The Early Search for Tractable Ways of Reasoning about Programs

Cliff B. Jones
University of Newcastle-upon-Tyne

This article traces the history—up to around 1990—of research on reasoning about programs. The main focus is on sequential imperative programs but some comments are made on concurrency. The main thesis is that the idea of reasoning about programs has been around since they were first written; the search has been to find tractable methods.

A program can only be judged as correct or not with respect to some independent specification of what it should achieve. A simple calculation shows that testing alone cannot ensure the correctness of even relatively simple programs: A program with n simple two-way (forward pointing) decision points has a maximum of 2^n paths; which shows that for programs with upward of a thousand branch points, all their paths cannot be tested; in fact, they will never all be used. But there is no general way of determining which paths will be used, and a "bug" occurs where a path is not designed correctly.

To avoid bugs, some technique other than testing must be used to establish that software satisfies its specification. Fortunately, under assumptions discussed in this article, it is possible to reason about computer programs. The ideal is that a relatively short specification should be the basis for a proof that a putative implementation satisfies its specification. But proofs can also contain errors and there is a correlation between complexity and the risk of errors. This article discusses two attempts to reduce the risk of accepting invalid proofs, followed by a brief discussion of structuring developments to reduce complexity. The article concludes with a review of the use of proof support tools.

This article traces the most important steps in the history of research on program verification.[1] Its central thesis is that the need to reason about programs was apparent from their first creation; the research challenge has been to find tractable methods.

Proofs about sequential algorithms

Imperative programs can be considered in terms of the effects they have on a computer that executes them. Such operational thinking has severe limitations and—by making the human into a slow imitation of a computer—does not yield deep understanding. Interestingly, Alan Mathison Turing in his classic paper on the *Entscheidungsproblem*[2] *[Decision problem]* introduced the idea of a Turing machine as a thought experiment to prove a deep result about formal systems.

In writing a program, errors are easily made; the larger the program, the greater the risk that testing will not detect errors. As this section shows, pioneers of computer programming were aware of the need to reason about programs to ensure that they have some desired properties. In most cases, the property sought was to show that a program satisfied a specification, that is, had no errors. The search has been for tractable notations for specifying and reasoning about programs.

I now trace the main line of development taking a key publication of Charles Antony Richard Hoare[3] as a pivotal point.

Pre-Hoare

The possibility of reasoning about computer programs was evident to some of electronic computing's pioneers.[4] Herman Heine Goldstine and John von Neumann wrote a paper[5] explaining how "assertion boxes" can be used to record the reasons for believing that a series of "operation boxes" have a particular effect.

The paper begins with a discussion that shows why Goldstine and von Neumann believed that the task of coding was nontrivial:

> The actual code for a problem is that sequence of coded symbols (expressing a sequence of words, or rather of half words and words) that has to be placed into the Selectron memory in order to cause the machine to perform the desired and

 Published by the IEEE Computer Society 1058-6180/03/$17.00 © 2003 IEEE

planned sequence of operations, which amounts to solving the problem in question. Or to be more precise: This sequence of codes will impose the desired sequence of actions on *C* by the following mechanism: *C* scans the sequence of codes, and effects the instructions, which they contain, one by one. If this were just a linear scanning of the coded sequence, the latter remaining throughout the procedure unchanged in form, then matters would be quite simple. Coding a problem for the machine would merely be what its name indicates: Translating a meaningful text (the instructions that govern solving the problem under consideration) from one language (the language of mathematics …), in which the planner will have conceived the problem, or rather the numerical procedure by which he has decided to solve the problem) into another language (that one of our code).

This, however, is not the case. We are convinced, both on general grounds and from our actual experience with the coding of specific numerical problems, that the main difficulty lies just at this point.[6]

They then move on to indicate the direction of their proposal:

Our problem is, then, to find simple, step-by-step methods, by which these difficulties can be overcome. Since coding is not a static process of translation, but rather the technique of providing a dynamic background to control the automatic evolution of a meaning, it has to be viewed as a logical problem and one that represents a new branch of formal logics. We propose to show in the course of this report how this task is mastered.[7]

Their basic design approach is to plan from a flowchart. After describing operation boxes (and substitution boxes), the key concept of *assertions* comes in the following text:

Next we consider the changes, actually limitations, of the domains of variability of one or more bound variables, individually or in their interrelationships. It may be true, that whenever *C* actually reaches a certain point in the flow diagram, one or more bound variables will necessarily possess certain specified values, or possess certain properties, or satisfy certain relations with each other. Furthermore, we may, at such a point, indicate the validity of these limitations. For this reason we will denote each area in which the validity of such limitations is being asserted, by a special box, which we call an *assertion box*.[8]

The description of how the consistency of the operation/assertion boxes is checked is interesting. One example is described as follows:

The interval in question is immediately preceded by an assertion box: It must be demonstrable, that the expression of the field is, by virtue of the relations that are validated by this assertion box, equal to the expression which is valid in the field of the same storage position at the constancy interval immediately preceding this assertion box. If this demonstration is not completely obvious, then it is desirable to give indications as to its nature: The main stages of the proof may be included as assertions in the assertions box, or some reference to the place where the proof can be found may be made either in the assertion box or in the field under consideration.[9]

Then, after a discussion of approximation processes and round-off errors, they write:

It is difficult to avoid errors or omissions in any but the simplest problems. However, they should not be frequent, and will in most cases signalize themselves by some inner maladjustment of the diagram, which becomes obvious before the diagram is completed. The flexibility of the system … is such that corrections and modifications of this type can almost always be applied at any stage of the process without throwing out of gear the procedure of drawing the diagram, and in particular without creating a necessity of 'starting all over again.'[10]

For reasons that become clear below, it is also interesting to quote an earlier part of this paper on "induction":

The reason why *C* may have to move several times through the same region in the Selectron memory is that the operations that have to be performed may be repetitive. Definitions by induction (over an integer variable); iterative processes (like successive approximations); … To simplify the nomenclature, we will call any simple iterative process of this type an *induction* or a *simple induction*. A multiplicity of such iterative processes, superposed upon each other or crossing each other will be called a *multiple induction*. …

To conclude, we observe that in the course of circling an induction loop, at least one variable (the induction variable) must change, and that this variable must be given its initial value upon entering the loop. Hence, the junction before the alternative box of an induction loop must be preceded by substitution boxes along both paths that lead to it: along the loop and along the path that leads to the loop. At the exit from an induc-

> tion loop the induction variable usually has a (final) value which is known in advance, or for which at any rate a mathematical symbol has been introduced. This amounts to a restriction of the domain of variability of the induction variable, once the exit has been crossed—indeed, it is restricted from then on to its final value, i.e. to only one value.[11]

Goldstine and von Neumann's paper shows that the authors were not only concerned with the problem of correctness but that they also had a clear idea that it was possible, and desirable, to reason about programs. The essential point is that the possibility to add some form of assertions that were separate from, and served to discuss the effect of, the operations of a program was evident at the beginning of the work on writing programs. A similar observation can be made about another milestone.

The paper Turing presented at a Cambridge, England, conference begins

> How can one check a routine in the sense of making sure that it is right?
>
> In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.[12]

This paper provides a beautiful account in just three typed (foolscap) pages. The quotation above makes Turing's motivation clear; the paper provides an example of an answer to its opening question. The aim is to reason about a program in general, not just to reduce errors by detecting exceptional cases with hand tests.

Turing begins with an analogy between the carries in an addition and the assertions that decorate his flow diagrams: both decompose the task of checking. The programming example tackled is computing factorial ("without the use of a multiplier, multiplication being carried out by repeated addition"), which became a standard example for demonstrating ways of reasoning about programs. His flow diagram and annotations are shown in Figure 1. (Turing used the box notation for factorial; this has been changed below to the more familiar $n!$ notation.) He explains the assertions as follows:

> At a typical moment of the process we have recorded $r!$ and $sr!$ for some r,s. We can change $sr!$ to $(s + 1)\, r!$ by addition of $r!$. When $s = r + 1$ we can change r to $r + 1$ by a transfer. Unfortunately there is no coding system sufficiently generally known to justify giving the routine for this process in full, but the flow diagram given in Fig. 1 [**Ed. note:** the upper part of Figure 1] will be sufficient for illustration.
>
> Each 'box' of the flow diagram represents a straight sequence of instructions without changes of control. The following convention is used:
>
> (i) a dashed letter indicates the value at the end of the process represented by the box.
>
> (ii) an undashed letter represents the initial value of a quantity.
>
> One cannot equate similar letters appearing in different boxes, but it is intended that the following identifications be valid throughout
>
> s content of line 27 of store
> r content of line 28 of store
> n content of line 29 of store
> u content of line 30 of store
> v content of line 31 of store
>
> it is also intended that u be $sr!$ or something of the sort e.g. it might be $(s + 1)\, r!$ or $s\,(r - 1)!$ but not e.g. $s^2 + r^2$.
>
> In order to assist the checker, the programmer should make assertions about the various states that the machine can reach. These assertions may be tabulated as in Fig. 2 [**Ed. note:** lower part of Figure 1]. Assertions are only made for the states when certain particular quantities are in control, corresponding to the ringed letters in the flow diagram. One column of the table is used for each such situation of the control. Other quantities are also needed to specify the condition of the machine completely: in our case it is sufficient to give r and s. The upper part of the table gives the various contents of the store lines in the various conditions of the machine, and restrictions on the quantities s,r (which we may call inductive variables). The lower part tells us which of the conditions will be the next to occur.
>
> The checker has to verify that the columns corresponding to the initial condition and the stopped condition agree with the claims that are made for the routine as a whole. In this case the claim is that if we start with control in condition A and with n in line 29 we shall find a quantity in line 31 when the machine stops which is $n!$ (provided this is less than 2^{40}, but this condition has been ignored).
>
> He has also to verify that each of the assertions in the lower half of the table is correct. In doing this, the columns may be taken in any order and quite independently. Thus for column B the checker would argue: "From the flow diagram we see that after B the box $v' = u$ applies.

From the upper part of the column for B we have $u = r!$. Hence $v' = r!$ i.e. the entry for v i.e. for line 31 in C should be $r!$. The other entries are the same as in B."[12]

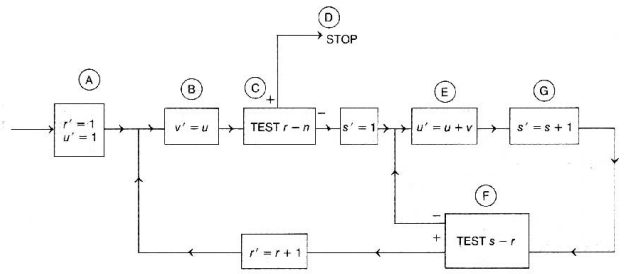
Turing's programming language is a hindrance—although the idea of using primed ("dashed") versions of identifiers is returned to later.

Turing also addresses the question of termination:

> Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. To the pure mathematician it is natural to give an ordinal number. In this problem the ordinal might be $(n-r)\omega^2 + (r-s)\omega + k$. A less highbrow form of the same thing would be to give the integer $2^{80}(n-r) + 2^{40}(r-s) + k$. Taking the latter case and the steps from B to C there would be a decrease from $2^{80}(n-r) + 2^{40}(r-s) + 5$ to $2^{80}(n-r) + 2^{40}(r-s) + 4$. In the step from F to B there is a decrease from $2^{80}(n-r) + 2^{40}(r-s) + 1$ to $2^{80}(n-r-1) + 2^{40}(r+1-s) + 5$.
>
> In the course of checking that the process comes to an end the time involved may also be estimated by arranging that the decreasing quantity represents an upper bound to the time till the machine stops.[12]

Compared to later work by Robert W. Floyd and others, the language of assertions is clearly limited. But the key idea of logical statements, which relate values of variables, is present.

Given the respective dates of Goldstine and von Neumann's work[5] and Turing's work,[12] it is tempting to speculate whether Turing knew of the earlier work. Turing visited von Neumann around 1947[13] and it's likely that Turing's work would have been discussed. Although he had a reputation for working everything out for himself, it is at least possible that Turing presented—in 1949—his own refinement of Goldstine and von Neumann's earlier ideas. This is, of course, pure speculation and would be unwise if any significant part of Turing's reputation depended on this rather nonce presentation. One link that seems to support this speculation is the use of the term inductive variable by Goldstine, von Neumann, and Turing. As pointed out by mathematician and physicist Douglas Hartree in the discussion that followed Turing's talk, this is probably not the most obvious of choices of terminology, and it is



(b)

ORAGE CATION	(INITIAL) (A) k = 6	(B) k = 5	(C) k = 4	(STOP) (D) k = 0	(E) k = 3	(F) k = 1	(G) k = 2
27					s	s + 1	s
28		r	r		r	r	r
29	n	n	n	n	n	n	n
30		|r	|r		s|r	(s + 1)|r	(s + 1)|r
31		|r	|r	|n	|r	|r	|r
	TO (B) WITH r' = 1 u' = 1	TO (C)	TO (D) IF r = n TO (E) IF r < n		TO (G)	TO (B) WITH r' = r + 1 IF s ≥ r TO (E) WITH s' = s + 1 IF s < r	TO (F)

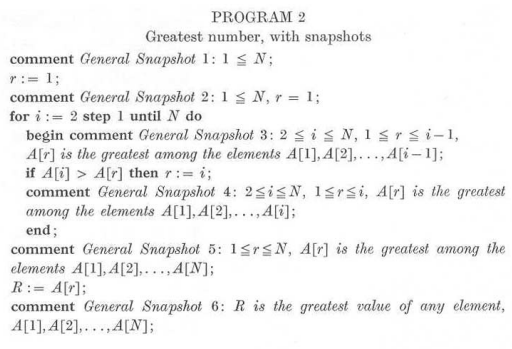
Figure 1. Turing's proof of a factorial routine. (From *IEEE Annals*, vol. 6, no. 2, April 1984.)

unlikely that both pioneers hit upon it purely by coincidence.

Neither Goldstine and von Neumann's work[5] nor Turing's work[12] appears to have been known to those who most influenced subsequent research on program verification. These early papers indicate that tractability was the research challenge: The basic idea that programs could be the subject of formal arguments was apparent early in the history of programming. In fact, mathematicians coming to computing would have found that the key difference between the notations of mathematics and programming languages was that the mathematics notation had been designed with manipulation in mind whereas it was difficult to reason about programs because programming languages were designed chiefly to facilitate translation into efficient machine code.[14]

There is no compelling explanation of why more than a decade elapsed before the next landmark. Possible reasons include hardware developments, and a period of optimism that the development of programming languages would make the expression of programs so clear as to eliminate errors. In fact, as the hardware became less restrictive, the programming task became much more complex. There were also far more programmers (to make mistakes) at the end of this period. Perhaps most tellingly, the hardware developments made it possible to implement systems for which the inadequacy of testing alone became increasingly evident.

At the May 1961 Western Joint Computer Conference, John McCarthy—the US professor



PROGRAM 2
Greatest number, with snapshots

```
comment General Snapshot 1: 1 ≦ N;
r := 1;
comment General Snapshot 2: 1 ≦ N, r = 1;
for i := 2 step 1 until N do
  begin comment General Snapshot 3: 2 ≦ i ≦ N, 1 ≦ r ≦ i−1,
  A[r] is the greatest among the elements A[1],A[2],...,A[i−1];
  if A[i] > A[r] then r := i;
  comment General Snapshot 4: 2≦i≦N, 1≦r≦i, A[r] is the greatest
  among the elements A[1],A[2],...,A[i];
  end;
comment General Snapshot 5: 1≦r≦N, A[r] is the greatest among the
elements A[1],A[2],...,A[N];
R := A[r];
comment General Snapshot 6: R is the greatest value of any element,
A[1],A[2],...,A[N];
```

Figure 2. An example of Naur's "General Snapshots." (Courtesy of *BIT*.)

who was an important figure in artificial intelligence—issued a clarion call to investigate a "Mathematical Theory of Computation." His 1963 paper, a slightly extended version of his 1961 presentation, includes the provocative sentences:

> It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.[15]

One impact of this call was to stimulate work (in which McCarthy played a significant early part) on formally describing the semantics of programming languages. There are some difficult issues in formulating such descriptions, but the reasons for tackling the problem should be clear: One cannot reason about programs in a language whose semantics are unknown; proving a program has certain properties from a language description is fruitless unless the compiler for the language faithfully reflects that description. Language semantics cannot be completely separated from program verification; the "Language semantics" section outlines the major issues. McCarthy's attention was more on reasoning about recursive functions than the imperative programs that are the focus here: McCarthy describes "recursion induction"[15] and the Lisp language[16] but later[17] shows a link between recursive functions and "Algolic Programs." This last citation also includes a clear statement of goals:

> Primarily, we would like to be able to prove that given procedures solve given problems … Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. For this to be possible, formal systems are required in which it is easy to write proofs.[17]

There is one problem that McCarthy only slightly touches on. If computer arithmetic is not exact, what are the rules for reasoning about evaluation of expressions? Adriaan van Wijngaarden, who was for many years the father figure of Dutch computer science,[18] presented a paper in 1964—and later published it[19]—which contains a careful discussion of the problems of reasoning about finite computer arithmetic and sketches axioms that might support proofs.[20]

Tantalizingly, van Wijngaarden was present at the 1949 Cambridge conference but he makes no attempt to link his quite separate contribution to that of Turing. (If only I had asked while he was alive: My checks with colleagues such as Jaco de Bakker and Michel Sintzoff have not turned up any memories of verbal references.) In content—although in ignorance of the Turing source—this link was made by Hoare five years after van Wijngaarden's talk.

The year 1966 saw a crucial step forward: The papers by Floyd and Peter Naur have a major influence on subsequent work. It appears that Naur and Floyd developed their ideas independently of each other. Naur's paper[21] includes a final note: "Similar concepts have been developed independently by Robert W. Floyd (unpublished paper, communicated privately)." This is presumably a reference to the mimeographed version of Floyd's paper dated 20 May 1966; the work normally cited[22] is in the proceedings of an April 1966 conference. Floyd (private communication, July 1991) confirms that by the time he and Naur met in summer 1966 they both had their ideas worked out and their publications were not affected. Floyd's paper has been more influential than Naur's largely because Floyd's presents a more formal foundation, but it is clear that these independent contributions were both of great significance to research on program verification.

Naur's "General Snapshots"[21] are written as comments in the text of Algol 60 programs and are clear statements about the relationships between variables without being expressions in a formal language (see Figure 2). The arguments as to why they should be believed are similarly carefully argued rather than being in some stated formal logic. Naur's paper opens with

> It is a deplorable consequence of the lack of influence of mathematical thinking on the way in which computer programming is currently being pursued, that the regular use of systemat-

> ic proof procedures, or even the realization that such proof procedures exist, is unknown to the large majority of programmers.[21]

In spite of this clear statement, it is obvious from later writings[23] that Naur views proof as one of a program designer's many tools; he is not interested in formality for its own sake. This might account for the controversy sparked by Naur—see the "Language semantics" section.

Anyone wishing to gain insight into the development of research on program verification should read Floyd.[22] He describes the challenge faced by all authors discussed here as proving properties of the form, "If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2." His summary of the approach taken is

> the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever the connection is taken.[22]

So Floyd's method is based on annotating a flow chart with assertions (propositions) that relate values of variables. For example, in Floyd's sample program that computes integer division by successive subtraction (see Figure 3), one finds

$$0 \le R < Y \wedge X \ge 0 \wedge X = R + Q * Y$$

to describe the situation where Q is the quotient and R the remainder of dividing positive integers X by Y. In addition to the assertions in Figure 3 that are concerned with the correct outcome, the ordered pairs are annotations that provide a termination argument. Floyd clearly saw this as essential; not all subsequent authors agreed.

The generality of using first-order predicate calculus for assertions and the explicit role of loop invariants are key contributions. Floyd gives precise verification conditions that ensure that the assertions correspond to the statements in his flow diagrams. In his 1967 paper,[22] Floyd looks at "strongest verifiable consequents." Tackled in this way, one starts with an assertion written before a statement and tries to conclude the strongest assertion that is true after the statement is executed. This viewpoint leads to a complicated rule for the assignment statement whose consequent uses an existential quantifier. Floyd also discovered the simpler backward rule. This rule was used in the work of Jim King, and David Charles Cooper was, according to Hoare, responsible for informing Hoare of the backward rule that he wisely adopted in his key 1969 paper.

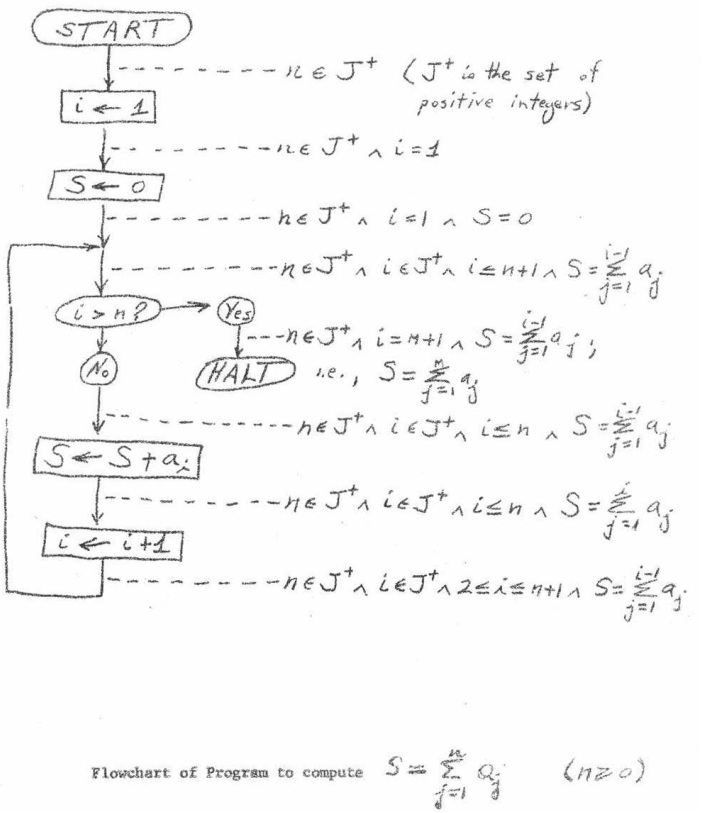


Figure 3. An example of Floyd's interpreted programs. (Source: Cliff Jones, personal files.)

Floyd's landmark paper includes a discussion of the verification conditions required for an Algol subset and recognizes the place of "general axioms," which should be true of any semantic definition. These can be compared with Edsger Dijkstra's "healthiness conditions" for "predicate transformers," discussed in the "Stepwise design" section.

As hinted at in Floyd's title, ("Assigning Meaning to Programs"), the paper's main thrust is as follows:

> The establishment of formal standards for proofs about programs in languages which admit assignments, transfer of control, etc., and the proposal that the semantics of a programming language may be defined independently of all processors for that language, by establishing standards of rigour for proofs about programs in the language, appear to be novel, although McCarthy has done similar work for program-

> ming languages based on evaluation of recursive functions.[22]

Floyd was unaware of Turing's work; the most important difference between Floyd's method and Turing's is that Floyd allows arbitrary logical expressions as assertions and can thus relate the values of variables to one another in more complicated ways than in Figure 1 where one appears to be constrained to have explicit expressions for the values of variables.[24] It would, for example, not be clear how to write quantified expressions in Turing's system. Although Goldstine and von Neumann[5] mention more general expressions, this would present problems for their system as well. Most important is the fact that Floyd is the first to offer formal rules for checking verification conditions.

There can be no doubt about the huge effect of Floyd's paper. Even given that Floyd saw the paper as a contribution to semantics rather than as a program proof method, it is interesting to record Floyd's comment that "These modes of proof of correctness and termination are not original; they are based on ideas of Perlis and Gorn, and may have made their earliest appearance in an unpublished paper by Gorn."[22] This reference has been traced (with some difficulty and only thanks to Dick Hamlet and Ralph London) to Gorn.[25] It would be fair to say that Gorn argues about correctness from a flow chart but does not present a systematic way of handling assertions. He does, however, formulate an interesting induction rule: "Every function or property of a set of storage positions which remains unchanged after each instruction of a program remains unchanged at the conclusion of that program."[25]

Hoare's axioms

The most widely cited paper mentioned here must be Hoare's "An Axiomatic Basis for Computer Programming."[26] Tracing the evolution of this paper is fascinating. Tony Hoare attended the 1964 IFIP Working Conference on Formal Language Description Languages at Baden bei Wien (Austria), and although he did not give a talk, he commented on "the need to leave languages undefined"[27] that presages one of the key points in his 1969 paper. At the meeting of IFIP Working Group 2.1 that followed the Baden conference, Hoare discussed the definition of functions via their properties (for example, the result of *abs* must be nonnegative and equal to either its argument or the negation thereof). In 1965, Hoare attended, with other members of the European Computer Manufacturers Association PL/I standardization group (TC10), a course on the language semantics work then being pursued at the IBM Laboratory in Vienna. He stayed at the Imperial Hotel on whose notepaper a first sketch of his axiomatic work was written. This event precedes the Naur/Floyd publications. A two-part draft (dated December 1967) of what was to evolve into his "Axiomatic Basis" paper[26] was written during a brief period of employment at the National Computer Center in Manchester. In one part of this privately circulated typescript, Hoare axiomatized execution traces by means of a partial order and clearly stated the goals of what was to become the "Axiomatic Basis." Although the objectives were clear in the 1967 paper, there was no coherent notion of how to axiomatize the programming language concepts discussed.

On arrival in Belfast in October 1968, Hoare stumbled upon the mimeographed draft of Floyd's paper. (Peter Lucas recalls having sent Hoare—in response to one of his earlier drafts—a copy of the preprint of Floyd's paper; it could be this was what was stumbled upon.) This had a major impact on Hoare's paper. A further two-part draft (dated December 1968) reflects Floyd's ideas and strongly resembles the final journal version.[26] The first part on data manipulation is clearly influenced by van Wijngaarden's ideas about providing axioms for data such as overflow arithmetic; the part on program execution makes the crucial step to what have become known as Hoare-triples. In his acknowledgments, Hoare writes (the papers mentioned above by Floyd, Naur, and van Wijngaarden are all cited):

> The suggestion that the specification of proof techniques provides an adequate formal definition of a programming language first appears in [Ref. 22]. The formal treatment of program execution presented in this paper is clearly derived from Floyd. The main contributions of the author appear to be: (1) a suggestion that axioms may provide a simple solution to the problem of leaving certain aspects of a language undefined; (2) a comprehensive evaluation of the possible benefits to be gained by adopting this approach both for program proving and for language definition.[26]

One reason for the impact of Hoare's paper is its clarity of exposition. Hoare grasped the significance of Floyd's contribution and made a key change in its presentation. Hoare-triples contain a pre- and postcondition placed around a piece of program. (In his paper, Hoare placed the braces around the program constructs; the move to bracket both assertions by

braces is indicative of their role as commentary for the program text.) Thus

$$\{x = r + y * (1 + q)\}\ q := 1 + q\ \{x = r + y * q\}$$

can be read operationally as stating that if the assignment statement $q := 1 + q$ is executed in any state where the values are such that

$$x = r + y * (1 + q),$$

then—providing execution terminates—the state after the execution of the assignment will be such that its values make the assertion $x = r + y * q$ true. This example is taken from Hoare's development[26] of an Algol-like version of the division by successive subtraction example used by Floyd.[22] It illustrates a number of points. The inference is valid although the assignment statement uses q as both a right-hand-side value (extracting a value from the initial state) and as a left-hand-side value (indicating which variable should change in the final state). This particular inference is an instance of a simple and perspicuous schema: Hoare presents his axiom of assignment (D0) thus

$$\vdash \{P_0\}\ x := f\ \{P\}$$

where x is a variable identifier; f is an expression; P_0 is obtained from P by substituting f for all occurrences of x. (Strictly, the substitution should only be for free occurrences of x.)

As Hoare points out, the axiom schema "is described in purely syntactic terms, and it is easy to check whether any finite text conforms to the pattern."[26] As mentioned, this backward rule for assignment is far simpler than the forward verification condition.[22] Hoare heard of the new rule via Cooper, who had been on sabbatical in the Carnegie Institute of Technology (later Carnegie Mellon University) and gave a talk in Belfast on his return. The axiom schema is only valid under certain assumptions—there are subtle points about subscripted variables.[28] To mention just one of the restrictions, consider the possibility that two identifiers denote the same location. In such cases, instances of the axiom schema may not correspond with any possible interpretation. One can decry languages that permit this but such sharing occurs with call-by-name in Algol 60 or call-by-variable in Pascal and has its use in writing efficient programs.

Hoare[26] gives general rules of consequence (D1) that justify the strengthening of preconditions or weakening of postconditions:

$$\text{If } \vdash \{P\}\ Q\ \{R\} \text{ and } \vdash R \Rightarrow S \text{ then } \vdash \{P\}\ Q\ \{S\}$$
$$\text{If } \vdash \{P\}\ Q\ \{R\} \text{ and } \vdash S \Rightarrow P \text{ then } \vdash \{S\}\ Q\ \{R\}$$

Rules are also given for composition so that from two instances of D0, one can also conclude

$$\{x = r + y * q \wedge y \leq r\}\ r := r - y;\ q := 1 + q\ \{x = r + y * q\}$$

The rule for iteration (D3) is particularly interesting:

$$\text{If } \vdash \{P \wedge B\}\ S\ \{P\} \text{ then } \vdash \{P\} \text{ while } B \text{ do } S\ \{\neg B \wedge P\}$$

This clearly shows the role of the invariant P in constructs like while statements: it plays the same part as an induction hypothesis in inductive proofs. D3 can be used to establish

$$\{x = r + y * q\} \text{ while } y \leq r \text{ do } (r := r - y;\ q := 1 + q)\ \{\neg\, y \leq r \wedge x = r + y * q\}$$

The iteration rule does not establish termination. Hoare points out in a section on reservations that "the notation $\{P\}\ Q\ \{R\}$ should be interpreted as 'provided that the program successfully terminates, the properties of its results are described by R'.'"[26] This result is often referred to as partial (as opposed to total) correctness.

One invaluable contribution made by these rules is the liberating effect of not needing to reason operationally about program execution: A program proof is presented as a sequence of lines mediated by the axioms and inference rules. Although it was possible to view Floyd's flow diagrams statically, the temptation to think operationally was greater. Like many important steps in the development of mathematical notation, presenting a proof with Hoare-triples encourages clear thinking. Another crucial aspect of Hoare-triples is that they are "compositional" in a way that Floyd's flow diagrams were not. This point is explored in the "Formal development methods" section.

Subsequent papers[28-32] tackle features of programming languages that were identified as "areas which have not been covered" in Hoare's 1969 paper. Notably, Hoare and Niklaus Wirth[33] attempted to provide axioms for a significant portion of Pascal. This was an obvious line of research given Hoare's stated aim to use this style of axioms for language definitions.[34] But time has shown, at least for languages designed without the constraint that they should have an

axiomatic semantics, the task of providing a complete axiomatization is impractical. The languages Euclid[35] and Gypsy[36] were designed with axiomatic definition in mind, but the most one would aspire to today would be to design a language with a denotational or operational semantics (see the "Language semantics" subsection) as the main guide with considerations of proof as an additional insurance of tractability. Much progress has been made from the stage when languages were designed as (relatively minor) abstractions from the architecture of the hardware on which the object programs had to run. Even under the narrowest interpretation of Dijkstra's famous letter to the editor[37] as being just about rules to limit the way programs should be written, the avoidance of goto statements can be seen as a step toward tractability of programs for reasoning rather than for compilation. In fact the issues raised in the debate about "structured programming" are much wider and have transformed thinking about the programming task.[38-40]

Post-Hoare

Hoare's 1969 paper linked the ideas of van Wijngaarden and Floyd as well as boosting program verification from a theoretically possible idea into a tractable technique.[41]

Program verification ideas—at least in the narrow sense of this section—have made the transition from research papers, via postgraduate textbooks,[42] for instance, to undergraduate texts.[43,44] Leaving aside the plethora of notational details, there are at least two substantive issues that distinguish approaches even within this simplest form of program verification.

In most publications, assertions—be they preconditions, invariants, or postconditions—are predicates of one state. This seemingly innocent decision causes problems when writing meaningful specifications. For example, the specification of the division task discussed previously only requires that the final state is such that $r < y \wedge x = r + y * q$, so an implementation could arbitrarily change x to 7 and y to 3 to ensure that $q = 2 \wedge r = 1$ is always the correct answer. In this simple case, one might specify in some other way that changes to the values of x and y are not allowed; in the case of a program that is specifically intended to change a large data structure (for example, in-place sorting or matrix inversion) it is not possible to get by with postconditions of the final state alone. Techniques, such as using auxiliary variables to record initial values, should be compared to the effect of making the step to postconditions of two states. Done by Zohar Manna as early as 1968,[45] this is something that has distinguished the Vienna Development Method (VDM) from most other approaches since 1973.[46] Seeing this as the natural way to present specifications, I[47] tolerated a set of proof rules that were clumsy compared to earlier work.[26] Fortunately, Peter Henry George Aczel wrote an extensive, but unpublished, note in January 1982 that shows how an elegant set of rules can be given for proofs using predicates of two states.[48] In fact, most of the rules are extensions of those for the single-state case. (The resulting rules are used in more recent VDM publications.[49] Others, including Hehner,[50] use similar specifications; J. Coenen et al.[51] clearly and concisely compare the one- and two-state approach; Tarlecki[52] goes so far as to use preconditions that are predicates of two states.) Among other changes, Aczel showed that it is more convenient in a postcondition to use a hooked version of variable names to mark values in the prior state (and undecorated names for the post state).[53]

The other significant issue to which approaches diverge is the way partial functions are handled in proofs about programs. Mathematicians might find division by zero a nuisance but partial functions are common enough in computing that they require systematic handling. Nondenoting terms can arise from recursive definitions and from the basic operators of data types like sequences. The need to prove results about logical formulas that include terms using partial functions casts doubt on the applicability of classical logic. (McCarthy[15] was one of the first to recognize this problem.) Approaches to handling partial functions that attempt to salvage classical logic include avoiding function application and the use of variant-equality notions. An alternative is to accept the need for a nonclassical logic. The most frequently used nonclassical logic adopts nonstrict (but also noncommutative) conditional versions of the propositional operators; alternatively one can use commutative, nonstrict, and/or operators as defined in Stephen C. Kleene's "strong truth-tables." A consistent and complete proof system for such a logic has been used in VDM. Howard Barringer et al. presented the "Logic of Partial Functions"[54] and also analyzed approaches to reasoning about partial functions.

Another area of research that should be mentioned here is various notions of induction. Key results by David Michael Ritchie Park,[55] Hans Bekič, and Dana Scott are to be more fully explored in a paper on language semantics. The concept of structural induction is so central to many program proofs that it deserves some air-

ing here. Rodney M. Burstall showed the usefulness of this rule in his talk at the 1967 Yorktown Heights conference. It was published[56] with connections to "recursion induction" that Burstall discussed with McCarthy at the conference. Essentially, structural induction shows that inductive proofs over the constructors of objects such as trees can be conducted directly without the need to recast them as numerical induction on nesting depth.

The Floyd–Hoare contribution shows the main line of research. Not surprisingly, there is research from other authors that has also had bearing on tractable ways of reasoning. It is not necessarily a value judgment on that work to omit it here: It has just had less influence sometimes because of an accident of timing.

Formal development methods

A crucial test for program development ideas is whether they scale—that is, whether they can be used to develop large programs without an exponential growth in effort. This section plots the course from proofs about tiny programs to development methods that are now applied in industry.

Stepwise design

Given methods—like those Hoare wrote about[26]—for reasoning about programs, it was natural to try to apply them to larger programs than in the original publications. Unfortunately, it quickly became clear that the use of a set of axioms to build a proof of a complete program was an inadequate response to the challenge of creating nontrivial programs. In fact, one of the first people to whom this became evident was Hoare himself. One of his earliest publications had concerned efficient programs related to sorting: He christened his famous sorting program Quicksort.[57] After publishing in 1969[26] he attempted to use the same method of presenting a program, then reasoning about it to justify the Find program that had been published in 1961, together with Quicksort. In fact, he submitted a first draft of a proof in this form to the *Communications of the ACM* (*CACM*) but at least one of the referees balked at the length and opacity of the proofs, arguing that it was difficult to gain confidence from such a presentation. Hoare came independently to the same realization and saw that it was necessary to move to a presentation in which design decisions could be taken and justified one at a time. A revised paper[58] relies on the same rules as his 1969 work but uses them to justify one stage of development at a time.[59]

Although the specifications of suboperations were not completely formalized, a crucial step toward development methods had been made. A *post facto* proof that a program satisfies its specification is only possible for a correct program; moreover, it's unlikely that attempts to construct such proofs for nontrivial programs is a cost-effective way to detect errors. The alternative—making one step of development and justifying it before further work—has huge advantages: The approach mirrors how a developer normally proceeds; it also has the potential to detect errors as soon as they are made. (In addition to the fact that testing cannot be relied on to uncover all errors, testing can only be applied once code has been developed.) Once developers have adopted a formal stepwise process, the potential exists for the use of formalism to increase the productivity of program development by reducing the "scrap and rework" inherent in basing design on erroneous decisions that are only detected in testing.

To make these ideas precise, it is worth saying more about the notion of "satisfaction." As described by Floyd, a specification can be given as pre- and postconditions that are assertions over states. These assertions can be written in first-order predicate calculus, which provides a precise notion of the set of states for which the assertion is true. An implementation is said to satisfy a specification if, for all states for which the precondition is true, the implementation both terminates and yields a state for which the postcondition is true.

The same notion of satisfaction can be used in discussing specifications of any subcomponents. The need to support stepwise development gives rise to a requirement of compositionality. This notion is easy to illustrate in the context of developing sequential programs. Suppose a program S, whose specification is given, is to be implemented; a designer might decompose the task into two subtasks $S1$ and $S2$ for which specifications are also recorded; the design might show that execution of $S1$ followed by $S2$ will satisfy the specification S; a *compositional* method is one in which any implementations that satisfy specifications $S1$ and $S2$ can be combined to yield an implementation that satisfies S. In other words, the specifications of the subcomponents tell their developers all that they need to know about their task—the researchers working on Gypsy used the suggestive term *independence principle*. This is easy to illustrate for sequential programs and not difficult to achieve—the "Concurrent programs" subsection explains why compositionality is a significant complication in developing concurrent programs.

Being able to justify stages of design was a significant step forward for developers from the situation where one could only prove that complete (correct) programs satisfied their specifications. The task of finding both code and assertions still remains. In general, it requires human intuition to design programs but it is possible to provide heuristics that make it easier to decide how to tackle particular forms of specification. Such heuristics are studied extensively for predicate transformers. The step from the forwards to the backwards assignment axiom is described previously; Dijkstra proposed regarding all program constructs as ways of transforming a (required) postcondition into a "weakest precondition" that would guarantee successful termination.[60] Thus, rather than viewing

$$\{x = r + y * q \wedge y \leq r\}\ r := r - y\ ;\ q := 1 + q\ \{x = r + y * q\}$$

as an inference from two other triples, one can ask what is the weakest precondition under which

$$r := r - y\ ;\ q := 1 + q$$

will terminate and deliver a state for which

$$x = r + y * q$$

is true. The weakest precondition is just

$x = r + y * q \wedge y \leq r$ but the change of viewpoint has some interesting consequences. First, there is a nice calculus of such predicate transformers. Writing $wp(S,R)$ as the weakest precondition for statement S to achieve a state satisfying R

$$wp((S1;S2),\mathrm{R}) = wp(S1,\ wp(\mathrm{s2},R))$$

Dijkstra's claim[60] to have "tightened things up a bit" (over Hoare's work[26]) is more than justified: He handles total correctness (including termination) where Hoare's axioms covered only partial correctness. Dijkstra's guarded commands embody and prompt a clear handling of nondeterminacy; there is a strong anti-operational flavor to Dijkstra's explanations.[61]

Besides providing predicate transformer semantics for a specific programming language that contains the main structured programming constructs, Dijkstra[60] also presents healthiness conditions that all predicate transformers should satisfy.[62,63]

Abstract data structures

For a specification to be a useful starting point for development, it should be shorter and clearer than a program that satisfies it. Postconditions are often easier to formulate—and are more compact—than programs written in a standard algorithmic language. In particular, the use of conjunction and negation are powerful ways of expressing a desired result without needing to describe how to compute it. The examples described thus far have used variables whose values are numbers; but many computing problems can only be discussed in terms of values that have more elaborate structure. Finding ways to represent such values in linearly addressable memory is one of the challenges of program design.

Unfortunately, the outcome of such a design is rarely short and almost never tractable in the sense that its properties are easily determined. It became clear in the 1960s that specification and design methods would have to cope with richer data structures than numbers. In particular, developers found it necessary to write specifications in terms of some abstract class of states and then find ways to relate this to the representation chosen in the design process. For example, a relational database system might be specified succinctly in terms of mappings (finite functions) from relation keys to sets of tuples, whereas the final representation might involve complex chains of pointers to achieve reasonable performance of update and query operations.

Following McCarthy's call in 1961, the largest formal descriptions in the 1960s were of programming languages. The IBM laboratory in Vienna was at the forefront of this work. In its cooperation with IBM's UK laboratory, two different ways of modeling access to local variables had arisen. Researchers were concerned as to the equivalency of the two definitions. Peter Lucas undertook a proof[64] that the two models defined the same language using a "twin machine" that combined two different representations of the underlying state. A predicate that defined valid elements of this cross product essentially fixed a relation between the two representations. This prompted me to write another Vienna report[65] that highlighted the advantage of a functional relationship between representation and abstraction (that is, there can be many possible representations for the same abstract value). Robin Milner had worked on this problem in Swansea from 1969,[66] and he presented a careful algebraic view of simulation in a Stanford technical report.[67] Hoare's paper on the topic[68] is one of the most com-

mon citations in this area. He also uses a function from representation to abstraction as the basis of his set of proof obligations.

Hoare acknowledges Milner's influence, but the value of historiography in this area is diminished when one knows that Christopher Strachey described the idea to Burstall in the mid-1960s, yet considered it so obvious as to be not worth writing down.[69] Some indication of how Strachey used the idea of developing programs via abstract data objects can be seen where he outlines the design of a program to play checkers.[70] In view of the date of this contribution, it's impossible to resist giving at least some quotes:

> In the early days of programming—say 15 years ago—mathematicians used to think that by taking sufficient care they would be able to write programs that were correct. Greatly to their surprise and chagrin, they found that this was not the case and that with rare exceptions the programs as written contained numerous errors …
>
> Although programming techniques have improved immensely since the early days, the process of finding and correcting errors in programs—known, graphically if inelegantly, as "debugging"—still remains a most difficult, confused and unsatisfactory operation …
>
> There is no doubt that with the current techniques we have nearly reached our limit in programming. Could we not, however, improve the techniques? …
>
> I have left to the end what seems to me the most difficult, but also the most interesting and potentially rewarding, problem concerning programming languages. This is to lay a firm mathematical foundation for the construction of hierarchical systems of programs and to develop a calculus for manipulating them.[70]

The simplest way to record a one-to-many relationship between a space of abstract states and their representations is by a function from the representations to the abstract states. These functions are called abstraction (or retrieve) functions, and they can be used to formulate correctness conditions for program development steps that are concerned with the design of data as opposed to the development of algorithms. Data refinement (or reification) often occurs early in the design of a program and is thus of importance in obtaining productivity from the use of formal methods. It is therefore surprising that relatively few books give prominence to this topic.[71]

A number of technicalities have become clearer over time. The approach described previously is often called model-oriented (in contrast to the property-oriented approach described next. There is a risk that a model can be chosen that is overly concrete in the sense that it conserves nonessential information. One can make this notion of "bias" quite precise.[72] The purpose of a specification is to fix a behavior that is made visible via operations and their arguments/results—the underlying state model is a hidden sort. Bias is almost inevitable in an implementation but can both obfuscate a specification and make its use in subsequent design justifications more burdensome. Terms such as (freedom from) implementation bias and full abstraction are used to characterize (the equivalence class of) specifications that retain only enough information to support the intended function of a system. For such specifications, abstraction functions can be found from any implementation. There are, however, systems for which it is not possible to achieve exactly this sort of freedom from bias. In particular, there are systems whose specifications allow nondeterminacy (which need not be present in an implementation) and whose states contain information that can also be dropped when designing implementations. Others have developed proof rules that resurrect the idea of relations between abstraction and representation.[73,74] In a sense made precise in the papers, these rules are complete.

Here, I describe model-oriented specifications first because they fit the flow of ideas but, if one were to count the early publications, one would observe that there was initially more research on what might be called property-oriented specifications of abstract data types. The observation earlier that a model-oriented specification can contain unnecessary detail led researchers to seek specifications that appear to contain no model at all; one only has to look at Peano's axioms for the natural numbers to see where the stimulus of this idea came from. The key is to define a data type with (the signatures of its operators, and) axioms giving relationships between its operators.

For example, one can fix part of the behavior of a stack by saying that pushing one element onto a stack, then popping the stack, returns it to its original value. To some extent, such specifications are like the Peano axioms for the natural numbers and exponents of such specifications often call them "algebraic specifications." (Claiming a monopoly on this term could be said to be a little unfair: Textbooks happily follow Peano-like characterizations of the natural numbers with a model for the rational numbers.) The most commonly cited

early references to this work are by John Guttag,[75] Steve Zilles,[76] and Joseph Goguen et al.[77] (See Lockwood Morris, also.[78])[79] In fact, Peter Lucas[80] employs the same idea of presenting operations by their relationships and this even uses stacks as an example. Lucas's papers were less developed than the citations here, but it is noteworthy that what might be considered to be a huge model—the VDL definition of the PL/I programming language—characterized its storage component by axioms (for the most accessible source for this material, with references to the 1969 version of the PL/I definition, see Hans Bekič and Kurt Walk[81]); Clifford Jones[82] also uses a correctness notion that rests on preserving relevant properties.

Once again, there are a number of technicalities with property-oriented specifications. There are, for example, different (initial, final, or loose) interpretations of the axioms and this choice influences how implementations are proved to satisfy specifications. Acute minds have succeeded in proving that there are classes of data types that cannot be specified by a finite set of axioms unless so-called "hidden functions" are employed. This casts doubt on the claim that such techniques avoid all danger of implementation bias. How to handle—in property-oriented specifications—partial operators or ones that are nondeterministic are issues on which full agreement is absent even now.[83]

It is not useful to view the model/property-oriented split as a schism—each concept has its applicability; there are data types naturally viewed as collections of (immutable) values and others whose description is best given in terms of a state. There are, however, some interesting technical questions about the two approaches' interrelationship. Coming from the model side, one can view a property-oriented specification as providing a model in terms of the word algebra of the operators; an unbiased model comes from the word algebra of the type generators. From the property-oriented viewpoint, one could say that model-oriented specifications use a particular set of hidden functions (the operators of the underlying types such as sets and mappings).

Development methods

The ideas of operation decomposition and data refinement can be combined in the sense that systems can be developed using both. What is required is a coherent notation and set of proof obligations. VDM was one of the earliest attempts to create such coherence. Space constraints prevent a detailed discussion here; suffice it to say here that VDM grew out of earlier work on the formal description of programming languages. The earliest full-length expositions of VDM are by Dines Bjørner and Jones[84] and Jones[47] Bjørner and Jones are concerned with denotational semantics and its use in the development of compilers; the later Jones work relates to general program development).[85] VDM specifications are model-oriented; a module defines a class of states and a series of operations that change and depend on those states; precise notions of operation decomposition and data reification are part of the development method. The VDM has been used in organizations far beyond that which originated it.

Another specification language closely related to VDM is known as Z. Jean-Raymond Abrial was the originator of Z but little was published[86] other than in early form. Researchers who continued to develop and apply Z at Oxford University have published a number of books.[87] Until recently, there has not been a development method associated with Z.

Both Z and VDM are now undergoing British and international standardization. Other development methods include Gypsy, which has been applied to a stack of implementations from a high-level programming language down to chip design.[88]

One key technical problem is the way in which large specifications can be built up from components. Clear[89] provides a valuable starting point but the problem is still an area of active research (and disagreement).

Controversies

This history has not been without its controversies and to some extent these have often confused the issue of what is really done in a development method: Richard DeMillo et al.,[90] for example, asserts that the concept of proof in mathematics has been misunderstood in program verification work. Acceptance of proofs in mathematics—it is argued—is a social process with putative proofs subjected to public scrutiny; the lengthy and detailed logical derivations used to justify some (steps of) program development are quite unlike mathematical proofs. This argument has some truth, although the debate sparked by DeMillo's article mainly concerned the extent to which the position attacked was a straw man rather than the way most active scientists were thinking. In fact, the overall structuring of a program design into justifiable steps can usefully be compared to the level of a normal mathematical proof. The incentive—in critical applications—to check the often tedious detail of subsidiary lemmas is to cross-check for resid-

ual errors. There is also a message hidden here in the need to structure knowledge about computing science. Ole-Johan Dahl was one of the first people to appreciate that formal development of programs could only make progress if knowledge were accumulated from one application to another in the form of theories. Several papers[91] hold out the hope of a real structuring of knowledge about computing ideas—this knowledge should evolve in the way that mathematical theories are built up over time.

In the late 1980s, a violent debate in the correspondence pages of *CACM* was caused by Arthur J. Fetzer[92] which—as its title "Program Verification: The Very Idea" suggests—casts doubt on the whole enterprise of verifying programs. The most interesting point raised in this debate was whether reasoning about an abstract algorithm actually tells us anything about the outcome of a program when run on a machine. Here again the attack pinpoints potential weaknesses, but recall that the earliest efforts to apply formalism were often aimed at languages and the development of compilers precisely because it was clear to researchers that the whole task of verifying programs in some language could be undermined if the compilers for that language contained errors. Furthermore, the realization that a compiler is developed with assumptions as to the meaning of the instructions for the object machine has led scientists to apply verification ideas also to the design of hardware. The danger of physical hardware failure remains, but—at least as far as undetected errors are concerned—microelectronic devices are now far more reliable than mechanical systems. If the point is simply to make everyone aware that absolute reliability is unattainable, then the author made a valid point.

In fact, there is a much more serious issue that has probably received too little attention in spite of its having been raised by those within the verification community. Probably the biggest danger for claims about verification is that they become seen as some guarantee of fitness of purpose; it must be seen that all that is even theoretically possible is to prove (under listed assumptions) that a program satisfies a formal specification. Whether the formal text of the specification actually does something useful is an issue that is not susceptible to mathematical argument.

Other issues

Many issues relate to program verification but have not been explored in as much detail as the basic theory of sequential programs. This section touches on four such issues; secondary literature is cited to provide pointers to the fields.

Concurrent programs

Sequential programs can be considered to run in isolation. In reality, this is rarely the case on modern computers. Machine resources are shared between many programs by an operating system, part of whose task is to provide the illusion of isolation. But the operating system itself is a member of an interesting class of programs in which concurrency is an issue. This class of programs contains many that are important to society. For example, programs that handle airline seat reservations might have to handle messages from many agents who are negotiating for the same resource.

Another area of difficulty comes from concurrent programs that must respond at particular times to stimuli that come from physical sensors. To stay in the same field, programs that control aircraft landing are in this class. Concurrent and real-time programs are extremely difficult to design, and the public has been made aware of their flaws by such news items as delayed launches of the US space shuttle. Although this article is confined to programming, there is a clear incentive to apply formality to the process of designing complex hardware: This is another area where the ability to handle concurrency is essential.

As the example of an operating system suggests, concurrency is best confined into kernel programs so that systems can be built with large parts being designed under the assumption of isolation. But this leaves a core of concurrent programs to design. The efficacy of testing—already inadequate as a way of ensuring correctness for sequential programs—is further reduced by the fact that running the same program twice can give different results depending on what happens in the environment. Clearly, formal methods for verifying concurrent programs are vitally important. Unfortunately, no consensus has evolved on how best to handle concurrency. In part, lack of consensus is almost certainly due to the diverse nature of program development tasks but it must also be seen as symptomatic of a field still trying to find tractable methods. Here, I can do no more than identify key references.

Dijkstra has had a major impact on research into concurrent systems. His own early work on the design of the THE operating system led him to understand the necessity of careful arguments about correctness; it also made him one

of the most prolific sources of challenge problems. A delightful example of his style[93] uses semaphores to control concurrent processes. Dijkstra's privately circulated series of "EWD" notes have—by posing challenges—provided a crucial stimulus to concurrency research. As so often in science, asking the right questions can goad people into solving problems.

One of the clearest paths of development toward making concurrent programs more tractable can be seen in Hoare's work. Semaphores are a low-level device for controlling concurrency. They are adequate in that they can be used to achieve any particular effect; they lack structure in that there is no way of checking that they are properly used. Hoare's conditional critical sections[94] and monitors[95] are progressively richer constructs whose structure rules out certain sorts of potential error. (Hoare's monitor concept was implemented in the programming language Pascal Plus.) Both of these language constructs are intended to harness shared-variable concurrency where programs interfere with each other by changing common variables. Hoare's boldest step was to a language called Communicating Sequential Programs (CSP)[96] in which no variables are shared between processes. Although it represents a larger discontinuity, CSP can be seen as another step along the path of increasing structure to make concurrency more tractable. Communication between processes is handled by (synchronous) message handling; interference between processes has not disappeared but now manifests itself in terms of the differing results of interactions.

Hoare's contribution above is illustrative—he would not pretend that this work was done in a vacuum. In fact, the development of the monitor idea is closely paralleled in the work of Per Brinch Hansen.[97] Robin Milner's CCS (Calculus of Communicating Systems)[98] is in key respects similar to CSP, but CCS was not designed as a programming language. Milner writes (private communication, December 1991):

> In designing CCS as a calculus, with an eye on algebraic and other theory and aiming for a small repertoire of *theoretical* constructions, I hit upon the same notion of synchronized communication which Hoare proposed in CSP as a means to control the proliferation of *programming* constructions. This agreement from two different viewpoints was encouraging.

The extensive research on CCS has focused on notions like observational equivalence of agents (processes). Park proposed the important concept of "bisimulation."[99]

The development of methods for designing concurrent programs bears an interesting resemblance to that for their sequential cousins. Most methods are initially explained with *post facto* proofs in mind and only later are the ideas evaluated for applicability in forming the basis of development methods. But the sobering observation is that most fail to yield usable development methods for concurrent systems. The problem is compositionality. Whereas it is relatively simple to completely characterize subcomponents that are to be executed in isolation, interfering programs are difficult to specify in a way that facilitates separate development.

In an article,[100] Ed Ashcroft and Zohar Manna wrote that the fact that two processes can interfere with each other's states is reflected by constructing an equivalent nondeterministic (sequential) program that needs a number of assertions related exponentially to the size of the programs. The approach is in no way compositional because the correctness of the two processes cannot even be considered until their final code is available. Susan Speer Owicki wrote her thesis under the supervision of David Gries. Owicki's thesis[101] broke the task of verifying concurrent programs into two stages: The first treats them as sequential programs; the second checks whether the proof of process A can be interfered with by statements in process B and vice versa.[102] Although this meant that some meaningful verification could be conducted separately, it is clear that the so-called Owicki–Gries method leaves open the risk that development work that meets all of its requirements might have to be discarded after detailed code has been designed (in this case, the interference freedom test).

Several researchers realized that some way of controlling the interference would have to be built into specifications: Nissim Francez and Amir Pnueli[103] specify interference but do not present a development method in the sense suggested here; Jones[104] uses rely and guarantee conditions to provide a partial specification of interference; a rely-guarantee approach was published independently.[105] A significant literature has developed on compositional methods; see Willem-Paul de Roever[106] for a good overview.

The proof methods discussed so far have nearly all been based on classical logic (first-order predicate calculus). As mentioned, Burstall suggested that some form of temporal logic could be used in program development. It would appear that Pnueli was the first person to suggest that temporal logic be used in specify-

ing and developing concurrent programs.[107] Within the temporal logic framework, Barringer et al. tackle the issue of compositionality.[108]

A major contributor to formalisms for concurrency has been Leslie Lamport. Among other contributions he outlined the issues of safety and liveness[102] and proposed a novel *Temporal Logic of Actions*.[109] An early attempt to use formalism on operating systems is by Hugh Lauer.[110]

A number of putative development methods have evolved for concurrent programs. The Unity method is perhaps the most widely discussed.[111]

As mentioned at the beginning of this section, the work on parallel systems is diverse. One contentious issue that has emerged is the contrast between interleaving and true concurrency. The latter position is taken by Carl Adam Petri and others working on Petri nets.[112]

Language semantics

The study of languages can be partitioned into syntax, semantics, and pragmatics; Heinz Zemanek applied this terminology to programming languages.[113] The valid strings of a language and some suggestion of their structure are defined by its syntax. Although the history of this area has little to do with my main topic in this article, an adequate method for defining the syntax of programming languages (Backus–Naur Form) was employed in the description of Algol 60.[114] Finding ways of defining the semantics, or meaning, of strings is much more challenging.[115]

Given a program in some programming language, its user might be interested in running it together with input data to determine the output or result. An *interpreter* for a programming language is a program that runs on a computer and interprets statements to transform input values into output; most implementations of Basic are interpreters. A *translator* (or *compiler*) translates programs in the given language into *object programs* in the machine language of some computer—the object program is then run to translate input to output. Two ways of recording the semantics of programming languages are abstractions of interpreters and translators. The first attempts to write formal semantics of programming languages were *abstract interpreters*. They performed exactly the same function as interpreters: Given a program and some input values, they computed the output. But, rather than being written in machine language, they were written as mathematical functions about which it is possible to reason. One of the earliest papers was by McCarthy,[116] describing only a tiny programming language but, together with ideas from Calvin C. Elgot and Peter John Landin, these provided the basis of definitions of major programming languages. A major milestone was the 1964 IFIP Working Conference on Formal Language Description Languages at Baden bei Wien whose proceedings[27] capture the key discussions. Many of Zemanek's Vienna group[117] were involved, and they went on to tackle the semantics of a large and complex language: PL/I.[118] The Vienna group called their descriptive style Universal Language Definition, but their notation became known as Vienna Definition Language (VDL).[119] Such abstract interpreters are said to give an *operational semantics* to a language: A meaning is given to a program plus input data but not to a program per se. This can present some problems when such a description is to be used to reason about implementations of the language. However, such definitions have been used in this way[120] and for larger languages in papers from the Vienna group, such as by Lucas.[64]

A compiler maps programs in a programming language into another language; a *denotational semantics* maps programs into some understood set of denotations. If these denotations are mathematical functions from input to output, **a** program's meaning can be discussed independently of particular input values. This can simplify the task of proving implementations or other general results. Strachey and Landin[121] made major early steps toward denotational semantics; problems of program components whose denotations are functions of a type whose domain includes its own type posed foundational problems that were solved by Dana Scott. Scott actually discovered his models of reflexive domains while in Oxford and the Programming Research Group—led by Strachey—made denotational semantics into a major research topic,[122,123] with Joe Stoy's work[124] being a classic of exposition. Interestingly, the Vienna group went on to adopt the denotational approach and develop VDM.[84,125,126]

Here again there has been controversy. Peter Naur, for instance, made an attack on formalization.[127] Because I cite my own work in this article, it is not fair to enter into further discussion of this controversy. Naur's position ever since his own early contributions appears to have been that formalism should be one tool and not be pursued in isolation from applications.

Although there are advantages in denotational definitions, it would be a mistake to assume that they have supplanted operational definitions. There are considerable technical difficulties especially with concurrency in find-

ing appropriate denotations. Furthermore, the difficulties in reasoning about operational definitions can be greatly reduced if care is exercised to avoid unnecessary mechanistic features. Gordon D. Plotkin made proposals for structured operational semantics[128] that provide a widely used notation for describing programming languages.

Both operational and denotational descriptions can be thought of as fixing semantics in terms of some sort of model. There are also proposals for property-oriented descriptions of programming languages. In fact, the "Proofs about sequential algorithms" section covers one such approach: Floyd and Hoare both claimed that inference rules could provide a semantics for a language, and Hoare's work is often called axiomatic semantics. Another approach that avoids the need for models is to describe semantics by giving sets of equivalences over texts of the language.[129,130] The method is often known as algebraic semantics: It works best if the language constructs have pleasing algebraic properties like associativity and distributivity. This is yet another echo of the quest for tractability in the design of languages.

The idea of providing computer support for program verification goes beyond the scope of this article.

Novel languages

The memory of the so-called von Neumann computer is a linear sequence of cells (bits, bytes, or words), and the first programmers had to map their programs and data onto this by hand. Since that time, considerable progress has been made toward finding more tractable languages in which to construct programs. Languages like Fortran made the programmer's task easier by making it possible to write a program in terms of named variables whose addresses in the linear memory were computed by a compiler. This is just one example of many where the symbol manipulation powers of computers have aided the task of using those same machines. After the first small steps were made away from the raw interface of the machine, researchers began to investigate the question of whether radically different languages could overcome the problem of designing programs that solve problems.

Languages that reflect the idea that computer memory is changed by some updating operation are known as imperative, and in most such languages, the updating is performed by assignment statements. Assignment statements have the undesirable property that so-called program variables contain different values at different points in program execution. (This problem is compounded by languages like Fortran where the updating operation was written as = resulting in statements like $X = X + 1$ that are nonsense if read as equations. At one time, the fact that program variables do not behave like mathematical variables led to a series of suggestions for alternative names.) This is one of the problems of reasoning about imperative programs.

As has been explained, one way around this problem is to reason about programs via assertions. A more radical approach is to ban assignment statements and work with languages that are purely functional. The hope is that programs in such languages would be much easier to reason about—a point of view strongly presented by John Backus.[131] This hope is perhaps easier to realize than the ensuing challenge of finding purely functional languages that can be efficiently translated to machine code. Efficient programs are mostly written in imperative or impure functional languages, and even if research on machine architectures makes pure functional languages more efficient, there is the question of expressing inherently state-based tasks.

A step beyond functional languages is to ask to what extent a translator for logic can be constructed. The dream is to see a computer execute a specification, thus obviating the need to design a program. The undecidability of any reasonably powerful logic shows that there will be limitations on the expressiveness of such logic languages but even decidable logics like propositional calculus have unacceptably slow decision procedures. The approach most clearly espoused by Robert Kowalski[132] is to use a defined subset of general logic (Horn clauses) and to help the translator find efficient search strategies by adding control information. One of the cornerstones of the Japanese fifth-generation program was to make machines more usable by exploiting languages like Prolog.[133] There would appear to be less conviction today that this approach is a general solution to programming problems but there is clearly a role for logic languages in some computing tasks.

The development of such novel (nonprocedural) languages gives an insight into an important aspect of the science of computing. In a physical science like astronomy, scientists study the reality they face in nature—any models have to match that reality. In pure mathematics, one can generate a system by choosing axioms and studying their consequences—beauty is the main criterion. What of computing? Clearly, one reaction to finding a class of languages intractable is to change the lan-

guage—this sounds like mathematics. But there still has to be a way to execute programs on physical devices, and the efficient design of such implementations can be a huge engineering task. It is true that the prodigious growth in the capacity and performance of electronic devices has made it possible to trade away machine efficiency if it results in better use of human effort but this works over linear—not exponential—factors. Computer science has progressed by developers' choosing new abstractions that are more tractable than their predecessors and then developing theories that match the new reality that has been created; computer engineering attempts to provide efficient implementations of the abstractions.

No better example of this can be given than the developments relating to communication-based concurrency. One of the first steps away from shared variable concurrency was in the languages that provided queued message passing;[134,135] some hint of the history of process algebras was discussed in the "Concurrent programs" subsection. "occam"[136] is a development of CSP; transputers provide an architecture for that language. One can see a growth of a theoretical concept to its realization in silicon, which took less than a decade.

A more recent trend in the development of usable languages is object-oriented languages. Their genesis is Ole-Johan Dahl and Kristen Nygaard's Simula 67 language.[137] Languages like Smalltalk-80[138] and Eiffel[139] are being recognized as having potential to increase the productivity of programmers; there is now progress in studying concurrency in this framework.[140]

Postscript

Alan Turing's 1936 paper on the *Entscheidungsproblem* had introduced the idea of what became known as the Turing machine. The profound influence of this abstract concept and his widely known work on computers led the Association for Computing Machinery to name its most prestigious award after Alan Turing: Many of the people mentioned in this article have been recipients of the Turing award.

The geographical distribution of research work on program verification is interesting. Although a fair proportion of the early contributions to formalism came from the US, the sort of formal methods discussed in this article have been more actively researched in Europe than in the US, where the contributions have come from a small number of institutions. Apparently, in the US, algorithm design and complexity theory are more popular research topics for formally minded computer scientists. Notable contributions have come from Japan and the (former) USSR, particularly on partial evaluation and mixed computation. (These are among a number of areas that this article fails to discuss. I should be grateful for any input because I do not rule out the creation of further editions of this history.)

Computing science is an exciting topic whose structure is evolving. As it does, more and more branches of mathematics are being shown to be relevant and are changing as a result of their contact with computing.

Acknowledgments

This article is an extended version of an unpublished paper. My research has been partly funded by the (UK) EPSRC grant for the Dependability IRC. I am grateful to Richard Giordano, Don Good, Ian Hayes, J.A.N. Lee, Peter Lindsay, Robin Milner, José Nuno Olivera, Brian Randell, and Mike Woodger for comments on drafts of this article; and to Fritz Bauer, Martin Campbell-Kelly, David Cooper, Bob Floyd, Dick Hamlet, Tony Hoare, Ralph London, Fred Schneider, John Tucker, Maurice Wilkes, and Heinz Zemanek for other input. This project was discussed at two meetings of IFIP's W.G. 2.3. Brian Randell and J.A.N. Lee drew my attention to Michael Mahoney's paper "What Makes History," which helped me avoid a number of trivial amateur's mistakes. I am extremely grateful to Alison McCauley without whose painstaking checking the bibliographic information would never have achieved its current level.

References and notes

1. This article is not a true history; for one thing, I am not a historian. My biases and a selective knowledge make the current text open to criticism. At best, this article will provide a source for subsequent historical research. One topic that is largely ignored here is that of numerical analysis.
2. A.M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proc. London Mathematical Soc., Series 2*, vol. 42, pp. 230-265, 1936. Correction published ibid., vol. 43, pp. 544-546, 1937. Reprinted in M. Davis, *The Undecidable*, Raven Press, 1965, pp. 115-154.
3. Normally, just Tony Hoare, but references show all initials.
4. Brian Randell has pointed out that a concern with correctness was already present in the pre-electronic phase: Charles Babbage (1791–1871) wrote about the "Verification of the Formulae Placed on the [Operation] Cards." See B. Randell, *The Origins of Digital Computers: Selected Papers*, Springer-Verlag, 2nd ed., 1975, pp. 45-47. Konrad Zuse's *Plankalkul* is one of the earliest

programming languages (see K. Zuse, *Der Computer: Mein Lebenswerk,* Springer-Verlag, 1984); Heinz Zemanek has kindly checked with Zuse and confirms that the concern was shared but that there were no specific provisions for correctness arguments.

5. H.H. Goldstine and J. von Neumann, "Planning and Coding of Problems for an Electronic Computing Instrument," 1947, part II, vol. 1 of a report prepared for US Army Ordnance Dept.; republished as pp. 80-151 of *John von Neumann: Collected Works*, vol. V, *Design of Computers, Theory of Automata and Numerical Analysis,* A.H. Taub, ed., Pergamon Press, 1963.
6. A.H. Taub, ed., *John von Neumann: Collected Works*, vol. V, *Design of Computers, Theory of Automata and Numerical Analysis,* Pergamon Press, 1963, pp. 81-82.
7. Ibid., p. 83.
8. Ibid., p. 92.
9. Ibid., p. 98.
10. Ibid., p. 100.
11. Ibid., pp. 84 and 92.
12. A.M. Turing, "Checking a Large Routine," *Report of a Conference on High Speed Automatic Calculating Machines*, Univ. Mathematical Laboratory, Cambridge, UK, June 1949, pp. 67-69. The printed text of Turing's paper contains so many transcription errors that it took considerable effort to decipher. For a corrected version that is also related to later work, see F.L. Morris and C.B. Jones, "An Early Program Proof by Alan Turing," *Annals of the History of Computing*, vol. 6, no. 2, Apr. 1984, pp. 139-143.
13. A. Hodges, *Alan Turing: The Enigma,* Burnett Books, 1983, p. 355.
14. J.A.N. Lee pointed out that John McCarthy said of the design of inter alia Algol 60: "It was stated that everyone was a gentleman and no one would propose something that he didn't know how to implement."See R.L. Wexelblat, ed., *History of Programming Languages,* Academic Press, 1981.
15. J. McCarthy, "A Basis for a Mathematical Theory for Computation," *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, eds., North-Holland, 1963, pp. 33-70.
16. J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine," *Comm. ACM*, vol. 3, no. 4, Apr. 1960, pp. 184-195.
17. J. McCarthy, "Towards a Mathematical Science of Computation," *Proc. Information Processing '62,* C.M. Popplewell, ed., North-Holland, 1963, pp. 21-28.
18. Edsger Wybe Dijkstra described in his Turing Award lecture (E.W. Dijkstra, "The Humble Programmer," *Comm. ACM,* vol. 15, no. 10, 1972, pp. 453-457) how his decision to work in computing was influenced by van Wijngaarden.
19. A. van Wijngaarden, "Numerical Analysis as an Independent Science," *BIT*, vol. 6, no. 1, 1966, pp. 66-81. (Text of 1964 talk.)
20. As mentioned, little is said here about the problems on numerical analysis. In fact, most early research on reasoning about programs was confined to discrete data—a notable exception is found in T.E. Hull, W.H. Enright, and A.E. Sedgwick, "The Correctness of Numerical Algorithms," *ACM SIGPLAN Notices*, vol. 7, no. 1, Jan. 1972, pp. 66-73.
21. P. Naur, "Proof of Algorithms by General Snapshots," *BIT*, vol. 6, no. 4, 1966, pp. 310-316.
22. R.W. Floyd, "Assigning Meanings to Programs," *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science,* Am. Mathematical Soc., 1967, pp. 19-32.
23. Naur was to develop his ideas, together with others on action clusters (see P. Naur, "Programming by Action Clusters," *BIT*, vol. 9, no. 3, 1969, pp. 250-258) and to fit them into his thoughtful but too little known book P. Naur, *Concise Survey of Computer Methods,* Studentlitteratur, 1974.
24. This is presumably the reason that Maurice Wilkes argues (M.V. Wilkes, *Memoirs of a Computer Pioneer*, MIT Press, 1985) that Turing (see Ref. 12) did not anticipate Floyd"s contribution: "Turing did use the word 'assertion' and he did point out the separate need to show the execution of the program would terminate. What was missing was the concept of loop invariant."
25. S. Gorn, "Common Programming Language Task: Final Report," Contract No. DA-36-039-SC-75047, DA Proj. No. 3-28-01-201, PR and C No. 58-ELC/D-4457, Part I, Section 5: "On The Logical Design of Formal Mixed Languages," 1959.
26. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, vol. 12, Oct. 1969, pp. 576-580, 583. Along with other major papers by Hoare, this is reprinted in *Essays in Computing Science,* C.A.R. Hoare and C.B. Jones, eds., Prentice Hall, 1989. Much of the description from here on in the main text was generated for the "link material" of that book and consists of private communications between Hoare and myself.
27. T.B. Steel, *Formal Language Description Languages for Computer Programming,* North-Holland, 1966.
28. K.R. Apt, "Ten Years of Hoare's Logic: A Survey—Part I," *ACM Trans. Programming Languages and Systems*, vol. 3, no. 4, 1981, pp. 431-483.
29. C.A.R. Hoare, "Procedures and Parameters: An Axiomatic Approach," *Symposium on Semantics of Algorithmic Languages,* LNM 188, E. Engeler, ed., Springer-Verlag, 1971, pp. 102-116.
30. M. Clint and C.A.R. Hoare, "Program Proving: Jumps and Functions," *Acta Informatica*, vol. 1, 1972, pp. 214-224.

31. E.A. Ashcroft, M. Clint, and C.A.R. Hoare, remarks on "Program Proving: Jumps and Functions," *Acta Informatica*, vol. 6, no. 3, 1976, pp. 317-318.
32. W.-P. de Roever, "Recursion and Parameter Mechanisms: An Axiomatic Approach, *Automata Languages and Programming*, LNCS 14, J. Loeckx, ed., Springer-Verlag, 1974, and *Call-by-Value versus Call-by-Name: A Proof Theoretic Comparison*, tech. report IW 23/76, Mathematical Center, Amsterdam, Sept. 1976.
33. 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.
34. It has often been argued that this goal was overambitious. In Hoare's presentation at the April 1969 meeting of IFIP Working Group WG 2.2 in Vienna, he responded to the challenge—that it had taken millennia of arithmetic before Peano's axioms were formalized—that he had not been on that task!
35. R.L. London et al., "Proof Rules for the Programming Language Euclid," *Acta Informatica*, vol. 10, no. 1, 1978, pp. 1-26.
36. D.I. Good et al., *Report on the Language*, version 2.0, tech. report ICSCA-CMP-10, Univ. of Texas at Austin, Sept. 1978.
37. E.W. Dijkstra, "Goto Statement Considered Harmful," *Comm. ACM*, vol. 11, no. 3, Nov. 1968, pp. 147-148.
38. O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, eds., *Structured Programming*, Academic Press, 1972.
39. E.W. Dijkstra, "A Constructive Approach to the Problem of Program Correctness," *BIT*, vol. 8, no. 3, 1968, pp. 174-186. Also, E.W. Dijkstra, *A Short Introduction to the Art of Programming*, Technisch Hogeschool Eindhoven (Eindhoven University of Technology), EWD-316, 1971.
40. N. Wirth, *Systematic Programming: An Introduction,* Prentice Hall, 1973. Also, N. Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, 1976.
41. A detailed technical assessment that focuses on the questions of consistency and completeness is given in Ref. 28 and in K.R. Apt, "Ten Years of Hoare's Logic: A Survey—Part II: Nondeterminism," *Theoretical Computer Science*, vol. 28, 1984, pp. 83-109. Another important study is J. de Bakker, *Mathematical Theory of Program Correctness*, Prentice Hall, 1980.
42. E.W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.
43. D. Gries, *The Science of Programming,* Springer-Verlag, 1981.
44. R.C. Backhouse, *Program Construction and Verification,* Prentice Hall, 1986.
45. Z. Manna, *Termination of Algorithms,* doctoral dissertation, Dept. of Computer Science, Carnegie Mellon Univ., Apr. 1968.
46. C.B. Jones, *Formal Development of Programs*, tech. report 12.117, IBM Laboratory, Apr. 1973.
47. C.B. Jones, *Software Development: A Rigorous Approach,* Prentice Hall, 1980.
48. Aczel (private communication) writes of the specifications: "It is familiar that this specification does not explicitly express all that we have in mind … a more flexible and powerful approach has been advocated by Cliff Jones in his book *Software Development: A Rigorous Approach …* His rules appear elaborate and unmemorable compared with the original rules for partial correctness of Hoare."
49. C.B. Jones, *Systematic Software Development Using VDM,* Prentice Hall, 1986.
50. E.C.R. Hehner, *The Logic of Programming*, Prentice Hall, 1984.
51. J. Coenen, W.-P. de Roever, and J. Zwiers, "Assertional Data Reification Proofs: Survey and Perspective," *4th Refinement Workshop*, J.M. Morris and R. Shaw, eds., Springer-Verlag, 1991, pp. 97-114.
52. A. Tarlecki, "A Language of Specified Programs," *Science of Computer Programming*, vol. 5, no. 1, 1985, pp. 59-81.
53. This led to some embarrassment for me when, at a seminar in Newcastle-upon-Tyne in 1982, Hoare gave a talk that accepted postconditions of two states but used the *x*, *x'* convention; I might have been pleased had I not been waiting to give my own talk with the first set of slides using the hook notation.
54. H. Barringer, J.H. Cheng, and C.B. Jones, "A Logic Covering Undefinedness in Program Proofs," *Acta Informatica*, vol. 21, no. 3, 1984, pp. 251-269.
55. Normally known as David Park. To avoid confusion, all references show only one initial. D. Park, "Fixpoint Induction and Proofs of Program Properties," *Machine Intelligence 5*, B. Meltzer and D. Michie, eds., Edinburgh Univ. Press, 1969, pp. 59-78.
56. R.M. Burstall, "Proving Properties of Programs by Structural Induction," *Computer J.*, vol. 12, no. 1, 1969, pp. 41-48.
57. C.A.R. Hoare, *Comm. ACM*, vol. 4, no. 7, July 1961, pp. 321-322.
58. C.A.R. Hoare, "Proof of a Program," *Comm. ACM*, vol. 14, no. 1, Jan. 1971, pp. 39-45.
59. Further applications are in M. Foley and C.A.R. Hoare, "Proof of a Recursive Program: Quicksort," *Computer J.*, vol. 14, no. 4, Nov. 1971, pp. 391-395, and in C.A.R. Hoare, "Proof of a Structured Program: 'The Sieve of Eratosthenes'," *Computer J.*, vol. 15, no. 4, Nov. 1972, pp. 321-325.
60. E.W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Comm. ACM*, vol. 18, no. 8, 1975, pp. 453-457.

61. Not only did the Blanchland (Oct. 1973) meeting of IFIP W.G. 2.3 have a part to play in the development (see Mike Woodger"s nice history in *Programming Methodology: A Collection of Articles by Members of IFIP W.G. 2.3*, D. Gries, ed., Springer-Verlag, 1978, pp. 1-5), but the firm rejection of arguments presented in an unnecessarily operational way has remained a hallmark of these gatherings.
62. A full discussion of predicate transformers can be found in E.W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990; see also W.-P. de Roever, *Dijkstra's Predicate Transformer, Non-determinism, Recursion and Termination*, tech. report 37, IRISA, Univ. of Rennes, 1976.
63. The so-called "refinement calculus" is a development whose impact goes beyond the 1990 cutoff for this article. See R.J.R. Back, *Correctness Preserving Program Refinements: Proof Theory and Applications*, tech. report, Mathematisch Centrum Tract, 131, 1980, and C. Morgan, *Programming from Specifications*, Prentice Hall, 1990.
64. P. Lucas, *Two Constructive Realizations of the Block Concept and their Equivalence*, tech. report TR 25.085, IBM Laboratory, June 1968.
65. C.B. Jones, *A Technique for Showing that Two Functions Preserve a Relation between their Domains*, tech. report LR 25.3.067, IBM Laboratory, Apr. 1970.
66. For example, see R. Milner, *The Difficulty of Verifying a Program with Unnatural Data Representation*, tech. report 3, Computation Services Dept., Univ. College of Swansea, Jan. 1969; *A Formal Notion of Simulation between Programs*, tech. report 14, Dept. of Computer Science, Univ. College of Swansea, Oct. 1970; and *Program Simulation: An Extended Formal Notion*, tech. report 17, Dept. of Computer Science, Univ. College of Swansea, Apr. 1971.
67. R. Milner, *An Algebraic Definition of Simulation between Programs*, tech. report CS-205, Computer Science Dept., Stanford Univ., Feb. 1971.
68. C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, vol. 1, no. 4, 1972, pp. 271-281.
69. Robin Milner also pointed out (private communication, Feb. 1992) that his submission to the *J. ACM* on this topic was thought by the referees to be too obvious to warrant publication. Milner rightly comments that "things don't have to be difficult to be useful!"
70. C. Strachey, "Systems Analysis and Programming," *Scientific Am.*, vol. 215, no. 3, Sept. 1966, pp. 112-124.
71. Exceptions include C.B. Jones, *Software Development: A Rigorous Approach;* J.C. Reynolds, *The Craft of Programming*, Prentice Hall, 1981; and C. Morgan, *Programming from Specifications*. A summary of the VDM research on data reification is given in C.B. Jones, "Data Reification," *The Theory and Practice of Refinement*, J.A. McDermid, ed., Butterworths, 1989, pp. 79-89.
72. C.B. Jones, "Implementation Bias in Constructive Specification of Abstract Objects," unpublished manuscript, Sept. 1977.
73. T. Nipkow, "Non-deterministic Data Types: Models and Implementations," *Acta Informatica*, vol. 22, no. 6, 1986, pp. 629-661.
74. J. He, C.A.R. Hoare, and J.W. Sanders, "Data Refinement Refined: Resumé," *ESOP 86*, LNCS 213, B. Robinet and R. Wilhelm, eds., Springer-Verlag, 1986, pp. 187-196. See also T. Nipkow, *Behavioural Implementation Concepts for Nondeterministic Data Types*, doctoral dissertation, Dept. of Computer Science, Univ. of Manchester, 1986. Reprinted as UMCS-87-5-3, May 1987.
75. J.V. Guttag, *The Specification and Application to Programming of Abstract Data Types*, CSRG-59, doctoral dissertation, Univ. of Toronto, Computer Systems Research Group, Sept. 1975.
76. S.N. Zilles, *Abstract Specifications for Data Types*, tech. report 11, MIT progress report, 1974.
77. J.A. Goguen et al., *Initial Algebra Semantics*, tech. report RC 5243, IBM, 30 Jan. 1975.
78. F.L. Morris, "Advice on Stucturing Compilers and Proving them Correct," *ACM Symp. Principles of Programming Languages*, ACM Press, 1973, pp. 144-152, and J.H. Morris, "Types are not Sets," *ACM Symp. Principles of Programming Languages*, ACM Press, 1973, pp. 120-124.
79. For a useful historical sketch, see J.A. Goguen et al., "Initial Algebra Semantics and Continuous Algebras," *J. ACM*, vol. 24, no. 1, 1977, pp. 68-95.
80. P. Lucas, "On the Semantics of Programming Languages and Software Devices," R. Rustin, *Formal Semantics of Programming Languages*, Prentice Hall, 1972, pp. 41-57.
81. H. Bekič and K. Walk, "Formalization of Storage Properties," *Symp. Semantics of Algorithmic Languages,* LNM 188, E. Engeler, ed., Springer-Verlag, 1971, pp. 28-61.
82. C.B. Jones, "Formal Development of Correct Algorithms: An Example Based on Earley's Recogniser, *ACM SIGPLAN Notices*, vol. 7, no. 1, Jan. 1972, pp. 150-169.
83. Text books in this area include H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, European Assoc. for Theoretical Computer Science (EATCS) Monographs on Theoretical Computer Science, Springer-Verlag, 1985, and H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1990.

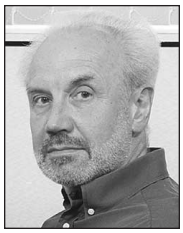
84. D. Bjørner and C.B. Jones, eds., *The Vienna Development Method: The Meta-Language*, LNCS 61, Springer-Verlag, 1978.
85. More recent publications are D. Bjørner and C.B. Jones, *Formal Specification and Software Development*, Prentice Hall, 1982; C.B. Jones, *Systematic Software Development Using VDM,* Prentice Hall, 1986; and C.B. Jones, "VDM Proof Obligations and their Justification," *VDM'87—A Formal Definition at Work*, LNCS 252, D. Bjørner et al., eds, Springer-Verlag, 1987, pp. 260-286.
86. J.-R. Abrial and S.A. Schuman, "Non-deterministic System Specification," *Semantics of Concurrent Computation: Proceedings,* LNCS 70, G. Kahn, ed., Springer-Verlag, 1979, pp. 34-50; also J.-R. Abrial, S.A. Schuman, and B. Meyer, "Specification language," *On the Construction of Programs*, R.M. McKeag and A.M. Macnaghten, eds., Cambridge Univ. Press, 1980, pp. 343-410.

    A reference that has not been located is Abrial's paper that was presented at the conference whose proceedings are published as *Mathematical Studies of Information Processing*, LNCS 75, E.K. Blum, M. Paul, and S. Takasu, eds., Springer-Verlag, 1979.
87. Treatments of what Z has evolved into include *Specification Case Studies*, I. Hayes, ed., Prentice Hall, 1986; and J. M. Spivey, *Understanding*, Cambridge Tracts in Computer Science 3, Cambridge Univ. Press, 1988, and *The Z Notation: A Reference Manual*, Prentice Hall, 1989.
88. D.I. Good and W.D. Young, "Mathematical Methods for Digital Systems Development," *VDM'91—Formal Software Development Methods, Vol. 2: Tutorials*, LNCS 552, S. Prehn and W.J. Toetenel, eds., Springer-Verlag, 1991, pp. 406-430.
89. R.M. Burstall and J.A. Goguen, "An Informal Introduction to Specifications Using CLEAR," *The Correctness Problem in Computer Science*, Int'l Lecture Series in Computer Science, R.S. Boyer and J.S. Moore, eds., Academic Press, 1981, pp. 185-214.
90. R.A. DeMillo, R.J. Lipton, and A.J. Perlis, "Social Processes and Proofs of Theorems and Programs," *Comm. ACM*, vol. 22, no. 5, May 1979, pp. 271-280.
91. O-J. Dahl, "Can Program Proving Be Made Practical?" *EEC-Crest Course on Programming Foundations*, M. Amirchahy and D. Néel, eds., IRIA, 1978, pp. 57-114. Also printed as tech. report 33, Inst. of Informatics, Univ. of Oslo. See also C.B. Jones, "Constructing a Theory of a Data Structure as an Aid to Program Development," *Acta Informatica*, vol. 11, no. 2, 1979, pp. 119-137, and more recently, J.V. Guttag, J.J. Horning, and J.M. Wing, *Larch in Five Easy Pieces*, tech. report 5, Digital Equipment Corp. Science Research Center, July 1985, and B. Mueller, "Formal Derivation of Pointer Algorithms," *Informatik und Mathematik*, M. Broy, ed., Springer-Verlag, 1991, pp. 419-440.
92. J.H. Fetzer, "Program Verification: The Very Idea," *Comm. ACM*, vol. 31, no. 9, Sept. 1988, pp. 1048-1063.
93. E.W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages*, F. Genuys, ed., Academic Press, 1968, pp. 43-112.
94. C.A.R. Hoare, "Towards a Theory of Parallel Programming," *Operating System Techniques*, C.A.R. Hoare and R. Perrot, eds., Academic Press, 1972, pp. 61-71.
95. C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," *Comm. ACM*, vol. 17, no. 10, Oct. 1974, pp. 549-557.
96. C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, vol. 21, no. 8, Aug. 1978, pp. 666-677. For a more recent description of CSP, see C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
97. P. Brinch Hansen, "A Programming Methodology for Operating System Design," *Proc. IFIP'74, Information Processing 74*, J.L. Rosenfeld, ed., 1974, pp. 394-397, and "The Programming Language Concurrent Pascal," *IEEE Trans. Software Eng.*, vol. 1, no. 2, June 1975, pp. 199-207. See also P. Brinch Hansen, *Concurrent Pascal—A Programming Language for Operating System Design*, tech. report 10, Information Science, Caltech, Apr. 1974, and "Distributed Processes: A Concurrent Programming Concept," *Comm. ACM*, vol. 21, no. 11, Nov. 1978, pp. 934-941.
98. R. Milner, "A Calculus for Communicating Systems," LNCS 92, Springer-Verlag, 1980. Revised and rewritten as *Communication and Concurrency*, Prentice Hall, 1989.
99. D. Park, "Concurrency and Automata on Infinite Sequences," *Theoretical Computer Science, 5th GI-Conference,* LNCS 104, Springer-Verlag, 1981, pp. 167-183.
100. E.A. Ashcroft and Z. Manna, "Formalization of Properties of Parallel Programs," *Machine Intelligence 6*, B. Meltzer and D. Michie, eds., Edinburgh Univ. Press, 1971, pp. 17-41.
101. S.S. Owicki, *Axiomatic Proof Techniques for Parallel Programs*, doctoral dissertation, Dept. of Computer Science, Cornell Univ., 1975. Published as tech. report 75-251.
102. S.S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs," *Acta Informatica*, vol. 6, no. 4, 1976, pp. 319-340. A related method is described in L. Lamport, "Proving the Correctness of Multiprocess Programs," *IEEE Trans. Software Eng.*, vol. 3, no. 2, Mar. 1977, pp. 125-143.
103. N. Francez and A. Pnueli, "A Proof Method for Cyclic Programs," *Acta Informatica*, vol. 9, no. 2, 1978, pp. 133-157.

104. C.B. Jones, *Development Methods for Computer Programs including a Notion of Interference*, doctoral dissertation, Oxford Univ., June 1981. Printed as Technical Monograph No. PRG-25. These ideas are in C.B. Jones, "Specification and Design of (Parallel) Programs," *Information Processing 83*, R.E.A. Mason, ed., North-Holland, 1983, pp. 321-332, and in K. Stølen, *Development of Parallel Programs on Shared Data-Structures*, doctoral dissertation, Dept. of Computer Science, Manchester Univ., 1990 (also published as tech. report UMCS-91-1-1).
105. J. Misra and K.M. Chandy, "Proofs of Networks of Processes," *IEEE Trans. Software Eng.*, vol. 7, no. 4, July 1981, pp. 417-426.
106. W.-P. de Roever, "The Quest for Compositionality: A Survey of Assertion-Based Proof Systems for Concurrent Programs: Part I: Concurrency Based on Shared Variables," E.J. Neuhold and G. Chroust, *Formal Models in Programming, Proc. Working Conf. The Role of Abstract Models in Information Processing,* North-Holland, 1985, pp. 181-205.

    See also J. Hooman and W.-P. de Roever, "The Quest Goes On: A Survey of Proof Systems for Partial Correctness of CSP," *Current Trends in Concurrency*, LNCS 224, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, eds., Springer-Verlag, 1986, and J. Zwiers, *Compositionality, Concurrency and Partial Correctness: Proof Theories for Networks of Processes, and their Relationship*, doctoral dissertation, Technical Univ. Eindhoven, 1988. Available as LNCS 321, Springer-Verlag.
107. A. Pnueli, "The Temporal Semantics of Concurrent Programs," *Theoretical Computer Science*, vol. 13, 1981, pp. 45-60.
108. H. Barringer, R. Kuiper, and A. Pnueli, "Now You Can Compose Temporal Logic Specifications," *Proc. 16th ACM STOC*, pp. 51-63, ACM Press, 1984.
109. L. Lamport, *A Temporal Logic of Actions*, tech. report 57, DEC, SRC, 1990. See also "The 'Hoare Logic' of Concurrent Programs," *Acta Informatica*, vol. 14, no. 1, 1980, pp. 21-37, and "Control Predicates are Better than Dummy Variables for Reasoning About Program Control," *ACM Trans. Programming Languages and Systems*, vol. 10, no. 2, Apr. 1988, pp. 267-281.
110. H.C. Lauer, *Correctness in Operating Systems*, doctoral dissertation, Dept. of Computer Science, Carnegie Mellon Univ., 1972.
111. K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.
112. C.A. Petri, *Kommunikation mit Automaten*, [Communication with Automation] doctoral dissertation, Univ. of Darmstadt, 1962; also *Nichtsequentielle prozesse*, tech. report ISF-76-6, GMD, 1976, and its translation, *Non-sequential processes*, tech. report ISF-77-05, GMD, 1977. A useful overview is contained in G. Rozenberg, *Advances in Petri-nets*, LNCS 188, Springer-Verlag, 1985.
113. H. Zemanek, "Semiotics and Programming Languages," *Comm. ACM*, vol. 9, no. 3, Mar. 1966, pp. 139-143.
114. J.W. Backus et al., "Revised Report on the Algorithmic Language 60," *Comm. ACM*, vol. 6, no. 1, Jan. 1963, pp. 1-17.
115. For an early view, see S. Gorn, "Specification Languages for Mechanical Languages and their Processors—A Baker's Dozen," *Comm. ACM*, vol. 4, no. 12, Dec. 1961, pp. 532-542.
116. J. McCarthy, "A Formal Description of a Subset of ALGOL," *Formal Language Description Languages for Computer Programming*, T.B. Steel, ed., North-Holland, 1966, pp. 1-12.
117. For background on the IBM Laboratory in Vienna, see P. Lucas, "Formal Semantics of Programming Languages: VDL," *IBM J. Research and Development*, vol. 25, no. 5, Sept. 1981, pp. 549-561.
118. Which might have been called "NPL" had the UK National Physical Laboratory not pointed out that they had a prior claim on the acronym; see *History of Programming Languages,* R.L. Wexelblat, ed., Academic Press, 1981, pp. 551-600.
119. Secondary material includes P. Lucas and K. Walk, *On The Formal Description of PL/I*, vol. 6, Part 3 of Annual Review in Automatic Programming, Pergamon Press, 1969.
120. J. McCarthy and J. Painter, *Correctness of a Compiler for Arithmetic Expressions*, tech. report CS38, Computer Science Dept., Stanford Univ., Apr. 1966. See also *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*, American Mathematical Soc., 1967, pp. 33-41.
121. P.J. Landin, "The Mechanical Evaluation of Expressions," *Computer J.*, vol. 6, no. 4, 1964, pp. 308-320, and "A Correspondence between ALGOL-60 and Church's Lambda-Notation," *Comm. ACM*, vol. 8, no. 2, Feb. 1965, pp. 89-101, 158-165.
122. D. Scott, "A Construction of a Model for the λ Calculus," manuscript, Nov. 1969; "Models for the λ Calculus," manuscript—draft, Dec. 1969; "A Simplified Construction for λ Calculus Models," manuscript, Apr. 1973; *Outline of a Mathematical Theory of Computation*, tech. report PRG-2, Oxford Univ. Computing Laboratory, Programming Research Group, Nov. 1970; and *Data Types as Lattices*, tech. report PRG-5, Oxford Univ. Programming Research Group, Sept. 1976.
123. C. Strachey and D. Scott, "Mathematical Semantics for Two Simple Languages," paper read at Princeton Univ., Aug. 1970.
124. J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

125. H. Bekič et al., *A Formal Definition of a PL/I Subset*, tech. report 25.139, IBM Laboratory, Dec. 1974.
126. H. Bekič, *Programming Languages and Their Definition*, LNCS 177, Springer-Verlag, 1984.
127. P. Naur, "Formalization in Program Development," *BIT*, vol. 22, no. 4, 1982, pp. 437-453.
128. G.D. Plotkin, *A Structural Approach to Operational Semantics*, tech. report DAIMI FN-19, Aarhus Univ., 1981.
129. For example, S. Igarashi, *An Axiomatic Approach to the Equivalence Problems of Algorithms with Applications*, doctoral dissertation, Univ. of Tokyo, 1964.
130. C.A.R. Hoare et al., "The Laws of Programming," *Comm. ACM*, vol. 30, no. 8, Aug. 1987, pp. 672-687. Also, see Corrigenda in vol. 30, p. 770.
131. J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. ACM*, vol. 21, no. 8, Aug. 1978, pp. 613-641.
132. R. Kowalski, "Algorithm = Logic + Control," *Comm. ACM*, vol. 22, no. 7, July 1979, pp. 424-436.
133. For a recent survey, see J.A. Robinson, "Logic and Logic Programming," *Comm. ACM*, vol. 35, no. 3, Mar. 1992, pp. 40-65.
134. G. Kahn and D. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing 77*, B. Gilchrist, ed., 1977, pp. 993-998.
135. D.B. MacQueen, *Models for Distributed Computing*, tech. report 351, INRIA, Apr. 1979.
136. Inmos, *occam 2: Reference Manual*, Prentice Hall, 1988.
137. O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA 67 Common Base Language*, tech. report S-2, Norwegian Computing Center, 1968.
138. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
139. B. Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.
140. P. America, "Issues in the Design of a Parallel Object-Oriented Language," *Formal Aspects of Computing*, vol. 1, no. 4, 1989, pp. 366-411.



Cliff Jones is a professor of computing science at Newcastle University and the project director of the Interdisciplinary Research Collaboration on Dependability of Computer-Based Systems. He received a doctorate under Tony Hoare in Oxford in 1981 and immediately moved to a chair at Manchester University where he built a strong Formal Methods group. Much of his research has focused on formal (compositional) development methods for concurrent systems. In 1996 he moved to Harlequin directing some 50 developers on Information Management projects and finally became overall technical director before leaving to re-join academia in 1999. His interests in formal methods have now broadened to reflect wider issues of dependability. Jones is a fellow of the BCS, the IEE, and the ACM.

Readers may contact Cliff Jones at Cliff.Jones@ncl.ac.uk.