

Interview

An Interview with C.A.R. Hoare

C.A.R. Hoare, developer of the Quicksort algorithm and a lifelong contributor to the theory and design of programming languages, discusses the practical application of his theoretical ideas.

THE COMPUTER HISTORY Museum has an active program to gather videotaped histories from people who have done pioneering work in this first century of the information age. These tapes are a rich aggregation of stories that are preserved in the collection, transcribed, and made available on the Web to researchers, students, and anyone curious about how invention happens. The oral histories are conversations about people's lives. We want to know about their upbringing, their families, their education, and their jobs. But above all, we want to know how they came to the passion and creativity that leads to innovation.

Presented here are excerpts^a from an interview with Sir Charles Antony Richard Hoare, a senior researcher at Microsoft Research in Cambridge, U.K. and Emeritus Professor of Computing at Oxford University, conducted in September 2006 by Jonathan P. Bowen, the chairman of Museophile Limited, and Emeritus Professor at London South Bank University.

What did you want to be growing up?

I thought I would like to be a writer. I

^a Oral histories are not scripted, and a transcript of casual speech is very different from what one would write. I have taken the liberty of editing and reordering freely for presentation. For the original transcript, see <http://archive.computerhistory.org/search/oh/>.

—Len Shustek



didn't know quite what I was going to be writing, but at school I was a rather studious and uncommunicative child, and so everybody called me "Professor." I found the works of Bernard Shaw very inspiring. He's of course an iconoclast, so he would appeal to an adolescent. Also Bertrand Russell, who wrote on social matters as well as philosophical and mathematical matters.

What was your first exposure to computers?

I began thinking about computers as a sort of philosophical possibility during my undergraduate course at Oxford University. I took an interest in

mathematical logic, which is the basis of the formal treatment of computer programming. I was sufficiently interested that one of my few job interviews was with the British Steel just after I finished my university course in 1956. I was attracted by their use of computers to control a steel milling line. A little later I attended an interview at Leo Computers Ltd. in London, who were building their own computers to look after the clerical operations of their restaurant chain. But I didn't follow up on either of those prospects of employment.

What was the first program you wrote?

In 1958 I attended a course in Mercury Autocode, which was the programming language used on a computer that Oxford University was just purchasing from Ferranti. I wrote a program that solved a two-person game using a technique which I found in a book on game theory by von Neumann and Morgenstern. I don't know whether it worked or not. It certainly ran to the end, but I forgot to put in any check on whether the answers it produced were correct, and the calculations were too difficult for me to do by hand afterward.

What was programming like in those days?

Very different from today. The programs were all prepared on punched cards or paper tape. It might take a day to get them punched up from the cod-

ing sheets, and then they were submitted to a computer maybe the following day. It would take a long time, if there were any faults in the program, to find out where they were.

How did you come to live in Russia?

I did national service, which was compulsory in those days, in the Royal Navy studying modern military Russian. I used to know the names of all the parts of a ship in Russian, even if I didn't know what the actual parts of the ship were. Later I continued my graduate career as a visiting student at Moscow State University for a year.

The 1960s were very exciting times in Russia, especially after the U.S. spy plane was shot down. I felt quite free, and no political problems obtruded. But our Russian friends were very suspicious of each other. We learned quite early on that you never introduce one Russian friend to another, because each of them thinks the other one is the informer. We knew that our rooms were bugged, so we would never talk about Russian friends inside our own rooms.

You developed the famous Quicksort algorithm at about this time. Why?

The National Physical Laboratory was starting a project for the automatic translation of Russian into English, and they offered me a job. I met several of the people in Moscow who were working on machine translation, and I wrote my first published article, in Russian, in a journal called *Machine Translation*.

In those days the dictionary in which you had to look up in order to translate from Russian to English was stored on a long magnetic tape in alphabetical order. Therefore it paid to sort the words of the sentence into the same alphabetical order before consulting the dictionary, so that you could look up all the words in the sentence on a single pass of the magnetic tape.

I thought with my knowledge of Mercury Autocode, I'll be able to think up how I would conduct this preliminary sort. After a few moments I thought of the obvious algorithm, which is now called bubble sort, and rejected that because it was obviously

I think Quicksort is the only really interesting algorithm that I've ever developed.

rather slow. I thought of Quicksort as the second thing. It didn't occur to me that this was anything very difficult. It was all an interesting exercise in programming. I think Quicksort is the only really interesting algorithm that I ever developed.

Where did you work after returning to England?

I met my future employers in Russia. I was an interpreter at an exhibition in Moscow, where Elliott Brothers, which at that time made small scientific computers, were exhibiting and selling their computer in Moscow. They offered me employment when I came back, with an additional 100 pounds a year on my salary because I knew Russian. I never had a formal interview.

What did you work on at Elliott?

They were embarking on the design of a new and very much faster computer, and they thought they would celebrate by inventing a new language to program it in. As a recent employee, with six months experience, I was put to designing the language. Fortunately I happened to see a copy of the *Report on the Algorithmic Language Algol 60*, and I was able to recommend to the company to implement that, rather than inventing a language of their own. That proved a very good commercial decision. And a good personal one, because I eventually married Jill, the other programmer who came to work on the same project. She had experience writing a compiler before, which in those days was quite unusual, and she was a much better-disciplined programmer than I ever was.

Was Algol well defined?

The syntax was formally defined. The

grammar of the language was written up in a way that was, I think, completely unambiguous. The semantics was a little less formally defined. It used ordinary English to describe what the effect of executing a program would be. But it was very well written by Peter Naur, and it was sufficient to enable us to write a compiler without ever consulting the original designers of the language. And it was sufficient for programmers in the language to write programs, which in the end actually ran on our compiler, without ever consulting us or the original designers of the language. It was a really very remarkable achievement—rather beyond what maybe we can achieve these days in the design of languages.

Did you collaborate with other compiler writers?

We didn't correspond with other people writing compilers, even for Algol, in those days. We didn't know each other. There was no real scientific community that one could join to talk over problems with other people who encountered the same problems. We worked pretty well on our own.

After moving to Queen's University in Belfast in 1968, you wrote a very important paper on the axiomatic approach to computer programming, now known as "Hoare Logic."

I was interested, as indeed many people were at that time, in making good the perceived deficiency of the Algol report: that while the syntax was extremely carefully and formally defined, the semantics was left a little vaguer. We were pursuing the goal of trying to get an equally good formalization of the semantics of the language, a goal that I think still is pretty advanced and maybe beyond our grasp.

I put forward the view that we didn't want the specification to be too precise. We didn't want the specification of a programming language to concentrate in too much detail on the way in which the programs were executed, but rather we should set limits on the uncertainty of the execution of the programs, to allow different implementations to implement the language in different ways. In those days the word lengths and the arithmetic of all of the

computers was different. Based on the ideas of mathematical logic that I studied at university, I put forward a set of axioms that describe the properties of the implementation without describing exactly how it worked. It would be possible, I hoped, to state those properties sufficiently precisely that programmers would be able to write programs using only those properties, and leave the implementers the freedom to implement the language in different ways, but at the same time taking responsibility for the fact that their implementation actually satisfied the properties that the program was relying on.

I haven't abandoned this idea, but it didn't turn out to be very popular among language designers.

You then turned to “structured programming,” and collaborated with Edsger Dijkstra and Ole-Johan Dahl on an important book.

I met Dijkstra and Dahl at a working conference in 1972 on formal language definition. Dijkstra was the other person writing an Algol compiler at the same time as I was, and Dahl was inventing a new simulation language called Simula, in which he introduced the ideas of object-oriented programming that would later have a great influence on programming and programming languages. All three of us had written draft monographs on our favorite topics: one by Dijkstra called “Notes on Structured Programming,” one by Dahl on hierarchical program structures, and my own notes on data structuring. I thought it would be interesting to collect these three together and publish them as a single book.

I spent quite some time trying to understand what Dahl had written. He was a very brilliant but very dense writer. I had great fun trying to simplify some of his really brilliant ideas on how to structure programs in the large. I did not work on Dijkstra's material; that was pretty clear and well written.

What was your involvement with Pascal?

Pascal was the language designed as a teaching language by my friend Niklaus Wirth, after the language we had designed together in the early 1960s had not been recommended.

We used to meet quite frequently to talk about the design. My research on applying the axiomatic method to programming language semantics had made quite considerable progress by then, and I thought it was time to see whether I could tackle a complete language using this style of definition. I managed to do the easy bits, but there were still quite a lot of challenges left over, which we just omitted from the definition of the language.

When did you start to look at monitors for operating systems?

In 1972 I organized a conference in Belfast where we assembled quite a brilliant group of scientists to talk about operating system techniques. I was interested in exploring the ideas of from an axiomatic point of view. Could I define axioms that would enable people to safely write concurrent programs in the same way as they can write sequential programs today? We devoted an afternoon to discussing the emerging ideas of monitors; I and Per Brinch Hansen were the main people to contribute to that discussion.

How did you come to move from monitors to communicating sequential processes?

The idea of a communicating process is that instead of calling its components as subroutine, or a method, you'd actually communicate the values that you wanted to transmit to it by some input or output channel, and it would communicate its results back by a similar medium. The reason for this was a technological advance in

It was easy to predict that the best way of making a large and fast computer would be to assemble a large number of very cheap microprocessors together.

the hardware: the advent of the micro-processor, a cheap and small machine with not very much store, but capable of communicating with other micro-processors of a similar nature along wires. It was easy to predict that the best way of making a large and fast computer would be to assemble a large number of very cheap microprocessors together, and allow them to cooperate with each other on a single task by communicating along wires that connected them. For this, a new architecture of programs would be appropriate, and perhaps a new language for expressing the programs. That was how the communicating sequential processes came to take a leading role in my subsequent research.

You've always been interested in the connection between theory and practice.

My move back to Oxford was partially motivated by my idea to study the Oxford ideas on the semantics of programming languages again. I hoped that it would be possible, using the Oxford techniques of defining semantics, to clarify the exact meaning of communicating processes in a way that would be helpful to people writing programs in that idiom.

It was a great loss that Christopher Strachey had recently died. I took over his chair at Oxford, quite literally sitting in his chair and sitting at his desk. I happened to open the drawer, and come across a final report on one of his research projects, in which he put forward his ideals for keeping the theory and practice of programming and computer science very much in step with each other. He said the theory could easily become sterile, and the practice could easily become ad hoc and unreliable, if you weren't able to keep one firmly based on the other, and the other firmly studying problems with the practical uses of computers.

Have there been successes using formal methods in practice?

At a keynote lecture for the British Computer Society I talked a bit about formalization and verification, and put forward a conjecture, fairly tentatively, that maybe the time was right to scale these things up by trial use

in industry. A senior director from IBM at Hursley came up to me after the lecture and invited me to put my commitment where my mouth was, and do something in collaboration with IBM. That made me gulp a bit, because I had the impression that IBM had produced some pretty complicated software, and this really would be a challenge. There was a good chance that we would fall flat on our faces. But you can't turn down an opportunity like that. So some colleagues and I set to work, and actually produced some very useful analyses for them using the Z notation of Jean-Raymond Abrial to help them with an ongoing project for restructuring and rewriting parts of their popular customer information software system, CICS. That work eventually led to a Queen's award for technology.

The Transputer was another example of a very practical application of your theoretical ideas.

The INMOS Transputer was as embodiment of the ideas that I described earlier, of building microprocessors that could communicate with each other along wires that would stretch between their terminals. The founder had the vision that the CSP ideas were ripe for industrial exploitation, and he made that the basis of the language for programming Transputers, which was called Occam.

When they came to develop the second version of their Transputer that had a floating-point unit attached, my colleagues Bill Roscoe and Jeff Barrett at Oxford actually used formal models of communicating processes, and techniques of program verification, to check that their designs for the implementation of the IEEE floating-point specification were in fact correct. The company estimated it enabled them to deliver the hardware one year earlier than would otherwise have happened. They applied for and won a Queen's award for technological achievement, in conjunction with Oxford University Computing Laboratory, for that achievement. It still stands out as one of the first applications of formal methods to hardware design.

I expect the future to be as wonderful as the past has been. There's still an enormous amount of interesting work to do.

What projects are you working on at Microsoft?

One of them is the pursuit of my lifetime goal of verification of programs. I was very interested in the ideas of assertions, which had been put forward by Bob Floyd and before; already in 1947 the idea of an assertion was described by Alan Turing in a lecture he gave to the London Mathematical Society. This idea of assertions lies at the very basis of proving programs correct, and at the very basis of my ideas for defining the semantics of programming languages. I already knew when I entered academic life way back in 1968 that these ideas would be unlikely to find commercial exploitation, really, throughout my academic career. I could look forward to 30 years of research uninterrupted by anybody who actually wanted to apply its results.

I thought that when I retired, it would be very interesting to see whether the positive side of my prediction would also come true, that these ideas would begin to be applied. And indeed, even since I've joined Microsoft in 1999, I've seen quite a bit of expansion in their use of assertions and other techniques for improving confidence in the reliability of programs. There are program analysis tools now that stop far short of actually proving correctness of programs, but they're very good at detecting certain kinds of errors. Some quite dangerous errors, which make the software vulnerable to intrusions and virus attacks, have been detected and removed as a result of the use of formal techniques of program analysis.

The idea of verifying computer

programs is still an idea for the future, although there are beginnings of using scientific technology to make the programs more reliable. The full functional verification of a computer program against formally specified requirements is still something we have to look forward to in the future. But the progress that we've made has really been quite spectacular in the last 10 years.

My other project is concurrency. I've set myself the challenge of understanding and formalizing methods for programming concurrent programs where the programs actually share the store of the same computer, rather than being executed on distinct computers as they are in the communicating sequential process architecture. Again, the motivation for studying this different form of concurrency is the advance of hardware technology: it now appears that the only way in which processors can get faster is for them to include more processing units on the same chip, and share the same store.

I'd always felt that parallel programs that actually shared main memory, and could interleave their actions at a very fine level of granularity—just a single memory access—were far too difficult for me. I could see no real prospect of working out a theory that would help people to write correct programs to exploit this capability. I thought it would be interesting to try again, and see whether the experience in formalization that has been built up over the last 20 or 30 years could be applied effectively to this extremely complicated form of programming.

What is on the horizon for computer science?

I expect the future to be as wonderful as the past has been. There's still an enormous amount of interesting work to do. As far as the fundamental science is concerned, we still certainly do not know how to prove programs correct. We need a lot of steady progress in this area, which one can foresee, and a lot of breakthroughs where people suddenly find there's a simple way to do something that everybody hitherto has thought to be far too difficult. ■

Edited by **Len Shustek**, Chair, Computer History Museum, Mountain View, CA.