Assertions: A personal perspective

——Kyoto Prize Lecture, 2000——

Charles Antony Richard Hoare

Computer Software (reprint)
Vol. 18, No. 4 (2001), pp. 2-17

Japan Society for Software Science and Technology (JSSST)

特別寄稿

Assertions: a personal perspective

Charles Antony Richard Hoare

Summary: An assertion is a Boolean formula written in the text of a program, at a place where its evaluation will always be true — or at least, that is the intention of the programmer. In the absence of jumps, it specifies the internal interface between the part of the program that comes before it and the part that comes after. The interface between a procedure declaration and its call is defined by assertions known as preconditions and post-conditions. If the assertions are strong enough, they express everything that the programmers on either side of the interface need to know about the program on the other side, even before the code is written. Indeed, such strong assertions can serve as the basis of a formal proof of the correctness of a complete program.

In this paper, I will describe how my early experience in industry triggered my interest in assertions and their role in program proofs; and how my subsequent research at university extended the idea into a methodology for the specification and design of programs. Now that I have returned to work in industry, I have 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.

Senior Researcher, Microsoft Research Ltd., Cambridge, England.

コンピュータソフトウェア, Vol.18, No.4(2001), pp.2–17.
[特別寄稿] 2001年5月25日受付.

Introduction.

My first job was as a programmer in the computer industry. Preliminary experience in writing machine code programs introduced me abruptly to the problem of program error. My first major assignment was the implementation of a new high-level programming language, ALGOL 60. Chomsky's context-free notation, used for defining the syntax of the language, helped greatly in forestalling error, both on the part of the implementers and of its users; it triggered intense interest in similar formal notations for specification of programming language semantics. I suggested an axiomatic method for specifying the general intentions of the language designer, while leaving the implementer free to fill in the intricate details, choosing the most efficient solution for the benefit of users of a particular machine.

In the early seventies, I moved to a university career, and encountered Floyd's assertional method for proving program correctness. I extended his method to cover all the main constructions of a sequential high-level programming language. Following Dijkstra, I always took a top-down view of the task of software construction, with assertions formulated as part of program specification, and with proofs conducted as part of program design. I hoped that this research would help to reduce the high costs of programming error, and the high risks of using computers in critical applications. But the real attraction for me was that the axioms underlying program proofs would provide at last an objective and scientific test of the quality of a proposed programming language: I suggested the principle that a language described by a small collection of obvious rules, easily applied, would be better demonstrably than one that required many rules with complex side-conditions.

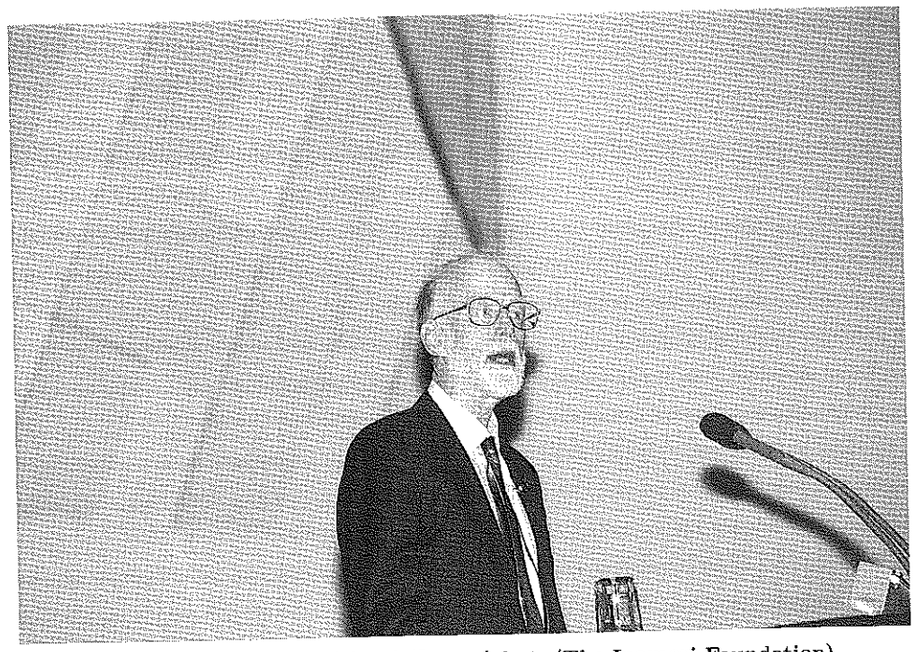
In scaling proof methods from small sequential algorithms to large software systems, it was necessary to extend the power of the assertion language. The Z specification language was developed by Abrial on the basis of Zermelo's set theory, which Frankel and others showed to be essentially adequate for expression of all concepts known to mathematics. It should therefore be adequate to express all the abstractions useful to computing, and for proof of the correctness of their representation. Dijkstra showed how to deal with non-determinism, by imagining non-deterministic choice to be exercised maliciously by a demon. Jones and his fellow-designers of VDM included initial as well as final values of program variables in the specification. The combination of these ideas were successfully tested by IBM Development Laboratories at Hursley in specifying some of the internal interfaces of a large software product, CICS.

The next challenge was to extend the technology to concurrent programs. Milner suggested that their meaning could be specified by the collection of tests which they passed. Following Popper's criterion of falsifiability, Roscoe and Brookes concentrated on failures of a test, which led them to the standard non-deterministic model for Communicating Sequential Processes. This was applied industrially by the British start-up microchip Company Inmos in the design of the programming language occam, and in the architecture of the transputer which implemented it. Finally, Hehner showed how the CSP model could be coded directly in the language of specifications, so that any kind of program, concurrent as well as sequential, can be interpreted as the strongest assertion that describes all its possible behaviours. As a result, all claims of correctness can be expressed and proved as mathematical implications between the program and its specification.

Assertions are widely used in the software industry today, primarily to detect, diagnose and classify programming errors during test. They are sometimes kept in product code to forestall the danger of crashes, and analyse them when they occur. They are beginning to be used by compilers as hints to improve optimisation. They are also beginning to be recognised by program analysis tools, to inhibit false complaints of potential error. The one purpose for which they are hardly ever used is for proof of programs. Nevertheless, assertions provide the fundamental tool for scientific investigation of professional disciplines of programming; and they show the direction for future advance in the development both of programming tools and of programming languages. There are still many issues that remain as a challenge for future research. And there are encouraging signs that the research is beginning to spread a beneficial influence on practices, tools and languages coming into use; and that this will lead to significant improvements in the quality of software products, for the benefit of their rapidly increasing numbers of users.

Experience in Industry, 1960–1968.

My first job was as a programmer for a small British computer manufacturer, Elliott Brothers of London at Borehamwood. My task was to write library programs in decimal machine code [1] for the company's new 803 computer. After a preliminary exercise which gave my boss confidence in my skill, I was entrusted with the task of implementing



Sir Charles Antony Richard Hoare (photo/The Inamori Foundation)

a new sorting method recently invented and published by Shell [2]. I really enjoyed optimizing the inner loops of my program to take advantage of the most ingenious instructions of the machine code. I also enjoyed documenting the code according to the standards laid down for programs to be delivered to customers as part of our library. 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, surely the programmers should pay the cost for removal of their own mistakes.

But not such fun was the kind of error that caused my test programs to run wild (crash); quite often, they even over-wrote 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 over-written by a number? The only way to find out was to add extra output instructions to the program, tracing its behaviour 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 forty years later, the problem of crashing programs is not altogether solved.

When I had been in my job for six months, an even more important task was given me, that of designing a new high-level programming language for the projected new and faster members of the Company's range of computers. By great good fortune, there came into my hands 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; we 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 Peter Naur's Report was that it was only twenty-one pages long. Yet it gave enough accurate information for an implementer to compile the language without any communication

with the language designers. Furthermore, a user could program in the language without any communication either with the implementers or with the designers. Even so the program worked on the very first time it was submitted to the compiler. Apart from a small error in the character codes, this is what actually happened one day at an exhibition of an Elliott 803 computer in Eastern Europe. Few languages designed since then have matched such an achievement.

Part of the credit for this success was the very compact yet precise notation for defining the grammar or syntax of the language, the class of texts that are worthy of consideration as meaningful programs. This notation was due originally to the great linguist, psychologist and philosopher Noam Chomsky [4]. It was first applied to programming languages by John Backus [5], in a famous article on the Syntax and the Semantics of the proposed International Algorithmic Language of the Zurich ACM-GAMM Conference, Paris, 1959. After dealing with the syntax, the author looked forward to a continuation article on the semantics. It never appeared: in fact it laid down a challenge of finding a precise and elegant formal definition of the meaning of programs, which inspires good research in Computer Science right up to the present day.

The syntactic definition of the language served as a pattern for the structure of the whole of our ALGOL compiler, which used a method now known as recursive descent. As a result, it was logically impossible (almost) for any error in the syntax of a submitted program to escape detection by the compiler. If a successfully compiled program went wrong, the programmer had complete confidence that this was not the result of a misprint that made the program meaningless. Chomsky's syntactic definition method was soon more widely applied to earlier and to later 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 is used to define.

In designing the machine code to be output by the Elliott ALGOL compiler [6], I took it as an overriding principle that no program compiled from the high level language could ever run wild. Our customers had to accept a significant performance penalty, because every subscripted array access had to be checked at run time 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 did not 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 the effect of every program could be inferred from the source text of the program itself, without knowing anything about the compiler or about the machine on which it was running. If only we had a formal semantics to complement the formal syntax of the language, perhaps the compiler would be able to help in detecting and averting other kinds of programming error as well

Interest in semantics was widespread. In 1964, a conference took place in Vienna on Formal Language Description Languages for Computer Programming [7]. It was attended by 51 scientists from 12 nations. One of the papers was entitled '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 would 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 counter-productive on an Elliott 803, with a word length of thirty-nine bits, to give the same numerical answers as the IBM machine, which had only thirty-six bits in a word – we could more efficiently give greater accuracy and range. 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 exploration of the concept of program correctness

I did not 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 the light of the characteristics of a particular machine architecture. I was very encouraged that this comment was applauded, and even Garwick expressed his agreement. In fact, I had mis-interpreted his title: his paper called for an abstract compiler for an abstract machine, rather than selection of an actual commercial product as standard.

The inspiration of my remark in Vienna dates back to 1952, when I went to Oxford as an undergraduate student. Some of my neighbours in College were mathematicians, and I joined them in a small unofficial night-time reading party to study Mathematical Logic from the text book by 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 which accompanies the contemplation of mathematical truth. 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 the 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 the common notations of mathematics, but they also contain a number of 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 the basic concepts of programming 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 which must be met in the design of a product, 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 more abstract approach of the pure mathematician is rewarded by the discovery that there are many different applications of the same axiom set. By

analogy, there could be many different implementations of the axiom set which defines a standard programming language. That was exactly the carefully circumscribed freedom that I wanted for the compiler writer, who has to take normal engineer's 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 school texts on algebra, but with fragments of program 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]. I showed my first pencilled draft of a paper on the axiomatic approach to Peter Lucas; he was leading a project at the IBM Research Laboratory in Vienna to give a formal definition to IBM's new programming language, known as PL/I [11]. He was attracted by the proposal, but he rapidly abandoned the attempt to apply it to PL/I. The designers of PL/I had a very operational view of what each construct of the language 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; and the axiomatic method had shown its capability of at least of revealing badness. Other evidence for the badness of PL/I was its propensity for 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; and in those days it was accepted that long-term research should take place in universities. I therefore welcomed the opportunity to take up a post as Professor of Computer Science at the Queen's University in Belfast. By a happy coincidence, as I was moving house, I came across a preprint of Robert Floyd's paper on 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. By this means, he presented an effective method of proving the total correctness of programs, not just their equality to other programs. 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 quality of the language 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 post-condition of the program, that is an assertion about the values of the variables of the program that is 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 execution of the program 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 body of the loop. 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 looked forward to the day when a verifying compiler could actually check the validity of all the assertions automatically before allowing the program to be run. This would be the ultimate solution to the problem of programming error, making it logically impossible in a running program; though I correctly predicted its achievement would be some time after I had retired from academic life, which would be in thirty year's time.

So I started my life-long project by first extending 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] [14] [15] [16] [17] [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]. Since then, the axiomatic method has been explicitly used to guide the design of 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 all 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 could be enforced by a compiler, so as to avert the risk of programming error. Restrictions that contribute to provability, I claimed, are what make a programming language good.

I was even worried that my axiomatic method was too powerful, because it could deal with jumps, which Dijkstra had pointed out to be a bad feature of the conventional programming of the day [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 now provided by program analysis tools like Lint [23] and PREfix [24], applied to less disciplined languages such as C; they identify just those constructions that would invalidate the simple and obvious proof methods, and warn 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 very difficult to look at an existing program and guess what these assertions should be. I thought this was an entirely mistaken objection. It was not 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 task of programming: the obligation of ultimate correctness should be the driving force in designing programs that were 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 proof of the program should be developed along with the program itself. Thus the most effective proofs are those constructed before the program is written. This philosophy has been beautifully illustrated in Dijkstra's own book on A Discipline of Programming [26], and in many subsequent textbooks on formal approaches to software engineering [27].

In all my work on the formalisation 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. I took as my first model of concurrency a kind of quasi-parallel programming (co-routines), which was introduced by Ole-Johan Dahl and Kristen Nygaard into Simula (and later Simula 67) for purposes of discrete event simulation [28] [29]. I knew the Simula concept of an object as a replicable structure of data, declared in a class together with the methods which 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). I suddenly realised that the purpose and criterion of correctness of the program 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 random access memory of the computer. The concept had to be represented in a complicated (but fortunately concealed) way, by storing temporarily unused data on a disk [30]. The correctness of the code 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 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 in the case of proof-driven program development, it is the obligation of correctness that should drive the design of a good programming language feature. Of course, efficiency of implementation is also important. A correct implementation of the abstraction has to 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 by Per Brinch Hansen in his efficient implementation of Concurrent PASCAL [35]. The monitor has since been adopted for the control of concurrency by the more recently fashionable language Java [36], but with extensions that prevent the use of the original simple proof rules.

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 implemented the system in an extended version of Pascal, called Pascal Plus, which they also designed and implemented [38]. We made extensive use on the **inner** statement of Simula 67, which enables the code of a user process to be embedded deep inside an envelope of code which implements the abstract resources that it uses. The semantics of the **inner** statement is described like that of the procedure call in ALGOL 60 (and inheritance in current object-oriented languages), in terms of textual copying of portions of the program inside certain other portions. Dijkstra rightly pointed out 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 level of abstraction of the program. So we spent some time together at a Marktoberdorf Summer School, exploring the underlying abstraction, and to design notations that would most clearly express it. But it took several more years of personal research, and I was still not satisfied with my progress. Inspiration eventually came from an unexpected direction.

That was the time at which the promise of very large scale integration was beginning to be realised in the form of low-cost microprocessors. In order to

multiply their somewhat modest computing power, it was an attractive prospect to connect several such machines by means of wires along which they could communicate with each other during program execution. To write programs for such an assembly of machines, a programming language would have to include input and output commands; these removed the need for an explanation by textual copying. The idea of sharing storage among microprocessors was ruled out by the expense, and without shared store, monitors are 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 [39], rather than the jump-like forking primitive made popular by C and UNIX. I also included a variant of Dijkstra's guarded command [39], to enable 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 [40]. To answer the question of the sufficiency of these few features, 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 very happy with the unification of programming concepts that I had achieved, but very dissatisfied that I had no means of proving the correctness of the programs that used them. Furthermore, there were a number of language design decisions which I left open, and 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 non-termination and of non-deterministic deadlock, which causes a computer to stop when a cycle of processes are each waiting for its neighbour. 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 Discipline of Programming [26] how to deal safely with the problem of non-determinism. 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 non-termination. Now I resolved that any acceptable proof method for CSP would have to incorporate Dijkstra's solutions.

Move to Oxford, 1977–1999.

At that time an opportunity arose to move to Oxford University, where I wanted to study the methods of denotational semantics that had been pioneered by Christopher Strachey and Dana Scott, and ably expounded in a more recent textbook by Joe Stoy [41]. Among my first research students, jointly supervised with Joe 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. Iain 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

neighbours in a network by input and output along simple wires [44]. He started up a Company called Inmos to design and make the hardware, he hired David May as its chief architect, and he hired me as a consultant on the design of 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. The level of abstraction provided by CSP gave just this assurance. For ten years or more, the transputer enjoyed commercial success and the language excited scientific interest; but today's advances in microprocessor power, storage capacity, and network communications technology favour a more dynamic model of network configuration and a buffered model of communication, which are more directly represented in more recent process algebras, like the pi-calculus [47].

Fundamental to the philosophy of top-down development of programs from their specifications is the ability of programmers to write the specifications in the first place. Obviously, these specifications have to 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, untrammelled by considerations of execution (or of efficiency), could express many important abstract concepts far more concisely than any executable language; and 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 is no conceivable way of proving a specification correct (against what specification would that be? Such a specification would have been preferable to the original). So the only hope is to make the original specification so clear and so easily understandable that it obviously describes what is wanted, and not some other thing. 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.

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; and he came to Oxford 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; and 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?

At that time, the IBM development laboratories in Hursley were supporting our research in Oxford, both financially and scientifically, in a project led by Ib Sorensen and Ian Hayes. One of their teams was responsible for the development of the Customer Information and Control System CICS, one of their most successful commercial software products; and they were 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 our new recommended top-down development method, starting with a specification in Z. This involved more work in the early stages of the project, but it gave good confidence in the soundness of the design of the new structure; and the early rigorous formalisation averted many errors that might have been troublesome at later stages in the project. In the end, the development costs, even on first use of Z, were less than on components developed in the traditional way, and the quality as perceived by the customer 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 behaviour of computer programs. The meanings of the variables, and the justification for the invariants, must be described informally in the extremely important natural-language prose that accompanies the specification. As in science, there are many common conventions: so in a schema that specifics 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. As a more specific example, when it is necessary to specify the timing properties of a program, just introduce a real-value variable called *time*. So *time'* would be the time at which a program terminates, and *time* would be when it starts. 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].

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 non-termination ensures that the composition of two schemas accurately describes the result of sequential execution of any pair of programs which satisfy those schemas. More formally, if P_1 and P_2 are programs, and if S_1 and S_2 are schemas, then the axiomatic proof rule for correctness of sequential composition of programs can be elegantly expressed

$$\frac{P_1 \text{ satisfies } S_1 \qquad P_2 \text{ satisfies } S_2}{(P_1; P_2) \text{ satisfies } (S_1; S_2)}$$

One day, Rick Hehner, on a sabbatical visit to Oxford, came into my office and spent an embarrassingly long time persuading me that something much simpler was possible [51] [52]. Just 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 behaviour 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, there is no need any longer 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 given above in the schema calculus. The proof rule displayed above 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. For the next ten years I travelled 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] [54] [55].

The first application of this wonderful insight was to solve the long-standing problem of the specification and proof of correctness of Communicating Sequential Processes. All that is needed is to introduce the observable attributes of a process, its *trace* and its *refusals*, as free variables of a Z schema. Then the various choice and parallel constructions of CSP are defined using predicate calculus as operators on schemas. This insight has inspired all my subsequent research. In a continuing collaboration with He Jifeng, we have developed a specification-oriented semantics for many other computational paradigms, including hardware and software, declarative and procedural, sequential and parallel. Even within parallel programming, there are many variations, some with distributed processing some with shared store, with dedicated channels or with shared buses, with synchronised or with buffered communication. It turns out that there is much in common between the mathematical properties of all the 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 successful doctoral student in Belfast [57].

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 the termination of the program, an assertion is generalised 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 the semantics of a program as the strongest assertion that describes all its possible behaviours, 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 towards practical application, and maybe help to progress it a bit further.

Back in Industry, 1999–

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 'do something useful and attractive, making as lit-

tle change as possible in the existing code base or its behaviour'. The details of the design are largely determined by what turns out to be possible and adequately efficient after exploration of the existing code and testing a number of possible changes by experiment. The only way of improving the correctness of the result is by debugging. The practice of specification of 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 is 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 very strongly in Microsoft code. A recent count discovered over quarter of a million of them in the code for Office. 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, in the absence of human intervention. Failure of an assertion triggers a dump of the program state, to be analysed 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, so avoiding the risk that the necessary diagnostic information is overwritten. So assertions have already found their major application, not to the proof of the correctness of programs, but to the diagnosis of their errors. They are applied as a partial solution to the problems of program crashes, which I first encountered as a new programmer in 1960. The other solution is the ubiquitous personal work-station, which reduces the turn-round for program correction from days to minutes.

Assertions are usually compiled differently for test runs and for code that is shipped to the customer. In ship code, the assertions are often omitted, to avoid the run time penalty and the confusion that would follow from an error diagnostic or a checkpoint dump in view of the customer. Ideally, the only assertions to be omitted are those that have been subjected to proof. But more practically, many teams leave the assertions in ship 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 handler for the exception makes a recovery that is sensible to the customer in the environment of use.

Assertions are also used to advantage by program analysis tools like PREfix [23]; this is being developed within Microsoft, for application to the maintenance of legacy code. The value of such tools is limited if they give so many warning messages that the programmer cannot 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. This 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 properties of the program must be maintained.

Microsoft is one of the few Companies in the world that has the motive, the skill, the resources and the courage to embark on software tool development on the requisite scale. 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 which are directly based on considerations of program correctness. And that is why program correctness has been, and still remains, a suitable topic for academic research.

Challenge

Many challenges remain. Assertional techniques for correctness still need to be extended from the simple sequential languages like PASCAL to cover the complexities of object orientation, including inheritance, over-riding, and pointer-swinging manipulations. Disciplined patterns of usage need to be formalised to simplify correctness reasoning; and the rules need to be policed by compile-time checks. The problems of concurrent programming, race conditions, deadlock and livelock need to be analysed and solved. Dynamic configuration and reconfiguration, mobile code, transactions and even exceptions are essential to modern systems software and applications; the errors to which they are most liable must be brought under control. To incorporate the results of this research into practical tools, further advances are required in automatic theorem proving and in automatic test case generation.

The ideas that crystallise from theoretical research into correctness of programs are often first subjected to practical evaluation in the context of an experimental programming language. For purposes of experiment, such a language must have a pure and simple semantics, achieved by exclusion of all extraneous features and complicating factors. To progress to the next stage of industrial use, a proposed language feature must be promulgated in the form of a design pattern, so that it can be exploited by users of existing programming languages. The pattern will include advice on the specification of interfaces, systematic coding techniques, and on the construction of test harnesses, including assertions to guard against errors. The advice needs support from program analysis tools, which will check observance of the disciplines on which correctness depends.

Finally, at intervals measured in decades rather than years, there arises an opportunity to introduce a new programming language onto the marketplace. This will never be done by taking a recent experimental language from the laboratory. The challenge of language design is to combine a multitude of features that have proved sufficiently successful that they have been tested and incorporated in all generally accepted research languages. The name of any language that aims at commercial success will be devised to reflect current buzz-words, and its syntax will be crafted to reflect the latest fashionable craze.

But there are now strong signs that the actual quality of the language is beginning to matter. A similar transformation occurred some time ago in the market place for cars, where reliability and safety once took second place to styling, chrome plating, and acceleration. In the progression from C to C++ and then to Java, each language designer has given more explicit attention to removing the

traps and insecurities of its predecessor. As with all evolutionary processes, progress has been exceedingly slow. One day, I expect the new programming language designer will learn how to use assertional methods as a design tool, to evaluate and refine the objectively evaluated quality of familiar language features, as well as averting the risks involved in the introduction of new features. To bring that day forward is surely still a worthy goal for academic research, both theoretical and experimental.

References

[1] Elliott 803 Programming Manual, Elliott Brothers (London) Ltd., Borehamwood, Herts (1960).

[2] Shell, D. : A high-speed sorting procedure, *Comm. ACM*, Vol. 2, pp. 30–32 (1959).

[3] Naur, P. (ed.) : Report on the algorithmic language ALGOL 60, *Comm. ACM*, Vol. 3, No. 5, pp. 299–314 (1960).

[4] Chomsky, N. : *Syntactic structures*, Mouton & Co, The Hague (1957).

[5] Backus, J. W. : The Syntax and the Semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference, ICIP Proceedings, Paris, pp. 125–132 (1959).

[6] Hoare, C. A. R. : Report on the Elliott ALGOL translator, *Comp J.*, Vol. 5, No. 4, pp. 345–348 (1963).

[7] Steel Jr., T. B. ed. : *Formal language description languages for computer programming*, North Holland (1966).

[8] Garwick, Jan V. : The definition of programming languages by their compilers, ibid.

[9] Quine, W. V. O. : *Mathematical Logic, Revised edition*, Harvard University Press (1955).

[10] Igarashi, S. : *An axiomatic approach to equivalence problems of algorithms with applications*, PhD. Thesis, Tokyo University (1964).

[11] Lucas, P. et al. : Informal introduction to the abstract syntax and interpretation of PL/I, ULD version II, IBM TR 25.03 (1968).

[12] Floyd, R. W. : Assigning meanings to programs, Proc. Am. Soc. Symp. Appl. Math., Vol. 19, pp. 19–31 (1967).

[13] Hoare, C. A. R. : An axiomatic basis for computer programming, *Comm. ACM*, Vol. 12, No. 10, pp. 576–80, 583 (1969).

[14] Hoare, C. A. R. : Procedures and parameters: an axiomatic approach, LNM 188, Springer Verlag (1971).

[15] Hoare, C. A. R. and Foley, M. : Proof of a recursive program: QUICKSORT, *Comput. J.*, Vol. 14, pp. 391–395 (1971).

[16] Hoare, C. A. R. : Towards a theory of parallel programming, in *Operating Systems Techniques*, Academic Press (1972).

[17] Hoare, C. A. R. and Clint, M. : Program proving: jumps and functions, *Acta Informatica*, Vol. 1, pp. 214–224. (1972).

[18] Hoare, C. A. R. : A note on the **for** statement, *BIT*, Vol. 12, No. 3, pp. 334–341 (1972).

[19] Hoare, C. A. R. and Wirth, N. : An axiomatic definition of the programming language PASCAL, *Acta Informatica*, Vol. 2, No. 4, pp. 335–355 (1973).

[20] London, R. L. et al. : Proof rules for the programming language EUCLID, *Acta Informatica*, Vol. 10, pp. 1–26 (1978).

[21] Meyer, B. : *Object-oriented software construction (2nd ed.)*, Prentice Hall PTR (1997).

[22] Dijkstra, E. W. : go to statement considered harmful, *Comm. ACM*, Vol. 11, pp. 147–148 (1968).

[23] Bush, W. R., Pincus, J. D. and Sielaff, D. J. : A static analyser for finding dynamic programming errors, *Software Practice and Experience* Vol. 30, pp. 775–802 (2000).

[24] Johnson, S. C. : Lint: a C program checker, UNIX Prog. Man. 4.2 UC Berkeley (1984).

[25] Dijkstra, E. W. : A constructive approach to the problem of program correctness. *BIT*, Vol. 8, pp. 174–186 (1968).

[26] Dijkstra, E. W. : *A discipline of programming*, Prentice Hall (1976).

[27] Morgan, C. : *Programming from Specifications*, Prentice Hall International (1990).

[28] Dahl, O-J. et al. : SIMULA 67 common base language, Norwegian Computer Centre (1967).

[29] Dahl, O-J. and Hoare, C. A. R. : Hierarchical program structures, in *Structured Programming*, Academic Press, pp. 175–220 (1972).

[30] Hoare, C. A. R. : A structured paging system, *Comp. J.*, Vol. 16, No. 3, pp. 209–215 (1973).

[31] Hoare, C. A. R. : Proof of correctness of data representations, *Acta Informatica*, Vol. 1, No. 4, pp. 271–281 (1972).

[32] Dijkstra, E. W. : Cooperating sequential processes, in *Programming Languages*, Genuys, F., ed., Academic Press (1968).

[33] Hansen, P. B. : Structured multiprogramming, *Comm. ACM*, Vol. 15, No. 7, pp. 574–578 (1972).

[34] Hoare, C. A. R. : Monitors, an operating system structuring concept, *Comm. ACM*, Vol. 17, No. 10, pp. 549–557 (1974).

[35] Hansen, P. B. : The programming language Concurrent Pascal, *IEEE Trans. Soft. Eng.*, Vol. 1, No. 2, pp. 199–207 (1975).

[36] Gosling, J., Joy, W. and Steel, G. : *The Java Language Specification*, Addison-Wesley (1996).

[37] Hoare, C. A. R. : The structure of an operating system, in Springer LNCS 46, pp. 242–265 (1976).

[38] Welsh, J. and Bustard, D. : *Concurrent Program Structures*, Prentice Hall International.

[39] Dijkstra, E. W. : Guarded commands, nondeterminacy, and the formal derivation of programs, *Comm. ACM*, Vol. 18, pp. 453–457 (1975).

[40] Hoare, C. A. R. : Communicating Sequential Processes, *Comm. ACM*, Vol. 21, No. 8, pp. 666–777 (1978).

[41] Stoy, J. : *Denotational semantics, the Scott-Strachey approach to programming language theory.* MIT Press, (1977).

[42] Brookes, S. and Roscoe, A. W. : An improved failures model for CSP, in Springer LNCS 197 (1985).

[43] Hoare, C. A. R. : *Communicating Sequential Processes*, Prentice Hall International (1985).

[44] INMOS Limited : Transputer reference manual, Prentice Hall International (1988).

[45] Hoare, C. A. R. : The transputer and occam: a personal story, *Conc. Pract. and Exp.*, Vol. 3, No. 4, pp. 249–264 (1991).

[46] Jones, G. and Goldsmith, M. : *Programming in occam 2.* Prentice Hall International.

[47] Milner, R. : *Communicating and mobile systems: the pi-calculus*, Cambridge University Press (1999).

[48] Abrial, J.-R. : Assigning programs to meanings, in *Mathematical Logic and Programming Languages*, Philosophical Transactions of the Royal Society, Series A, Vol. 31 (1984)

[49] Collins, B. P., Nicholls, J. E. and Sorensen, I. H. : Introducing formal methods, the CICS experience, IBM TR 260, Hursley Park, Winchester (1989).

[50] Jones, C. B. : *Software Development, A Rigorous Approach*, Prentice Hall International (1980).

[51] Hehner, E. C. R. : *The logic of programming*, Prentice Hall International (1984)

[52] Hoare, C. A. R. and Hehner, E. C. R. : A more complete model of communicating processes, *Theor. Comp. Sci.*, Vol. 26, Nos. 1–2., pp. 105–120 (1983).

[53] Hoare, C. A. R. and Roscoe, A. W. : Programs as executable predicates, in 5th Gen. Comp. Sys., Tokyo, ICOT (1984).

[54] Hoare, C. A. R. : Programs are predicates, in *Mathematical Logic and Programming Languages*, Phil. Trans. Royal Soc. Ser. A, Vol. 31 (1984).

[55] Hoare, C. A. R. : Programs are predicates, New Gen. Comp., Vol. 38, pp. 2–15 (1993).

[56] Hoare, C. A. R. and Jifeng, H. : *Unifying theories of programming*, Prentice Hall International (1998).

[57] Hoare, C. A. R. and Lauer, P. E. : Consistent and complementary formal theories of the semantics of programming languages, *Acta Informatica*, Vol. 3, No. 2, pp. 135–153 (1974).