Assertions: A Personal Perspective

C.A.R. Hoare
Microsoft Research

Assertions are Boolean formulas placed in program text at places where their evaluation will always be true. If the assertions are strong enough, they express everything that the programmers on either side of an interface need to know about the program on the other side, even before the code is written. Indeed, assertions can serve as the basis of a formal proof of the correctness of a complete program.

Editor's Note

This article is based on the Laureate Lecture delivered on 12 November 2000 at a symposium organized by Akinori Yonezawa on the occasion of the award of the Kyoto Prize for Advanced Technology.

My interest in assertions and their role in program proofs was triggered by my early industry experience; subsequently, through my university research, I extended the concept into a methodology for program specification and design. Now that I have returned to industrial employment, I've had the opportunity to investigate the current role of assertions in industrial program development. My personal perspective illustrates the complementary roles of pure research, aimed at academic ideals of excellence, and the unexpected ways in which the results of such research contribute to the gradual improvement of engineering practice.

Experience in industry, 1960–1968

My first job was as a programmer for a small British computer manufacturer, Elliott Brothers of London. My task was to write library programs in decimal machine code[1] for the company's new 803 computer (see Figure 1). After a preliminary exercise that gave my boss confidence in my skill, I was entrusted with implementing a new sorting method recently invented and published by Donald Shell.[2] I enjoyed optimizing the inner loops of my program to exploit the machine code's most ingenious instructions. I also enjoyed documenting the code according to the company's standards for customer products. Even testing the program was fun; tracing the errors was like solving mathematical puzzles. How wonderful that programmers get paid for that too! In fairness, I thought, the programmers should pay back to their employers the cost for removal of their own mistakes.

What wasn't such fun was the kind of error that caused my test programs to run wild and crash; often, errors overwrote the data needed to diagnose the cause of the error. Was the crash due perhaps to a jump into the data space or to an instruction overwritten by a number? The only way to find out was to add extra output instructions to the program, tracing its behavior up to the moment of the crash. But the sheer volume of the output only added to the confusion. Remember, in those days the lucky programmer was one who had access to the computer just once a day. Even 40 years later, the problem of crashing programs is not altogether solved.

After six months on the job, I was assigned an even more important task—that of designing a new high-level programming language for the company's newer, faster computers. By good fortune, I happened to acquire a copy of Peter Naur's "Report on the Algorithmic Language Algol 60,"[3] which had recently been designed by an international committee of experts. The company decided to implement a subset of that language, which I selected with the goal of efficient implementation on the Elliott computers. In the end, I thought of an efficient way of implementing nearly the whole language.

An outstanding merit of Naur's report was that it was only 21 pages long, yet it was sufficiently informative to write a compiler for the language without any communication with the language designers. Furthermore, a programmer could program in the language without any communication either with the compiler writers or designers. Even so, it was possible for the program to work the first time it was submitted to the newly implemented compiler. And, in fact, apart from a small error in the character codes, we did get one of our customers' programs to work the first time at an

 Published by the IEEE Computer Society 1058-6180/03/$17.00

exhibition of an Elliott 803 computer in Eastern Europe. Few languages designed since then have enabled compiler writers and programmers to match such an achievement.

Part of the credit for this success was the compact yet precise notation for the grammar or syntax of the language, which defines the class of texts that are worthy of consideration as meaningful programs. This notation was due originally to Noam Chomsky, the great linguist, psychologist, and philosopher.[4] John Backus first applied the notation to programming languages, in a 1959 article on the syntax and semantics of the proposed language of the Zurich ACM-GAMM Conference.[5] After dealing with the syntax, Backus anticipated a continuation article on the semantics. It never appeared. In fact, the original article challenged researchers to find a precise and elegant formal definition of the meaning of programs, which inspires solid research in computer science even today.

The syntactic definition of the language served as a pattern for the structure of our Algol compiler, which used recursive descent. As a result, it was logically impossible (almost) for any error in a submitted program's syntax to escape detection by the compiler. If a successfully compiled program went wrong, the programmer had complete confidence that the cause was not the result of a misprint that made the program meaningless.

Chomsky's syntactic definition method was soon more widely applied to other programming languages, with results that were rarely as attractive as for Algol 60. I thought that this failure reflected the intrinsic irregularity and ugliness of the syntax of these other languages. One purpose of a good formal definition method is to guide the designer to improve the quality of the language it's used to define.

Object code design

In designing the machine-executable object code to be output by the Elliott Algol compiler,[6] I believed that no program compiled from the high-level language should ever run wild. Our customers had to accept a significant performance penalty because every subscripted array access had to be checked at runtime against both upper and lower array bounds. They knew how often such a check fails in a production run, and they told me later that they didn't want even the option to remove the check. As a result, programs written in Algol would never run wild, and debugging was relatively simple, because a programmer could infer the effect of every program from the program's source text without knowing anything about the compiler or about the machine on which it was running. If only we had a formal semantics, I thought, to complement the language's formal syntax, perhaps the compiler would be able to help in detecting and averting other kinds of programming error as well.

Figure 1. An early 803 computer in typical surroundings. (Courtesy of Museu Virtual de Informática da Universidade do Minho, Portugal, http://www.dsi.uminho.pt/museuv/.)

Interest in semantics was widespread. In 1964, 51 scientists from 12 nations attended a Vienna conference on formal language description languages.[7] One of the papers was "The Definition of Programming Languages by their Compilers,"[8] by Jan Garwick, pioneer of computing science in Norway. The title appalled me because it suggested that the meaning of any program is determined by selecting a standard implementation of that language on a particular machine. So if you wanted to know the meaning of a Fortran program, for example, you'd run it on an IBM 709 and see what happened. Such a proposal seemed to me grossly unfair to all computer manufacturers other than IBM, at that time the world-dominant computing company. It would be impossibly expensive and counterproductive on an Elliott 803, with a word length of 39 bits, to give the same numerical answers as the IBM machine, which had only 36 bits in a word—we could more efficiently give greater accuracy and range for integers and floating-point numbers.

Even more unfair was the consequence that the IBM compiler was by definition correct; but any other manufacturer would be compelled to reproduce all of its errors—they would have to be called just anomalies—because errors would be logically impossible. Since then, I have always avoided operational approaches to programming language semantics. The principle that "a program is what a program does" is not

a good basis for exploring the concept of program correctness.

I didn't make a presentation at the Vienna conference, but I did make one comment: I thought that the most important attribute of a formal definition of semantics should be to leave certain aspects of the language carefully undefined. As a result, each implementation would have carefully circumscribed freedom to make efficient choices in the interests of its users and in light of the characteristics of a particular machine architecture. I was encouraged that this comment was applauded, and even Garwick expressed his agreement. (In fact, I had misinterpreted his title. His paper called for an abstract compiler for an abstract machine, rather than selecting a commercial product as a standard.)

The inspiration of my remark in Vienna dates back to 1952, when I was an undergraduate student at Oxford University. Some of my neighbors in college were mathematicians, and I joined them in a small, unofficial nighttime reading party to study mathematical logic from the textbook by Willard Van Orman Quine.[9] Later, a course in the philosophy of mathematics pursued more deeply this interest in axioms and proofs, as an explanation of the unreasonable degree of certainty that accompanies the contemplation of mathematical truth.

The axiomatic method

It was this background that led me to propose the axiomatic method for defining the semantics of a programming language while preserving a carefully controlled vagueness in certain aspects. I drew an analogy with the foundations of the various branches of mathematics, like projective geometry or group theory; each branch is in effect defined by the set of axioms that are used without further justification in all proofs of the theorems of that branch. The axioms are written in common mathematical notations, but they also contain undefined terms, like lines and points in projective geometry, or units and products in group theory; these constitute the conceptual framework of that branch. I was convinced that an axiomatic presentation of basic programming concepts would be much simpler than any compiler of any language for any computer, however abstract.

I still believe that axioms provide an excellent interface between the roles of the pure mathematician and the applied mathematician. The pure mathematician deliberately gives no explicit meaning to the undefined terms appearing in the axioms, theorems, and proofs. It is the task of the applied mathematician and the experimental scientist to find in the real world a possible meaning for the terms and check by carefully designed experiment that this meaning satisfies the axioms. The engineer is even allowed to take the axioms as a specification that must be met by a product design—for example, the compiler for a programming language. Then all the theorems for that branch of pure mathematics can be validly applied to the product or to the relevant real-world domain. And surprisingly often, the pure mathematician's more abstract approach is rewarded by the discovery that the same axiom set has many different applications. By analogy, there could be many different implementations of the axiom set that defines a standard programming language. That was exactly the carefully circumscribed freedom that I wanted for the compiler writer, who has to take on the engineer's typical responsibility that the implementation satisfies the axioms as well as efficiently running its users' programs.

My first proposal for such an axiom set took the form of equations, as encountered in algebra textbooks, but with program fragments on the left- and right-hand sides of the equation instead of numbers and numeric expressions. The same idea was explored earlier and more thoroughly in a doctoral dissertation by Shigeru Igarashi at the University of Tokyo.[10] In November 1967, I showed my first draft of a paper on the axiomatic approach to Peter Lucas; he was leading a project at the IBM Research Laboratory in Vienna to formally define IBM's new programming language, later known as PL/I.[11] Lucas was attracted by the proposal but soon abandoned the attempt to apply it to PL/I as a whole. The PL/I designers had an operational view of what each language construct would do, and they had no inclination to support a level of abstraction necessary for an attractive or helpful axiomatic presentation of the semantics. I was not disappointed. In the arrogance of idealism, I was confirmed in my view that a good formal definition method would be one that clearly reveals the quality of a programming language, whether bad or good; the axiomatic method had shown its capability of at least revealing badness. And I regarded PL/I as a bad language because it gave little protection against crashing programs.

Research in Belfast, 1968–1977

By 1968, it was evident that research into programming language semantics was going to take a long time before it found application in industry—in those days, it was accepted that long-term research should take place in universities. I therefore welcomed the opportunity to move to the Queen's University in Belfast as professor of computer science. During this time,

I came across a preprint of Robert Floyd's paper, "Assigning Meanings to Programs."[12] Floyd adopted the same philosophy as I had, that the meaning of a programming language is defined by the rules that can be used for reasoning about programs in the language. These could include not only equations but also rules of inference. In his paper, Floyd presented an effective method of proving the total correctness of programs, not just their equality to other programs (see Figure 2 for an example). I saw this as the achievement of the ultimate goal of a good formal semantics for a good programming language, namely, the complete avoidance of programming error. Furthermore, the language quality was now the subject of objective scientific assessment, based on simplicity of the axioms and the guidance they give for program construction. The axiomatic method is a way to avoid the dogmatism and controversy that so often accompanies programming language design, particularly by committees.

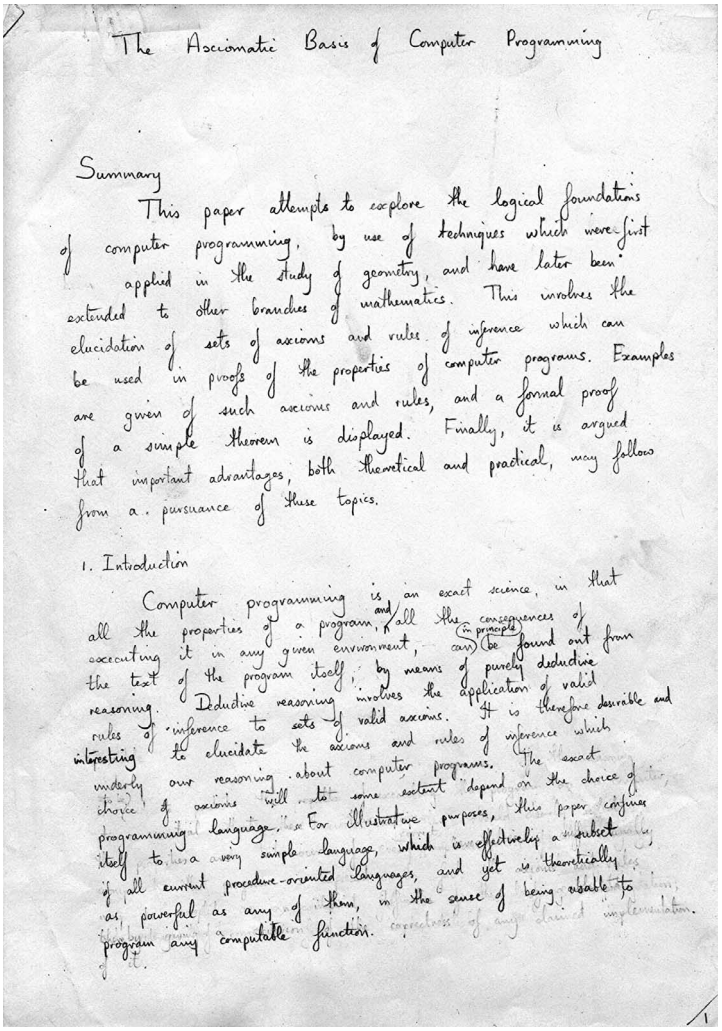
For a general-purpose programming language, correctness can be defined only relative to the intention of a particular program. In many cases, the intention can be expressed as a postcondition of the program, that is, an assertion (about the values of the program variables) intended to be true when the program terminates. The proof of this fact usually depends on annotating the program with additional assertions in the middle of the program text; these are expected to be true whenever program execution reaches the point where the assertion is written. At least one assertion, called an invariant, is needed in each loop: It is intended to be true before and after every execution of the loop body. Often, the correct working of a program depends on the assumption of some precondition, which must be true before the program starts. Floyd gave the proof rules whose application could guarantee the validity of all the assertions except the precondition, which had to be assumed. He even anticipated the day when a verifying compiler could automatically validate the assertions before running the program. This would be the ultimate solution to the problem of programming error, making it logically impossible in a running program. I correctly predicted its achievement would occur after I had retired from academic life, which would be in 30 years' time.

Lifelong research project

The first paper that I wrote as a professor in Belfast was a complete rewrite of my earlier drafts expounding and extolling the axiomatic method. Figure 3 shows a page from the man-

```
ASSERT divisor > 0 ;
remainder := dividend ; quotient := 0 ;
INVARIANT dividend == quotient * divisor + remainder ;
while divisor ≤ remainder do
        {remainder := remainder – divisor ;
         quotient := quotient + 1 }
ASSERT remainder < divisor &
        dividend == quotient * divisor + remainder
```

Figure 2. This was the first program that I proved correct. It performs positive integer division by the lengthy process of counting the number of times the divisor can be subtracted from the dividend. The assertion at the beginning states the precondition, which the user of the routine must guarantee before entry. The final assertion describes the postcondition that the routine will make true on exit. The invariant in the middle is true before and after every iteration of the loop. It is a key to the proof that explains why the loop works.



The Axiomatic Basis of Computer Programming

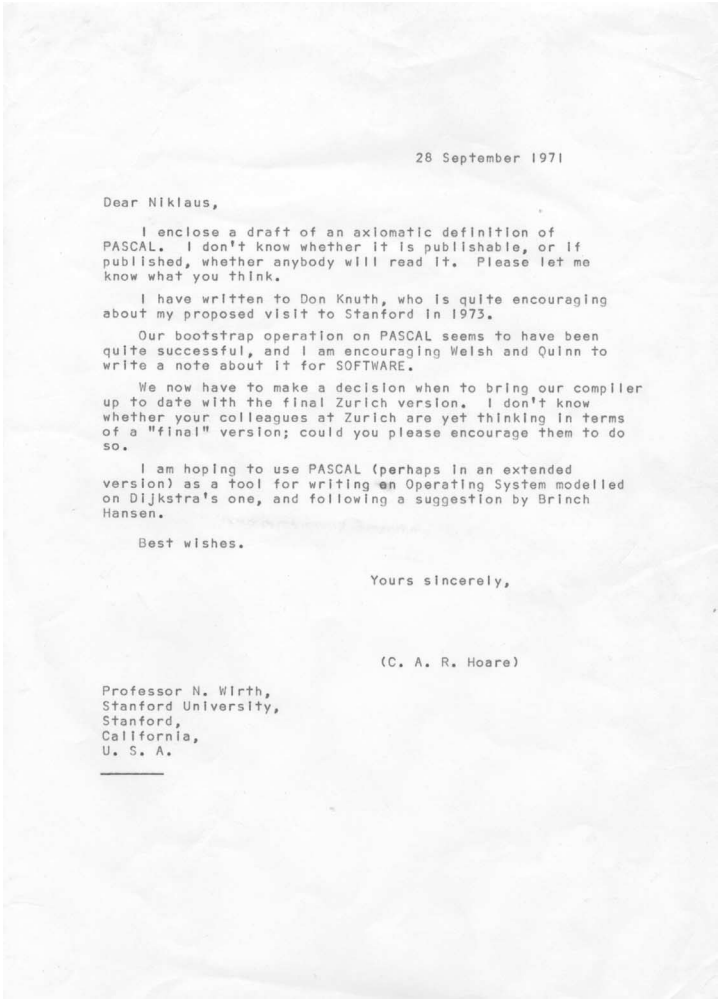
Summary

This paper attempts to explore the logical foundations of computer programming, by use of techniques which were first applied in the study of geometry, and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

1. Introduction

Computer programming is an exact science, in that all the properties of a program, and all the consequences of executing it in any given environment, can (in principle) be found out from the text of the program itself, by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underly our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper confines itself to a very simple language, which is effectively a subset of all current procedure-oriented languages, and yet is theoretically as powerful as any of them, in the sense of being usable to program any computable function.

Figure 3. A page from my manuscript on the axiomatic method. (Source: C.A.R. Hoare.)



28 September 1971

Dear Niklaus,

I enclose a draft of an axiomatic definition of PASCAL. I don't know whether it is publishable, or if published, whether anybody will read it. Please let me know what you think.

I have written to Don Knuth, who is quite encouraging about my proposed visit to Stanford in 1973.

Our bootstrap operation on PASCAL seems to have been quite successful, and I am encouraging Welsh and Quinn to write a note about it for SOFTWARE.

We now have to make a decision when to bring our compiler up to date with the final Zurich version. I don't know whether your colleagues at Zurich are yet thinking in terms of a "final" version; could you please encourage them to do so.

I am hoping to use PASCAL (perhaps in an extended version) as a tool for writing an Operating System modelled on Dijkstra's one, and following a suggestion by Brinch Hansen.

Best wishes.

Yours sincerely,

(C. A. R. Hoare)

Professor N. Wirth,
Stanford University,
Stanford,
California,
U. S. A.

Figure 4. The letter I wrote to Niklaus Wirth about my manuscript on an axiomatic definition of the programming language Pascal. (Source: C.A.R. Hoare.)

uscript of my article draft.[13] This was the start of a lifelong research project; it was followed by articles that extended the set of axioms and rules to cover all the familiar constructions of a conventional high-level programming language. These included iterations, procedures and parameters, recursion, functions, and even jumps.[13-18]

Eventually, there were enough proof rules to cover almost all of a reasonable programming language, like Pascal, for which I developed a proof calculus in collaboration with Niklaus Wirth.[19] Figure 4 shows the letter to him that accompanied my first tentative draft.

Since the late 1960s, developers have used the axiomatic method in designing languages like Euclid and Eiffel.[20,21] These languages were prepared to accept the restrictions on the generality of expression that are necessary to make the axioms consistent with efficient program execution. For example, the body of an iteration (`for` statement) should not assign a new value to the controlled variable; the parameters of a procedure should be distinct from each other, no aliases; and all jumps should be forward rather than backward. I recommended that these restrictions should be incorporated in the design of any future programming language; they were all of a kind that a compiler could enforce, to avert the risk of programming error. Restrictions that contribute to provability, I claimed, are what make a programming language good. They often make the program more efficient, too.

I was even worried that my axiomatic method was too powerful because it could deal with jumps, which Edsger W. Dijkstra had identified in 1968 as a bad feature of contemporary programming.[22] My consolation was that the proof rule for jumps relies on a subsidiary hypothesis and is inherently more complicated than the rules for structured programming constructs. Subsequent wide adoption of structured programming confirmed my view that simplicity of the relevant proof rule is an objective measure of quality in a programming language feature. Further confirmation is provided by program analysis tools, like Lint[23] and Prefix,[24] applied to less disciplined languages such as C. These tools identify those constructions that would invalidate the simple and obvious proof methods, and warn the programmer against their use.

A common objection to Floyd's method of program proving was the need to supply additional assertions at intermediate points in the program. It is difficult to look at an existing program and guess what these assertions should be. I thought this was an entirely mistaken objection. It wasn't sensible to try to prove the correctness of existing programs, partly because they were mostly going to be incorrect anyway. I followed Dijkstra's constructive approach[25] to the task of programming. The obligation of ultimate correctness should be the driving force in designing programs that are going to be correct by construction. In this top-down approach, the starting point for a software project should always be the specification, and the program proof should be developed along with the program. Thus, the most effective proofs are those constructed before the program is written. This philosophy has been beautifully illustrated in Dijkstra's *A Discipline of Programming*[26] and in many subsequent textbooks on formal approaches to software engineering.[27]

Concurrent program execution

In all my work on the formalization of proof methods for sequential programming languages, I knew that I was only preparing the way for a much more serious challenge, which was to extend the proof technology into the realm of concurrent program execution. In the early 1970s, I took as my first model of concurrency a kind of quasi-parallel programming, coroutines, which was introduced by Ole-Johan Dahl and Kristen Nygaard into Simula, and later Simula 67, for discrete event simulation.[28,29] I knew the Simula concept of an object as a replicable data structure, declared in a class together with the methods that are allowed to update its attributes. As an exercise in the application of these ideas, I took the structured implementation of a paging system: virtual memory (see Figure 5).

I suddenly realized that the purpose and criterion of this program's correctness was to simulate the more abstract concept of a single-level memory, with a much wider addressing range than could be physically fitted into the computer's random access memory. The concept had to be represented in a complicated (but fortunately concealed) way, by storing temporarily unused data on a disk.[30] The code correctness could be proved with the aid of an invariant assertion, later known as the *abstraction invariant*, that connects the abstract variable to its concrete representation.[31] The introduction of such abstractions into programming practice is one of the main achievements of the still-current craze for object-oriented programming.

The real insight that I derived from this exercise was that exactly the same proof was valid, not only for sequential use of the virtual memory, but also for its use by many processes running concurrently. As with proof-driven program development, it's the obligation of correctness that should drive the design of a good programming language feature. Of course, implementation efficiency is also important. A correct implementation of the abstraction must prevent more than one process from updating the concrete representation at the same time. This is efficiently done by use of Dijkstra's semaphores protecting critical regions;[32] the resulting structure was called a *monitor.*[33,34] The idea was simultaneously put forward and successfully tested in an efficient implementation of Concurrent Pascal by Per Brinch Hansen, a leading Danish computer scientist, who has made his career in the US.[35] The monitor has since been adopted for concurrency control by the more recently fashionable language Java,[36] but with extensions that prevent the use of the original simple proof rules.

```
{ram:real_addr → word; cache:word ;
disk:sector_addr → sector ;  page-tables, tags, etc.;
ASSERT virtual_memory: virtual_addr → word;
INVARIANT virtual_memory == ....
        a function of ram, disk, etc...;
method fetch(i:virtual_addr); ....
        ASSERT cache := virtual_memory[i];
method assign(i:virtual_addr); .....;
        ASSERT virtual_memory[i] := cache ; }
```

Figure 5. This is the skeleton of my proof of correctness of an implementation of virtual memory. The abstract memory is declared in an assertion as an array of machine words. The invariant describes how the abstract memory is concretely represented as a complicated function of page tables in RAM and content on disk. The invariant is true before and after every fetch and assignment to the memory. The purpose of each of these methods is described as an assignment involving the abstract virtual memory. The body of these methods (omitted above) is proved to have the corresponding effect on the concrete representation.

Testing my ideas

To test the applicability of these ideas, I used them to design the structure of a simple batch-processed operating system.[37] Jim Welsh and Dave Bustard, recent doctoral graduates of the Queen's University, implemented the system in an extended version of Pascal, called Pascal Plus, which they also designed and implemented.[38] We used the `inner` statement of Simula 67, which enables the code of a user process to be embedded deep inside an envelope of code that implements the abstract resources it uses. The same effect is achieved in object-oriented languages today by methods that initialize and finalize the object. In Simula, the semantics of the `inner` statement is described like that of the Algol 60 procedure call, and like inheritance in current object-oriented languages, in terms of copying textual portions of the user program inside the object code that it's using.

Dijkstra explained to me that such a copy rule completely fails to explain or exploit the real merit of the language feature, which is to raise the program's level of abstraction. We spent some time together at the 1975 Marktoberdorf Summer School in Germany, exploring the underlying abstraction, and designing notations that would most clearly express it. But I spent several more years of personal research on the topic, and I was still not satisfied with my progress. Inspiration eventually came from an unexpected direction.

At that time, the promise of very large scale integration was beginning to materialize as low-cost microprocessors. To multiply their somewhat modest computing power, researchers found it an attractive prospect to connect several such

machines by wires along which the microprocessors could communicate with each other during program execution. To write programs for such an assembly of machines, a programmer would need a language that included input and output commands; these removed the need for an explanation by textual copying. But shared memory was too expensive and thus was ruled out, and without shared memory, monitors were unnecessary.

An obvious requirement for a parallel programming language is a means of connecting two program fragments in parallel, rather than in series. Naturally I chose the structured parallel command (**`parbegin` … `parend`**) suggested by Dijkstra,[32] rather than the jumplike forking primitive made popular by C and Unix. I also included a variant of Dijkstra's guarded command,[39] enabling a program to reduce latency by waiting for the first of two or more inputs to become available. The resulting program structures were known as communicating sequential processes (CSP).[40] To answer the question of the features' sufficiency, I showed that they could easily encode many other useful programming language constructions, both sequential and parallel. These included semaphores, subroutines, coroutines, and of course monitors.

I was happy with the unification of programming concepts that I had achieved, but dissatisfied that I had no means of proving the correctness of the programs that used them. Furthermore, I left open numerous language design decisions, which I wanted to resolve by investigating their impact on the ease of proving programs correct. I hoped that a communicating process could be understood in terms of the trace, or history, of all the communications in which it could engage. On this basis, I found it was possible to get proofs of partial correctness, but only by ignoring problems of nontermination and of nondeterministic deadlock, which causes a computer to stop when a cycle of processes are each waiting for its neighbor. I was by then ashamed that I had ignored such problems in my early exposition of Floyd's proof method. Fortunately, Dijkstra had shown in his book on programming discipline[26] how to deal safely with the problem of nondeterminism. He assumed that it would be resolved maliciously by a demon, intent on frustrating our intentions, whatever they might be. He also dealt correctly with the problem of nontermination. Now I resolved that any acceptable proof method for CSP would have to incorporate Dijkstra's solutions.

Move to Oxford, 1977–1999

In 1977, an opportunity arose to move to Oxford University, where I wanted to study the methods of denotational semantics that Christopher Strachey and Dana Scott had pioneered, and ably expounded in a textbook by Joe Stoy.[41] Among my first research students, jointly supervised with Stoy, were a couple of brilliant mathematicians, Bill Roscoe and Steve Brookes. We followed the suggestion of Robin Milner that the meaning of a concurrent program could be determined by the collection of tests that could be made on it. Following Karl Popper's criterion of falsification for the meaning of a scientific theory, Roscoe and Brookes concentrated on failures of these tests, with particular attention to the circumstances in which they could deadlock or fail to terminate. This led to the now standard model of CSP, with traces, refusals, and divergences.[42,43]

This research found remarkably early application in industry. Iann Barron, who had earlier worked for Elliott Brothers on the design of the 803 computer, was inspired by the vision of a new computer architecture, the *transputer*, which he defined as a complete microprocessor, communicating with its neighbors in a network by input and output along simple wires.[44] In 1976–1977, he started the company Inmos, to design and make the hardware; he hired David May as its chief architect, and he hired me as a consultant to design a programming language based on CSP to control it. The language was named occam,[45,46] after the medieval Oxford philosopher, who proposed simplicity as the ultimate touchstone of truth.

An important commercial goal of the company was to ensure that the same parallel program would have logically the same effect when implemented by multiprogramming on a single computer as when distributed over multiple processors on a network. CSP's level of abstraction gave just this assurance. For 10 years or more, the transputer enjoyed commercial success and the language excited scientific interest. Today's advances, however, in microprocessor power, storage capacity, and network communications technology favor a more dynamic model of network configuration and a buffered model of communication, which are more directly represented in recent process algebras, like the pi-calculus.[47]

Fundamental to the philosophy of top-down program development from program specifications is the ability of programmers to write the specifications in the first place. Obviously, these specifications must be at least an order of magnitude simpler and more obviously correct than the eventual program is going to be. In the 1980s, it was accepted wisdom that the language for writing specifications should itself be

executable, making it, in effect, just another more powerful programming language. But I knew that, in principle, a language like that of set theory, untrammeled by considerations of execution or of efficiency, could express many important abstract concepts far more concisely than any executable language. I believed that these concepts drawn from mathematics would make it easier to reason about the correctness of the program at the design stage.

There's no conceivable way to prove a specification correct—against what specification would that be? Such a higher-level specification, if it existed, should have been chosen originally as the starting place for the design. So the only hope is for developers to make the original specification so clear and so easily understandable that it obviously describes what is wanted, and not some other thing. That's why it would be dangerous to recommend for specification anything less than the full language of mathematics. Even if this view is impractical, it represents the kind of extreme in expressive power that makes it an appropriate topic for academic research. Certainly, if the basic mathematical concepts turn out to be inadequate to describe what is wanted, there is little hope for help from mathematics in making correct programs.

Uniform notational framework

Mathematicians through the ages have developed a great many notations, and each branch of the subject uses the same notations for different purposes, and unfortunately, different notations for the same purpose. What is needed for purposes of programming is a uniform notational framework to match the generality of a general-purpose programming language, and sufficiently powerful for the definition of all concepts of any particular branch of mathematics that might be relevant to any computer application in the future. Fortunately, this was provided by abstract set theory, developed as a foundation for mathematics by logicians at the beginning of the last century. Set theory already provides a range of concepts known to be relevant in computing—Cartesian products, direct sums, trees, sequences, bags, sets, functions, and relations.

The same idea had inspired Jean-Raymond Abrial, a successful French software engineer, who came to Oxford in the early 1980s to continue his work on the Z specification language.[48] The power of the Z notation was first tested by researchers at Oxford, working on small tutorial examples. Many improvements resulted, both in notation and style of usage. But the crucial question was, Would they provide any practical benefit when applied to a large programming project in industry?

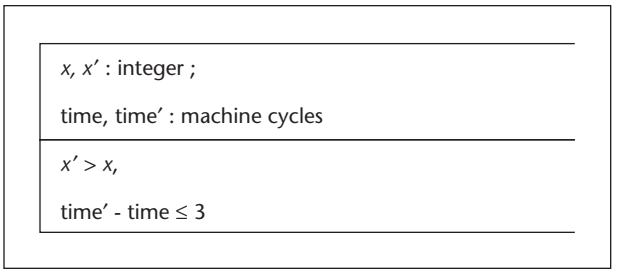


Figure 6. A schema in Z is a specification for a fragment of code. It contains (above the central line) a declaration of the global variables, with a dash on those variables that stand for a value at the end of execution; and below the line it contains a predicate describing the desired relationship between the initial and final values. The example specifies an action that strictly increases the value of *x* , and which does not take more than three machine cycles to execute.

In 1981, the IBM development laboratories in Hursley, UK, placed a research contract with the Oxford University Computing Laboratory to support a project led by Ib Sorensen and, later, Ian Hayes, and in 1983, they began to use Z for specification. One of the teams at Hursley was responsible for the development of the Customer Information and Control System (CICS), one of IBM's most successful commercial software products. IBM was planning the next release of this system, primarily devoted to the restructuring of some of its basic components. For one of the more tricky components, they bravely decided to try Oxford's new recommended top-down development method, starting with a specification in Z (see Figure 6). This involved more work in the early stages of the project, but it inspired confidence in the soundness of the new structure's design, and the early rigorous formalization averted many errors that might have been troublesome at later stages in the project. When the product was finally delivered in 1991, IBM calculated that the development costs were less than on components developed more traditionally, and the quality, as customers perceived it, was greater.[49]

The characteristic feature of Z is the *schema*, consisting of a declaration of the names of certain free variables and their types, together with a predicate expressing a desired invariant relationship between the values of those variables. The free variables play the same role as in a scientific theory—they stand for measurements like time and distance that can be made in the real world, or (in our application) they stand for observations of the state or behavior of computer programs. The meanings of the variables, and the justification for the invari-

ants, must be described informally in the extremely important natural-language prose that accompanies the specification.

As in science, there are many common conventions. Similarly, in a schema that specifies a fragment of a sequential program, a dashed variable x' always stands for the final value of a global program variable whose initial value is denoted by x. It was Cliff Jones, a leader in the development of the Vienna Development Method (VDM), who persuaded me of the need to make explicit both initial and final values of all the variables.[50]

The extra flexibility of these extra variables makes it easy to introduce extensions to the model of a programming language. For example, to model timing properties, we just need to introduce a special real-valued variable called *time*. So *time′* would be the time at which a program terminates, and *time* would be when it starts. Of course, the programmer is not allowed to assign arbitrary values to such a special variable. It can be updated only by special operations like *delay (interval)*, whose effect is simply modeled by adding the *interval* to the *time*; although the intended implementation is simple—just wait for the clock on the wall to move on. Such extra variables played a vital role in my later attempts at unifying theories of programming.

Like predicates in logic, Z schemas can be connected by any of the operators of the propositional calculus: conjunction, disjunction, and even negation. But the schema calculus also uses sequential composition; which is defined in the same way as the binary composition of relations in relational calculus. The final values of the variables of the first program (before the semicolon) are identified with the initial values of the second program (after the semicolon), and these intermediate values are hidden by existential quantification. A careful treatment of nontermination ensures that the composition of two schemas accurately describes the result of sequential execution of any pair of programs that satisfies those schemas. More formally, if P1 and P2 are programs, and if S1 and S2 are schemas, then the axiomatic proof rule for correctness of programs' sequential composition can be elegantly expressed as follows:

$$\frac{\text{P1 satisfies S1} \quad \text{P2 satisfies S2}}{\text{(P1;P2 satisfies (S1;S2)}}$$

Simpler semantics

In 1981, Rick Hehner, professor of computer science at the University of Toronto on a sabbatical visit to Oxford, came into my office and spent an embarrassingly long time persuading me that something much simpler than the axiomatic proof rules was possible.[51,52] His idea was to simply define the semantics of the programming language directly in terms of the schema calculus of Z. Each program is interpreted as the strongest schema describing its observable behavior on all its possible executions. As a result, the concept of satisfaction of a specification can be identified with the most pervasive concept in all mathematical reasoning, that of logical implication.

Furthermore, defining the semantics in terms of Z's schema calculus obviates the need for an axiomatic semantics because all the useful proof rules can themselves be proved as theorems. All the operators of the programming language are defined simply as operators on schemas. For example, the definition of semicolon in the programming language is identical to its definition I gave in the schema calculus. The displayed proof rule is no longer an axiom; it is a proven theorem stating the simple fact that relational composition is monotonic in both its operands, with respect to implication ordering.

From then on, I traveled the world giving a series of keynote addresses with different illustrative examples, but with the same message and the same title: "Programs are Predicates."[53-55]

The first application of this insight into a simpler semantics definition was to solve the long-standing problem of the specification and proof of correctness of communicating sequential processes. All that's needed is to introduce the observable attributes of a process, its trace, and its refusals, as free variables of a Z schema. Using predicate calculus, we then define CSP's various choice and parallel constructions as operators on schemas. This insight has inspired all my subsequent research. In a continuing collaboration with He Jifeng, a Chinese computing researcher working at Oxford from 1983 to 1998, we have developed a specification-oriented semantics for many other computational paradigms, including hardware and software, declarative and procedural paradigms, with sequential and parallel programming capability.

Even within parallel programming, there are many variations—some with distributed processing, some with shared memory, with dedicated channels or with shared buses for communication, either with synchronization or with buffering of messages. It turns out that there is much commonality between the mathematical properties of all paradigms, and this led us to describe our activity as unifying theories of programming.[56] This work brought to

fruition a strand of my research that was started by Peter Lauer, my first doctoral graduate student in Belfast.[57]

Research into assertions

That concludes a brief account of my long research association with assertions. They started as simple Boolean expressions in a sequential programming language, testing a property of a single machine state at the point that control reaches the assertion. By adding dashed variables to stand for the values of variables at program termination, an assertion is generalized to a complete specification of an arbitrary fragment of a sequential program. By adding variables that record the history of interactions between a program and its environment, assertions specify the interfaces between concurrent programs. By defining a program's semantics as the strongest assertion that describes all its possible behaviors, we give a complete method for proving the total correctness of all programs expressed in the language.

My interest in assertions was triggered by problems that I had encountered as a programmer in industry. The evolution of the idea kept me occupied throughout my academic career. Now, on return to industrial employment, I have the opportunity to see how the idea has progressed toward practical application and maybe help to progress it a bit further.

Back in industry, 1999–present

The contrast between my academic research and current software engineering practice in industry could not be more striking. A programmer working on legacy code in industry rarely has the privilege of starting again from scratch. If a specification is provided, it is usually no more than the instruction to do something useful and attractive, making as little change as possible in the existing code base or its behavior. The details of the design are largely determined by what turns out to be possible and adequately efficient after exploring the existing code and testing several possible changes by experiment. The only way of increasing confidence in the correctness of the changes is by debugging. The practice of specifying an interface even as simple as a histogram graphics package is quite unattractive, and formal proof is clearly inconceivable on existing code bases, measured in millions of lines of code. So how can the results of theoretical research, inspired by purely academic ideals, be brought to bear on the pervasive problems of maintaining large-scale legacy code written in legacy languages?

It's the concept of an assertion that links my earlier research with current industrial software engineering practice and provides the basis for hopes of future improvement. Assertions figure strongly in Microsoft code. A recent count discovered more than a quarter million of them in the code for its Office product suite.

The primary role of an assertion today is as a test oracle, defining the circumstances under which a program under test is considered to fail. A collection of aptly placed assertions is what permits a massive suite of test cases to be run overnight, without human intervention. Failure of an assertion triggers a dump of the program state, to be analyzed by the programmer on the following morning. Apart from merely indicating the fact of failure, the place where the first assertion fails is likely to give a good indication of where and why the program is going wrong. And this indication is given in advance of any crash, avoiding the risk that the necessary diagnostic information is overwritten.

So assertions have already found their major application, not to the proof of program correctness, but to the diagnosis of their errors. Assertions are applied as a partial solution to the problems of program crashes, which I first encountered as a new programmer in 1960. The other partial solution is the ubiquitous personal workstation, which reduces the turnaround for program correction from days to minutes.

Assertions are usually compiled differently for test runs than for code shipped to the customer. In shipped code, the assertions are often omitted, to avoid the runtime penalty and the confusion that would follow if an error diagnostic or a checkpoint dump were to appear as a customer worked onscreen. Ideally, the only assertions to be omitted are those that have been subjected to proof. But more practically, many teams leave the assertions in shipped code to generate an exception when false; to continue execution in such an unexpected and untested circumstance would run a grave risk of crash. So, instead, the exception handler makes a recovery that is reasonable to the customer in the environment of use.

Assertions are also used to advantage by program analysis tools like Prefix,[23] being developed within Microsoft for legacy code maintenance. The value of such tools is limited if they give so many warning messages that the programmer can't afford the time to examine them. Ideally, each warning should be accompanied by an automatically generated test case that would reveal the bug, but that will depend on further advances in model checking and theorem proving. Assertions and assumptions provide a means for the programmer to explain that a certain error cannot occur

or is irrelevant, and the tool will suppress the corresponding sheaf of error reports. Report suppression is another motivating factor for programmers to include more and stronger assertions in their code. Another acknowledged motive is to inform programmers engaged in subsequent program modification that certain program properties must be preserved.

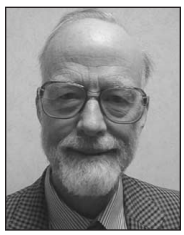
My work with Microsoft concentrates on further design and development of tools to assist in the programming of trustworthy systems and applications. In other engineering disciplines, design automation tools embody an increasing amount of scientific knowledge, mathematical calculations, and engineering know-how. My hope is that similar tools will lead the way in delivering the results of research into programming theory to the working software engineer, even to one who is working primarily on legacy code.

I suggest that assertional proof principles should define the direction of evolution of sophisticated program analysis tools. Without principles, a program analysis tool has to depend only on heuristics, and after a time, further advance becomes increasingly difficult. There is the danger that programmers can learn to write code that has all the characteristics of good style as defined by the heuristics, and yet be full of bugs. The only principles that guard against this risk are those directly based on considerations of program correctness. And that is why program correctness has been, and still remains, a suitable topic for academic research.

References and notes

1. Elliott Brothers, *Elliott 803 Programming Manual*, London Ltd., 1960.
2. D. Shell, "A High-Speed Sorting Procedure," *Comm. ACM,* vol. 2, no. 7, July 1959, pp. 30-32.
3. P. Naur, ed., "Report on the Algorithmic Language Algol 60," *Comm. ACM,* vol. 3, no. 5, May 1960, pp. 299-314.
4. N. Chomsky, *Syntactic Structures*, Mouton & Co., 1957.
5. J.W. Backus, "The Syntax and the Semantics of the Proposed International Algebraic Language of the Zurich ACM–GAMM [Assoc. for Computing Machinery–German association for Applied Mathematics and Mechanics] Conference," *Proc. Int'l Congress for Information Processing,* 1959, pp. 125-132.
6. C.A.R. Hoare, "Report on the Elliott Algol Translator," *Computer J.,* vol. 5, no. 4, Jan. 1963, pp. 345-348.
7. T.B. Steel Jr., ed., *Formal Language Description Languages for Computer Programming*, North Holland, 1966.
8. J.V. Garwick, "The Definition of Programming Languages by their Compilers," *Formal Language Description Languages for Computer Programming,* North Holland, 1966.
9. W.V.O. Quine, *Mathematical Logic,* revised ed., Harvard Univ. Press, 1955.
10. S. Igarashi, *An Axiomatic Approach to Equivalence Problems of Algorithms with Applications*, doctoral dissertation, Tokyo Univ., 1964.
11. P. Lucas et al., *Informal Introduction to the Abstract Syntax and Interpretation of PL/I, ULD version II*, IBM TR 25.03, IBM, 1968.
12. R.W. Floyd, "Assigning Meanings to Programs," *Proc. Am. Soc. Symp. Applied Mathematics,* vol. 19, 1967, pp. 19-31.
13. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, vol. 12, no. 10, Oct. 1969, pp. 576-580, 583.
14. C.A.R. Hoare, "Procedures and Parameters: An Axiomatic Approach," *Symp. Semantics of Algorithmic Languages,* LNM 188, E. Engeler, ed., Springer-Verlag, 1971, pp. 102-116.
15. C.A.R. Hoare and M. Foley, "Proof of a Recursive Program: Quicksort," *Computer J.,* vol. 14, no. 4, Nov. 1971, pp. 391-395.
16. C.A.R. Hoare, "Towards a Theory of Parallel Programming," *Operating Systems Techniques*, Academic Press, 1972, pp. 61-71.
17. C.A.R. Hoare and M. Clint, "Program Proving: Jumps and Functions," *Acta Informatica,* vol. 1, 1972, pp. 214-224.
18. C.A.R. Hoare, "A Note on the For Statement," *BIT*, vol. 12, no. 3, 1972, pp. 334-341.
19. C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," *Acta Informatica*, vol. 2, no. 4, 1973, pp. 335-355.
20. R.L. London et al., "Proof Rules for the Programming Language EUCLID," *Acta Informatica,* vol. 10, 1978, pp. 1-26.
21. B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.
22. E.W. Dijkstra, "Go To Statement Considered Harmful," *Comm. ACM,* vol. 11, no. 3, Mar. 1968, pp. 147-148.
23. S.C. Johnson, "Lint: A C Program Checker," *UNIX 4.2 Programming Manual,* Univ. of California, Berkeley, 1984.
24. W.R. Bush, J.D. Pincus, and D.J. Sielaff, "A Static Analyser for Finding Dynamic Programming Errors," *Software Practice and Experience,* vol. 30, no. 7, June 2000, pp. 775-802.
25. E.W. Dijkstra, "A Constructive Approach to the Problem of Program Correctness," *BIT,* vol. 8, 1968, pp. 174-186.
26. E.W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.
27. C. Morgan, *Programming from Specifications*, Prentice Hall, 1990.

28. O-J. Dahl et al., *SIMULA 67 Common Base Language*, Norwegian Computer Center, 1967.
29. O-J. Dahl and C.A.R. Hoare, "Hierarchical Program Structures," *Structured Programming,* Academic Press, 1972, pp. 175-220.
30. C.A.R. Hoare, "A Structured Paging System," *Computer J.*, vol. 16, no. 3, Aug. 1973, pp. 209-215, 1973.
31. C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, vol. 1, no. 4, 1972, pp. 271-281.
32. E.W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages*, F. Genuys, ed., Academic Press, 1968.
33. P. Brinch Hansen, "Structured Multiprogramming," *Comm. ACM*, vol. 15, no. 7, July 1972, pp. 574-578.
34. C.A.R. Hoare, "Monitors, An Operating System Structuring Concept," *Comm. ACM,* vol. 17, no. 10, Oct. 1974, pp. 549-557.
35. P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Trans. Soft. Eng.,* vol. 1, no. 2, June 1975, pp. 199-207.
36. J. Gosling, W. Joy, and G. Steel, *The Java Language Specification*, Addison-Wesley, 1996.
37. C.A.R. Hoare, "The Structure of an Operating System," *Language Hierarchies and Interfaces*, LNCS 46, F.L. Bauer and K. Samelson, eds., Springer, 1976, pp. 242-265.
38. J. Welsh and D. Bustard, *Concurrent Program Structures*, Prentice Hall, 1988.
39. E.W. Dijkstra, "Guarded Commands, Nondeterminacy, and the Formal Derivation of Programs," *Comm. ACM,* vol. 18, no. 8, Aug. 1975, pp. 453-457.
40. C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM,* vol. 21, no. 8, Aug. 1978, pp. 666-777.
41. J. Stoy, *Denotational Semantics, the Scott-Strachey Approach to Programming Language Theory,* MIT Press, 1977.
42. S. Brookes and A.W. Roscoe, "An Improved Failures Model for CSP," *Proc. Pittsburgh Seminar on Concurrency*, LNCS 197, S.D. Brookes, A.W. Roscoe, and G. Winskel., eds., Springer, 1985.
43. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
44. Inmos, *Transputer Reference Manual,* Prentice Hall, 1988.
45. C.A.R. Hoare, "The Transputer and occam: A Personal Story," *Concurrency: Practice and Experience*, vol. 3, no. 4, 1991, pp. 249-264.
46. G. Jones and M. Goldsmith, *Programming in occam 2*, Prentice Hall, 1988.
47. R. Milner, *Communicating and Mobile Systems: The pi-calculus*, Cambridge University Press, 1999.
48. J.-R. Abrial, "Assigning Programs to Meanings," *Mathematical Logic and Programming Languages*, Philosophical Trans. Royal Society, series A, vol. 312, 1984.
49. B.P. Collins, J.E. Nicholls, and I.H. Sorensen, *Introducing Formal Methods, The CICS Experience with Z*, IBM TR 12.260, IBM, 1989.
50. C.B. Jones, *Software Development, A Rigorous Approach*, Prentice Hall, 1980.
51. E.C.R. Hehner, "Predicative Programming," *Comm. ACM,* vol. 27, no. 2, Feb. 1984, pp. 134-151.
52. C.A.R. Hoare and E.C.R. Hehner, "A More Complete Model of Communicating Processes, *Theoretical Computer Science,* vol. 26, no. 1-2, Sept. 1983, pp. 105-120.
53. C.A.R. Hoare and A.W. Roscoe, "Programs as Executable Predicates," *Proc. Int'l Conf. Fifth Generation Computer Systems*, Tokyo, Inst. for New Generation Computer Technology, 1984, pp. 220-228.
54. C.A.R. Hoare, "Programs are Predicates," *Mathematical Logic and Programming Languages,* Philosophical Trans. Royal Society, series A, vol. 312, 1984, pp. 475-489.
55. C.A.R. Hoare, "Programs are Predicates," *New Gen. Comp.*, vol. 38, 1993, pp. 2-15.
56. C.A.R. Hoare and H. Jifeng, *Unifying Theories of Programming*, Prentice Hall, 1998.
57. C.A.R. Hoare and P.E.Lauer, "Consistent and Complementary Formal Theories of the Semantics of Programming Languages, *Acta Informatica,* vol. 3, no. 2, 1974, pp. 135-153.



C.A.R. (Tony) Hoare is a senior researcher at Microsoft Research, which he joined in 1999. His current research interests are in program analysis, programmer productivity tools, and the challenge of an automatic verifying compiler. With a degree from Oxford University in Latin, Greek, philosophy, and ancient history, Hoare was first employed as a computer programmer in 1960. He led a team implementing an early compiler for Algol 60. He was appointed to the chair of computer science at the Queen's University, Belfast, in 1968, and moved back to Oxford University in 1977 until retirement in 1999.

Readers may contact C.A.R. (Tony) Hoare at thoare@microsoft.com.