floyd and Tony Hoare discovered the concept of program correctness.

(The idea actually goes back to von Neumann. As Fred Schneider has observed, a number of explorers discovered America before Columbus—but when Columbus discovered America, it stayed discovered. That's why the U.S. celebrates Columbus Day, not Leif Erickson Day.)

Some people initially working on program correctness thought that, in the future, everyone would prove that their programs were correct. Programming errors would then disappear.

We soon learned how naive that hope was. Floyd and Hoare showed that we could, *in principle*, do this. But *in practice*, it turned out to be very hard.

Today, we know that verification is economically feasible only in a small number of applications—mainly, for fairly small programs that perform life-critical functions. Verification techniques *are* being used successfully to help debug programs, at Microsoft and elsewhere. But actually proving the correctness of a program is rarely done.

But Floyd and Hoare did make a profound contribution to our way of thinking about programs. They taught us that a program is a mathematical object, and we could think about it using logic.

This idea inspired me, in early 1977, to write a short note titled *How to Tell a Program from an Automobile*. Let me read part of it.

An automobile runs, a program does not. (Computers run, but I'm not discussing them.) An automobile requires maintenance, a program does not. A program does not need to have its stack cleaned every 10,000 miles. Its *if* statements do not wear out through use. (Previously undetected *errors* may need to be corrected, or it might be necessary to write a new but similar program, but those are different matters.) An automobile is a piece of machinery, a program is some kind of mathematical expression.

Programmers may be disheartened to learn that their programs are not like automobiles, but are mathematical expressions. Automobiles can be loveable—one can relate to them almost as if they were alive. Mathematical expressions are not loveable—they are hard to relate to. Many programmers may feel like clinging to their belief that programs are like automobiles. However, further thought reveals that mathematical expressions do have certain advantages over automobiles.

Unlike an automobile, a mathematical expression can have a meaning. We can therefore ask whether it has the *correct* meaning. One cannot talk about the correctness of an automobile—it may run properly, but it makes no sense to say that it is correct. Since a program is a mathematical expression, one can (at least in principle) decide if it is correct. The user and the programmer can agree in advance what the meaning of the program should be, and the programmer can prove mathematically that his program has that meaning. An automobile salesman can never prove that he sold a properly running automobile, he can only give enough evidence to satisfy a jury.

I have tried briefly to indicate the difference between an automobile and a program. The programmer should now apply his new knowledge in the field, and see if he can tell the difference by himself. He should first examine his automobile, and ask whether it is running properly, or if it has some bugs and requires maintenance. If he cannot answer this question, he may have to ask an auto mechanic to help him. He should then examine a program, and ask what its meaning should be, and whether he can prove that it has the correct meaning. If he can't answer these questions, it is probably because the program was written

> by someone who thought he was building an automobile. In that case, I suggest that the programmer ask these questions about the next program he writes—while he is writing it! In this way, any programmer can learn to tell a program from an automobile.

All I advocated in this note was that a programmer should ask himself (or herself) two questions:

1. What is the meaning of the program?

   In other words, what is it supposed to do?

2. Can he (or she) prove that it has the correct meaning?

   In other words, can he or she prove that it does what it's supposed to?

I didn't say that programmers should *write down* precisely what their programs mean. I didn't say that they should actually *prove* that their programs have those meanings; just that they ask if they *can* prove it. So, I thought that this was not in any way controversial.

At the time, I was working at Mass. Computer Associates. When I distributed my note to my colleagues there, it provoked outrage. I'm still not sure why. But they objected to the idea of regarding a program as a mathematical object and thinking about it logically.

Later in 1977, I moved to SRI International. Sometime within the next couple of years, the following incident occurred.

Those were the days of the Arpanet, the precursor of today's Internet. Widespread communication by email was still somewhat new, and glitches were frequent. At one point, our mail program displayed the following behavior.

If I tried to use the *reply* option for email from the Stanford artificial intelligence lab, the program would type out something like

> *SUAI* is not a correct address

However, if I used the *send-message* option and typed *SUAI* as the destination address—exactly the same address it just told me was incorrect—then everything worked fine.

So, I called our system administrator to report this problem. He explained that the source of the problem was that Stanford was sending out email that was formatted incorrectly. Since it was their fault, he didn't want to do anything about it.

This was perfectly reasonable. However, when talking to him, it became clear that he saw nothing fundamentally wrong with the program saying *SUAI* is an incorrect address, and then accepting *SUAI* as correct. This contradictory behavior did not bother him at all.

I realized that there was something very wrong with the way that system administrator's mind worked. It is illogical for a program to say that *SUAI* is incorrect and then to accept *SUAI* as correct input. Someone who finds nothing wrong with illogical behavior is not going to write programs that act logically. Which means that he is not going to write programs that you or I want to use.

That all happened twenty years ago. In retrospect, thinking of programs as automobiles wasn't so bad. Automobiles are pretty simple. If their car stops working, people expect any good mechanic to be able to figure out why and fix the problem.

Twenty years ago, people felt that way about programs too. Programs were something they could understand. Even if they weren't willing to admit that a program is a mathematical object, they still felt that, like an automobile, a program could be debugged with the aid of logical reasoning.

That's no longer the case. In the intervening years, computer programs have become much more complicated. People no longer expect to be able to deduce what's wrong when a program doesn't work.

Instead of thinking of programs as automobiles, people now think of them as biological systems.

Biology is very different from logic—or even from the physical sciences. If biologists find that a particular mutation of a gene is present in 80% of people suffering from a certain disease, and missing from 80% of a control group, then that's a significant result.

If physicists tried to report an experiment in which 80% of the events supported their theory, they'd be required to explain what happened in the other 20%.

And imagine if a mathematician submitted a theorem that he claimed was true because it was correct on 80% of the examples he tried.

I don't mean to put down biology or biologists. It's a difficult field because the systems they study are so complex. The human body is a lot more complicated, and hence a lot harder to understand, than an automobile.

And a lot harder to understand than logic.

When you get systems that are too complicated to understand, people respond with superstition. Thousands of years ago, people didn't understand astronomy. So superstitions arose—like praying to appease the spirits thought to be responsible for an eclipse.

Superstitions arise because people (and other animals) are naturally inclined to interpret coincidence as causality. If someone handles a frog and then develops a wart, he will naturally assume that touching the frog caused the wart. Especially if he does not understand what causes warts. Since a person will tend to forget all the times he has touched frogs without getting warts, it's easy to see how people came to believe that touching frogs caused warts. And since the human body is so complicated and not well understood, people have come to believe in a number of medical superstitions like homeopathy and faith healing.

We understand automobiles. There are no homeopathic automobile repair shops, that try to repair your car by putting infinitesimal dilutions of rust in the gas tank. There are no automotive faith healers, who lay their hands on the hood and pray. People reserve such superstitions for things that they don't understand very well, such as the human body.

Today's computer programs seem as complicated and mysterious to people as the human body. So they respond with superstitions. Just as a person might avoid frogs because he once got a wart after touching one, he might avoid a particular sequence of commands because he did it once and his computer crashed. He doesn't expect to understand why touching a frog might cause a wart, or why that particular sequence of commands might cause his computer to crash. Superstitions arise because we *don't* understand.

If this trend continues, we may soon see homeopathic computer repair. Perhaps it will consist of adding a virus to a program, deleting the virus, and then running the program. You may soon be able to take your laptop to a faith healer, who will lay his hands on the keyboard and pray for the recovery of the operating system.

Is this the future of computing? Or can we instead build on the idea that a computer program is a mathematical object that can be understood through logic?

When considering this question, we must realize that the nature of computing has changed dramatically in the last twenty years. Twenty years ago, computers performed only simple, logical operations. Responding to an email message is a simple operation with a simple mathematical description. Computers were used by technically sophisticated people, who interacted with them only in terms of those logical operations.

Today, computers are used by almost everyone. Most people want to act like human beings when they interact with their computers. And human beings do not naturally interact through logic.

Consider the drag-and-drop feature in modern computer interfaces. If I drag a file icon from one folder and drop it into another, that moves the file to the other folder. If I drag a file icon from a folder and drop it onto a program icon, that runs the program with the file as input.

Moving a file and running a program on the file are logically very different operations. Yet, I find it quite natural for the same operation of drag-and-drop to perform these two logically different operations. Why? Because that's how the human mind works.

Consider these two sentences:

- I'm running – *Ich laufe.*
- The computer is running – *Der computer läuft.*

These are logically two different meanings of run/*laufen*. But it seems perfectly natural to use the same word for both concepts—a person moving rapidly on foot, and a computer sitting in one place executing a program. Why?

Humans think and speak with metaphors. When machines were invented, people thought and talked about them using concepts they were already familiar with. In Germany and England, they used the metaphor of a person running to describe what a machine does. So machines, and hence computers, *laufen.*

In China, the metaphor they used was that of a person doing labor. So in Chinese, machines *work*. (But computers *execute.*)

We think in terms of metaphors, and the metaphors we use influence how we think. Thirty years ago, there was something of a debate about whether to use the name *memory* or *storage* for what we now call memory. I was opposed to personifying computers, so I thought it should be called *storage*. I was wrong. *Storage* calls up an image of boxes neatly lined up in a warehouse. It was appropriate for the simple arrays used then in most programs. But *memory* is a more amorphous term. It's much more appropriate for the variety of structures with which data is maintained by today's programs.

And, quite appropriately, today we reserve the term *storage* for media like disks, in which data is kept in neat arrays of blocks.

Metaphors are not logical. There is no logic to machines *running* in English and German, *working* in Chinese, and *turning* in French. Metaphors can be good or bad, depending on how well they work. *Memory* works better than *storage* at helping us write programs.

Drag-and-drop works very well. Dragging a file to a folder and bringing the file to a program that will use it are good metaphors for moving a file and for running a program on the file.

If how we use computers is to be based on metaphors, and metaphors are not based on logic, what role does logic have in the future of computing?

I said that I could perform two different operations by drag-and-dropping a file icon onto a folder icon, or onto a program icon. That's not quite true. I can do that if the file icon is in a folder. But if the file icon represents a file attached to a message, I can't. Using the *Outlook* mail program, I can drag-and-drop the attached file onto a folder icon. As expected, that puts a copy of the file into the folder. But I can't drag-and-drop it onto a program icon. That doesn't work.

Why not? I don't know. I can find no explanation. Not only is the behavior illogical, but it's hard to understand how it could have been programmed that way.

Here's how I think drag-and-drop works in Windows. When the user clicks on the mouse to begin dragging the item, the program on which it is clicking tells Windows what is being dragged. When dragging from a folder, that program is the File Explorer. When dragging from a message, the program is Outlook.

The File Explorer program and the Outlook program must be responding to the mouse click by telling Windows two different things, since two different things happen when I drop the file onto a program icon. Why? Why should there even be two different things a program *could* tell Windows?

The only answer I can find is that this part of Windows was programmed by the former SRI System Administrator. Remember him? The one who saw nothing fundamentally wrong with a mail program saying that *SUAI* is an incorrect address, and then accepting *SUAI* as a correct address. If not by him, then by someone very much like him.

Computers interact with users through metaphors. Metaphors are not based on logic, but they are not illogical.

Metaphor and logic are not opposites. They are complementary. A good program must use good metaphors, *and* it must behave logically. The metaphors must be applied consistently—and that means logically.

The inconsistency with the *SUAI* address in my old mail program at SRI, and with the handling of drag-and-drop by *Outlook*, are minor nuisances. It was easy enough to type *SUAI* to the SRI mail program; I can drag-and-drop an *Outlook* message attachment into a folder and then drag-and-drop it from there onto a program icon.

I am dwelling on these problems because of what they reveal about our software and the people who write it. They tell us that there is a fundamental illogic in the programs. And that illogic comes from an inability of programmers to think logically about their programs.

Today's computer programs are very large and complex. They have to be large and complex because they do many things. The old SRI mail program just handled simple ASCII text messages. The *Outlook* mailer handles

- formatted text—including letters and accents from many languages,
- images,
- audio files,
- calendar appointments,
- and all sorts of attachments.

It

- maintains multiple message folders,
- keeps an address book—remembering the addresses of everyone from whom I've received email,
- issues reminders,
- sends vacation messages when I'm away,
- filters spam,

and does all sorts of other things. And it uses a sophisticated graphical interface that most people find much more natural and easy to use than the command-line interface provided by the old mail program.

How can programmers and system designers cope with this complexity? There are two approaches.

The first is to use the paradigm of the complex systems with which most people are familiar—namely, biological systems. We let computer systems *evolve*. We regard malicious attacks as *viruses*, which we combat by mimicking the complex defense systems used by living organisms.

This sounds like a marvelous approach. Look at how well it works in nature. The biological system known as *homo sapiens* is a marvel of engineering—much more sophisticated than any system designed by man.

Unfortunately, there are some problems with adopting this approach. First of all, it's very slow. It took nature millions of years. We don't have that much time.

Second, its results are far from perfect. Just consider all the auto-immune diseases, like arthritis, that we suffer from. Biological systems tend to be just good enough to survive (which admittedly has to be quite good given what it takes to survive in a hostile world). We should have higher aspirations for our software than "just good enough to survive".

Third, biological systems respond quite slowly to a changing environment. Nature can keep up with changes in weather patterns that take place over thousands of years. But species often become extinct when faced with the rapid changes caused by man. We must adapt our computer systems to rapid changes in technology.

The fundamental problem with approaching computer systems as biological systems is that it means giving up on the idea of actually *understanding* the systems we build. We can't make our software dependable if we don't understand it. And as our society becomes ever more dependent on computer software, that software must be dependable.

The other approach to dealing with complexity is through logic. As Floyd and Hoare showed us so many years ago, a program is a mathematical object. We can understand complex mathematical objects by structuring them logically.

The best way to cope with complexity is to avoid it. Programs that do a lot for us are inherently complex. But instead of surrendering to complexity, we must control it. We must keep our systems as simple as possible. Having two different kinds of drag-and-drop behavior is more complicated than having just one. And that is needless complexity.

We must keep our systems simple enough so we can understand them. And the fundamental tool for doing this is the logic of mathematics. We face two tasks in applying this tool.

Our first task is learning how. Floyd and Hoare pointed the way by showing how mathematics could be applied to tiny programs. We need to learn how to extend what they did to the real world of large programs.

Extending what they did does not mean trying to mathematically prove the correctness of million-line programs. It means learning how to apply the idea of a program as a mathematical object to the task of building large programs that we can understand, and that do what we intend them to do.

There are many ways in which mathematics can be used to produce better computer systems. My own work in this area has been directed towards getting engineers to start using mathematical methods in the earliest stage of system design. In my lecture yesterday,[1] I described the progress that I've made so far, and how hardware designers at Intel are taking advantage of it. I'm just starting to try to understand how these methods can be applied to software development at Microsoft.

Other researchers are developing tools for applying the ideas pioneered by Floyd and Hoare to such varied problems as improving the coverage of testing and checking device drivers supplied to Microsoft by other companies.

In addition to the task of learning how to apply mathematics to large systems, we also face the task of teaching programmers and designers to think logically. They must learn how to think about programs as mathematical objects. They must learn to think logically. We will not have understandable programs as long as our universities produce generation after generation of people who, like my former colleagues, cannot understand that programs are different from automobiles. And like that SRI system administrator who saw nothing wrong with the mail program's illogical behavior.

I'm not familiar with how computer science is taught in European universities. I know that American universities are doing a terrible job of it.

[1] This was a lecture about the TLA$^+$ specification language. Some of the things I said can be found in the paper *High-Level Specifications: Lessons from Industry*, to appear in the Proceedings of the First International Symposium on Formal Methods for Components and Objects, held 5-8 November 2002 in Leiden, The Netherlands. It is available on my publications page, accessible from `http://lamport.org`.

There is a complete separation between mathematics and engineering. I know of one highly regarded American university in which students in their first programming course must prove the correctness of every tiny program they write. In their second programming course, mathematics is completely forgotten and they just learn how to write C programs. There is no attempt to apply what they learned in the first course to the writing of real programs.

In my lecture yesterday, I described how being forced to describe their designs mathematically taught the Intel engineers how to think about them mathematically. And thinking mathematically improved their design process.

Thinking mathematically is something that engineers should be taught in school. They shouldn't have to learn it on the job. They can learn the details of some particular programming language on the job. Teaching how to think is the task of the university. But today, students in the U.S. can take courses in C++ and Java and the details of the Unix operating system. But few American universities have courses devoted to teaching how to think logically and mathematically about real systems. Universities need to do better.

When people who can't think logically design large systems, those systems become incomprehensible. And we start thinking of them as biological systems. And since biological systems are too complex to understand, it seems perfectly natural that computer programs should be too complex to understand.

We should not accept this. That means all of us, computer professionals as well as those of us who just use computers. If we don't, then the future of computing will belong to biology, not logic. We will continue having to use computer programs that we don't understand, and trying to coax them to do what we want. Instead of a sensible world of computing, we will live in a world of homeopathy and faith healing.