# Does it always have to be so complicated?

1987

C.A.R. Hoare F.R.S.

Professor of Computation

Oxford University

The most serious obstacle to the wider exploitation of the promise extended by the hardware of modern computers is the impenetrable and still growing complexity of their software. The removal of this barrier will be the main achievement of research in computer science in the next decades.

Not long ago, the smallest stored-program digital computer filled a large room with shelves of thermionic valves and tanks of mercury. Through the magnificent achievements of hardware engineers, the size of a computer rapidly reduced to a few cabinets, a single cabinet, a single shelf, a single board, and now just a few chips. This progress is expected to continue.

Progress in software has been in the opposite direction. For the earliest computers, a few dozen instructions of a bootstrap loader sufficed as an operating system; the program library contained a few mathematical routines; and the machine was programmed in a decimal machine code, translated by an assembler of some hundreds of instructions. Gradually, the size of the software grew to fill and overfill the increasing capacity of the hardware. Operating systems, program libraries, and compilers for high level languages are now measured in millions of instructions.

This is certainly progress, but is it in the right direction? If our present computers are thousands of times easier to use now than they were in the beginning, we should be very happy. I fear this is not so. The reason is that the software has become so complicated that most of its users can no longer understand or control it. Furthermore, they can no longer trust its reliability or its stability: a constant stream of new releases remedy defects of previous releases, and introduce new faults of their own.

A crude symptomatic measure of this problem is the size of the manuals required to explain to the hopeful user how to take advantage of each item of software, and what to do when things go wrong. The slim manual for an early computer was just a description of the machine structure, and an account of the effect of the few dozen instructions which comprised its machine code. Soon the manuals grew to book size, and then to multiple volumes; they rapidly overflowed shelves and racks; and now a complete set of software manuals for a modern mainframe computer occupies a complete room. Software documentation is now as voluminous as the hardware of the earliest computers!

Many professional programmers tolerate or even welcome this growth in complexity; a malicious explanation is that the complexity ensures that the computer cannot be properly used by the layman without professional assistance. In the same way, administration, accountancy, and the law also tend toward such complication that they can be understood only by specialised professional administrators, accountants and lawyers.

This kind of man-made complexity is to be sharply contrasted with the complexities of nature, which are the concern of practising engineers, doctors and scientists. Indeed, it is the primary objective of research scientists in these disciplines to discover the simple laws which govern the complex phenomena of nature, so that vast catalogues of experimental observations can be reduced to a few formulae of mathematics, which can be understood and readily applied by the non-specialist.

The achievement of such simplicity is far from simple. The discovery that thousands of chemical compounds can be explained in terms of a mere hundred elements; the explanation of these hundred elements in terms of a few atomic particles; the explanation of hundreds of high energy particles in terms of a few dozen quarks; each of these achievements eluded the greatest intellects of earlier times, though now these results are common knowledge among schoolchildren.

My own area of software expertise is Programming Languages; these provide excellent examples of cancerous growth in complexity - PL/I, ALGOL 68, CHILL, and now ADA. And yet in the design of programming languages there has been no lack of simple ideas of great generality.

(1) The parametrised subroutine, which made FORTRAN into an extensible language.

(2) The block structure of ALGOL 60, which introduced the methods of structured programming.

(3) The multi-register desk (now pocket) calculator, on which the attractive simplicity of BASIC was founded.

(4) The data type declaration of PASCAL, which extends the concept of structure to the storage of data.

(5) The general recursive function, as incorporated in LISP.

(6) The communicating sequential process, on which the tasking feature of ADA has been based.

But why do languages which start with such simple ideas become so hideously complicated?

There are many reasons:

(1) the pursuit of efficiency on unsympathetic computer hardware; this explains the PROG feature in LISP, and irregularities in the type structure of PASCAL.

(2) simple error or oversight, for example, the own variable of ALGOL 60, or the "shallow" binding of LISP.

(3) historical accident: the FORMAT of FORTRAN is copied from some input/output routines which happened to be available for the IBM 650.

(4) design by committee leads to inclusion of features whose complexity and interactions have not been fully explored. ALGOL 68 and ADA provide sad examples.

(5) overambition, which leads to the vain search for a single language to solve all problems.

(6) the need for compatibility with every previous version of a language prevents any subsequent simplification.

(7) commercial interest, which leads a manufacturer to add features to his implementation that will make it impossible for customers to transfer their programs to machines of a different manufacturer.

(8) academic respectability, which requires that every language design emanating from a University must be original.

(9) premature inclusion of necessary features such as input and output, before the simplicity of the relevant mathematical theory has been discovered.

(10) Intellectual stagnation of programmers, who have suffered the trauma of learning the complexity of one programming language, and recoil from the suggestion that another language might be better.

But I believe that in the next two decades we shall see the solution to many of these problems. The difficulties encountered by a new generation of programmers of personal computers has led to a wider recognition of the scale of the problem, and has added strength to the will to solve it. Long experience has taught us the many subtle temptations of complexity, and the need to avoid them. Recent discoveries in the mathematical theory of programming and programming language semantics offer the same promise of simplification that has been realised in the spectacular advances of modern science which depend wholly on the relevant mathematics. Recent developments in VLSI and computer architecture have mitigated the pressures imposed by efficiency of implementation.

Thus I have some confidence in setting targets of simplicity for new programming languages of the future. Let us use the crude but simple measure of the size of the accompanying documentation.

(1) A formal definition of the language in terms of mathematical equations should occupy one or two pages - no more than the axioms on which the major branches of mathematics are based.

(2) An informal description, in the style of the Report on the Algorithmic Language ALGOL 60, should occupy ten to twenty pages.

(3) A textbook should describe how to specify, design, document and implement programs in the language. It should fill one or two hundred pages; and should be suitable for use in schools.

(4) A library of useful algorithms expressed in the language should also be published openly in both readable and magnetic form.

These goals will be attained not just for a single language, but for a variety of languages covering different applications areas, and based on different mathematical foundations. For example, a language like PROLOG will be based on the predicate calculus; a functional language like LISPKIT will be based on recursive function theory; and a language of Communicating Sequential Processes will be based on newer mathematical theories, like those explored by Milner in his Calculus of Communicating Systems. The new language "occam" designed by David May at INMOS is a good example of this class.

The task of preserving the simplicity of these languages in a practical implementation is a major challenge to computer science. For example, a practical implementation must deal with long-term storage of voluminous data held on unsympathetic devices like discs. It must deal with the peculiarities of various input and output devices, such as screens, printers, plotters, voice encoders, etc; and it must do so without significant loss in efficiency. The great temptation will be to add new "features" to the language, and thereby pass the problem on to the user. The temptation is increased by the fact that the user can be easily persuaded to welcome such additional features. It is sad to reflect that even when the implementer has succeeded in hiding all the complexities, the user will be ignorant of the problems he has been spared, and will not appreciate the benefits of his ignorance.

When a new range of simple well-defined programming languages has been widely accepted, I believe that we shall see a general improvement in the quality and reliability of programs expressed in these languages. The designers of operating systems and libraries, and also the professional applications programmers, will recognise the possibility of basing their designs on well-specified mathematical models, and will respond to the challenge of implementing them efficiently in a manner which preserves the simplicity and elegance of a well-defined user interface. Thus we will achieve an essential objective that the ordinary user can always understand what the computer is doing for him, and he can always retain control and responsibility over the actions of his machine.

This, I believe, is the right way to overcome the dangerous attitude of helpless incomprehension with which many of the general public approach the computer. An alternative which may seem superficially attractive is to program the computer to try to understand the modes of thought of its user. But this could be very dangerous. It means that the machine could deceive its user and persuade him to accept the assumptions consciously or unconsciously built into the original program. The consequences of inadvertent or unscrupulous manipulation are even more frightening than the dangers of misunderstanding due to excessive complexity.

To avoid the dangers widely predicted in science fiction, we must ensure that it is always the man that understands and controls the machine, never the other way round. To ensure this requires a deliberate and determined pursuit of simplicity and reliability in the design of programs, software, and systems. I hope that progress in this new direction will be signposted by the designers of new programming languages, in which the new programs will be expressed.