On the Search for Tractable Ways of Reasoning about Programs

C. B. Jones

July 25, 2001

Abstract

This paper traces the important steps in 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. Initially, researchers focussed on ways of verifying that a program satisfies its specification (or that two programs were equivalent). Over time it became clear that *post facto* verification is only practical for small programs and attention turned to verification methods which support the development of programs; for larger programs it is necessary to exploit a notation of compositionality. Coping with concurrent algorithms is much more challenging – this and other extensions are considered briefly. The main thesis of this paper is that the idea of reasoning about programs has been around since they were first written; the search has been to find tractable methods.

1 Introduction

A program can only be judged to be correct –or otherwise– 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; even without branch points which cause repetition, which shows that for programs with upwards of a thousand branch points not even all of their paths can 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 some path is not designed correctly.

If bugs are to be avoided, some technique other than testing must be used to establish that software satisfies its specification. Fortunately, under assumptions which are discussed below, 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. Two attempts to reduce the risk of accepting invalid proofs are touched on below: appropriate structuring of developments to reduce complexity is mentioned in Section 3 and the use of proof support tools is reviewed in Section 4.3.

This paper 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. Section 2 describes in detail the history of work on the verification of sequential imperative algorithms. Over time it became clear that *post facto* verification is limited to rather small programs; Section 3 explains how early results have been applied to change the way programs are developed. Coping with concurrent algorithms is more challenging – this and other extensions are considered in Section 4.

2 Proofs about sequential algorithms

Imperative programs can be thought of in terms of the effects they have on a computer which 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 (1912–54) in his classic paper on the *Entscheidungsproblem* [Tur36][2] introduced the idea of a 'Turing machine' as a thought experiment to prove a deep result about formal systems.

Anyone who has written a program knows that errors are easily made; the larger the program, the greater the risk that errors will not be detected by testing. This section indicates that the pioneers of computer programming were aware of the need to reason about programs in order to ensure that they have desired properties. In most cases, the property sought was to show that a program satisfied a specification. The search has been for tractable notations for specifying and reasoning about programs.

Sections 2.1–2.3 trace the main line of development taking a key publication of Charles Antony Richard Hoare (b1934)[3] as a pivotal point; some related but less central issues are discussed in Section 2.4.

2.1 Pre-Hoare

The fact that it is possible to reason about computer programs was evident to some of the pioneers of electronic computing.[4] Herman Heine Goldstine (b1913) and John von Neumann (1903–57) wrote a paper [GvN47][5] which explains how 'assertion boxes' (see below) can be used to record the reasons for believing that a series of 'operation boxes' have a particular effect.

[1] For several reasons, it is clear that this cannot be a real history; for example, its author is no historian! Biases of the author and a selective knowledge make the current text open to criticism (where conscious that an anecdote is personal, I have used the first person singular). At best, this paper will provide a source for subsequent historical research. One topic which is largely ignored here is that of numerical analysis.

[2] Reprinted in [Dav65, pp115–154].

[3] Normally just Tony Hoare but references show all initials. The original publisher of this historical note asked that full names and year-of-birth of key people were included; these have been preserved in this version.

[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 [Ran75, pp45–47]. Zuse's *Plankalkul* is one of the earliest programming languages (see [Zus84]); Heinz Zemanek has kindly checked with Prof. Zuse and confirms that the concern was shared but that there were no specific provisions for correctness arguments.

[5] Reprinted in [Tau63, pp80–151]; all page number references below are to this version.

The paper begins with a discussion which shows why they believe that the task of coding is non-trivial (cf. pp81–82)

> 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 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.

They then move on to indicate the direction of their proposal (cf. p83)

> 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.

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 (cf. p92)

> 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*.

The description of how the consistency of the operation/assertion boxes is checked is interesting: one example is described as follows (cf. p98)

> 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.

Then, after a discussion of 'approximation processes' and 'round-off errors', they write (cf. p100)

> 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".

For reasons which become clear below, it is also interesting to quote an earlier part of this paper on 'induction' (cf. p84 and p.92)

> 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 induction 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.

This paper clearly 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 which 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 [Tur49][6] presented by Alan Turing at a conference in Cambridge, England, 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.

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 the opening question. The aim is to reason about a program in general not just to reduce errors by detecting exceptional cases with hand tests.

The paper begins with an analogy between the 'carries' in an addition and the 'assertions' which 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 wrote $\lfloor \underline{n}$ for factorial n; 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 [here 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

[6] The printed text of Turing's paper contains so many transcription errors that it took considerable effort to decipher: [MJ84] contains a corrected version and relates it to later work.

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 [here 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).[7]
>
> 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."

Turing's programming language is a hindrance – although the idea of using primed ('dashed') versions of identifiers is returned to below in Section 2.3.

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

[7] Concern with the question of overflow is discussed below as 'clean termination'.

Figure 1: Turing's proof of a factorial routine

> 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.

In comparison to later work, the language of assertions is clearly limited. But the key idea of logical statements which relate values of variables is present. As well as its early date, Turing's paper is remarkable for its economical exposition.

Given the respective dates of [GvN47] and [Tur49] it is tempting to speculate whether Turing knew of the earlier work. Turing visited von Neumann (e.g. 1947, cf. [Hod83, p355]) and it is unlikely that [GvN47] would not 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 the 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 which seems to support this speculation is the use of the term 'inductive variable'; as pointed out by Douglas Hartree in the discussion which followed Turing's talk, this is probably not the most obvious of choices and it is unlikely that both pioneers hit upon it purely by coincidence.

Neither [GvN47] nor [Tur49] appears to have been known to those who most influenced subsequent research on program verification. These early papers indicate that it is tractability which has been 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 former had been designed with manipulation in mind whereas it was difficult to reason about the latter because they had mainly been designed so as to facilitate translation into efficient machine code.[8]

There is no compelling explanation of the gap of a decade in the landmarks of this field. One can guess that this was an era of hardware developments and perhaps that there was a period of optimism that the development of programming languages would make the expression of programs so clear that they would not contain errors. In fact, as the hardware became more copious, the programming task became much more complex. It is also true that there were far more programmers (to make mistakes) at the end of this decade. Perhaps most tellingly, the developments in hardware 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 (b1927) issued a clarion call to investigate a 'Mathematical Theory of Computation' (the most accessible source of a slightly extended version of this contribution is [McC63a]). This paper 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

[8] JAN Lee pointed out that 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 [Wex81, p167].

> that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.

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. The topic of 'Language Semantics' cannot be completely separated from program verification and an outline of the major issues is given in Section 4.2. McCarthy's attention was more on reasoning about recursive functions than the imperative programs which are the focus here: [McC63a] describes 'recursion induction', [McC60] discusses the LISP language, but [McC63b] shows a link between recursive functions and 'Algolic Programs'. This last citation also includes one of the clearest statements of goals. For example

> 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.

There is one problem which is only slightly touched on above: if computer arithmetic is not exact, what are the rules for reasoning about evaluation of expressions? Adriaan van Wijngaarden (1916–87) who was for many years the father figure of Dutch computer science[9] presented a paper in 1964 –published as [vW66]– which contains a careful discussion of the problems of reasoning about finite computer arithmetic and sketches axioms which might support proofs.[10]

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.[11] In content –although in ignorance of the Turing source– this link was made by Hoare five years after van Wijngaarden's talk (see Section 2.2).

The year 1966 saw a crucial step forward: the papers by Robert W Floyd (b1936) and Peter Naur (b1928) have a major influence on subsequent work. It appears that Naur and Floyd developed their ideas independently of each other.[12] Naur's paper [Nau66] 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

[9] Edsger Wybe Dijkstra (b1930) described in his Turing Award lecture [Dij72] how his decision to work in computing was influenced by van Wijngaarden. Dijkstra adds 'One moral of the above story is, of course, that we must be very careful when we give advice to younger people: sometimes they follow it!'.

[10] As mentioned above, 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 [HES72].

[11] 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.

[12] Floyd's acknowledgements to other prior work are discussed below.

Figure 2: An example of Naur's 'General Snapshots'

of Floyd's paper dated May 20, 1966; the normal citation is to [Flo67] which is in the proceedings of a conference which took place in April 1966. 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 the former 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' [Nau66] 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 careful rather than formal. 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 systematic proof procedures, or even the realization that such proof procedures exist, is unknown to the large majority of programmers.

In spite of this clear statement, it is obvious from later writings[13] that Naur views proof as one of many weapons which should be in the armoury of a program designer; he is not interested in formality for its own sake. This might account for the controversy sparked by Naur – see Section 4.2.

[13] Naur was to develop his ideas, together with others on 'action clusters' [Nau69], and to fit them into his thoughtful –but too little known– book [Nau74].

Anyone wishing to form their own picture of the development of research on program verification is strongly advised to read [Flo67]. The challenge faced by all authors discussed in this section is described by Floyd 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.

So Floyd's method is based on annotating a flow chart with assertions (propositions) which relate values of variables. For example, in Floyd's sample program which computes integer division by successive subtraction (see Figure 3), one finds $0 \leq R < Y \wedge X \geq 0 \wedge X = R + Q * Y$ 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 which are concerned with the correct outcome, the ordered pairs are annotations which 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' which ensure that the assertions correspond to the statements in his flow diagrams. In his 1966/7 paper, 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 which is true after the statement is executed. This viewpoint leads to a complicated rule for the assignment statement whose consequent uses an existential quantifier. The discovery of a simpler backward rule was also made by Floyd. This rule was used in the work of King (see Section 4.3) and David Charles Cooper (b1931) was according to Hoare responsible for informing the latter of the backwards rule which he wisely adopted in his key 1969 paper.

Floyd's landmark paper includes a discussion of the verification conditions required for 'An ALGOL Subset' but not only does Floyd give rules for specific language constructs, he also recognises the place of 'general axioms' which should be true of any semantic definition These can be compared with Dijkstra's 'healthiness conditions' for 'predicate transformers' which are discussed in Section 3.1.

As hinted at in the title of [Flo67] ('Assigning Meaning to Programs') the main thrust of the paper is not to present a way of proving programs correct: to quote

> 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 programming languages based on evaluation of recursive functions.

Floyd was unaware of Turing's work; it is interesting to note that the most important difference between Floyd's method and that of Turing is that Floyd

Figure 3: An example of Floyd's 'interpreted programs'

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.[14] It would, for example, not be clear how to state in Turing's system $\neg(\exists i \leq j)(A(i)$ **divides** $x)$ in a prime number program. Although [GvN47] mentions 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 on what follows. Even given that Floyd saw the paper as a contribution to semantics rather than 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.' This reference has been traced[15] to [Gor59]. 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.

2.2 Hoare's axioms

The most widely cited paper mentioned in this section must be [Hoa69].[16] 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 made a comment [Ste66, pp142–3] on 'the need to leave languages undefined' which presages one of the key points in his 1969 paper. At the meeting of IFIP Working Group W.G. 2.1 which followed the Baden conference, Tony Hoare discussed the definition of functions via their properties (e.g. the result of *abs* must be non-negative and equal to either its argument or the negation thereof). In 1965, Hoare attended, with other members of European Computer Manufacturers Association (ECMA) PL/I standardization group (TC10), a course on the language semantics work then being pursued at the IBM Laboratory Vienna. He stayed at the Imperial Hotel on whose notepaper a first sketch of his axiomatic work was written. Notice that this precedes the Naur/Floyd publications. A two-part draft (dated December 1967) of what was to evolve into [Hoa69] was written during a brief period of employment at the National Computer Centre in Manchester. One part of this privately circulated typescript axiomatized execution traces using a partial order and clearly stated the goals of what was to become 'Axiomatic Basis'. Although the objectives were clear in the 1967 paper, there was no coherent

[14] This is presumably the reason that Maurice Wilkes argues [Wil85, p145] that Turing (in [Tur49]) 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.'

[15] With some difficulty and only thanks to Dick Hamlet and Ralph London.

[16] Along with other major papers by Hoare, this is reprinted in [HJ89]. Much of the description which follows was generated for the 'link material' of that book and consists of 'private communications' between Hoare and the current author.

notion of how to axiomatize the programming language concepts discussed. On arrival in Belfast to take up a chair in October 1968, Hoare 'stumbled upon' the mimeographed draft of Floyd's paper.[17] This had a major impact on [Hoa69]. A further two-part draft (dated December 1968) reflects Floyd's ideas and strongly resembles the final journal version [Hoa69]. 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 acknowledgements, 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 [Flo67]. 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.

One reason for the impact of [Hoa69] is its clarity of exposition.[18] Hoare grasped the significance of Floyd's contribution and made a key change in its presentation. Hoare-triples contain a pre- and post-condition 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 the development in [Hoa69] of an Algol-like version of the division by successive subtraction example used in [Flo67]. 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

[17] Peter Lucas recalls having sent Tony 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'.

[18] Given what has been published since, people might find the Floyd article easy reading but I can remember making a trip to Pittsburgh in August 1967 to ask Bob Floyd for help in understanding the mimeographed version of his paper; and when Dana Scott was invited to the Vienna Laboratory in 1969, the original intention was to discuss Floyd's contribution to semantics (the impact of the hand-written note [dBS69] which he brought with him will be described in a joint paper with Joe Stoy on the history of Program Language Semantics.) Furthermore, Maurice Wilkes writes (letter to Tony Hoare, August 1981) '... loop invariants ... The idea is apparently simple, but really rather deep. Once you have grasped it fully – and I must confess that, although Floyd was very patient with me, it was sometime before I did this – you will never look at the subject in the same way again.'

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

where

x is a variable identifier;

f is an expression;

P_0 is obtained from P by substituting[19] f for all 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.' As mentioned above, this backward rule for assignment is far simpler than the forward verification condition in [Flo67]. Hoare heard of the new rule via Cooper who had been on sabbatical in the Carnegie Institute of Technology and gave a talk in Belfast on his return. The axiom schema is only valid under certain assumptions: subtle points about subscripted variables etc. are discussed in [Apt81]. 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 which 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 gives –in [Hoa69]– general rules of consequence (D1) which justify the strengthening of pre-conditions or weakening of post-conditions

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

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

If $\vdash \{P \wedge B\}\ S\ \{P\}$ then $\vdash \{P\}$ **while** B **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

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

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".' 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.

[19] Strictly, the substitution should only be for free occurrences of x.

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 Section 3.

Subsequent papers[20] tackle features of programming languages which were identified as 'areas which have not been covered' in Hoare's 1969 paper. Notably [HW73] 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.[21] But time has shown, at least for languages which were designed without the explicit constraint that they should have an 'axiomatic semantics', the task of providing a complete axiomatization to be impractical. The languages Euclid [LGH+78], and Gypsy [GCH+78] 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 Section 4.2) 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 (see Section 4.4). Even under the narrowest interpretation of Dijkstra's famous letter to the editor [Dij68c] as being just about coding rules, the avoidance of 'goto' statements can be seen as a step towards tractability of programs for reasoning rather than for compilation. In fact the issues raised in the debate about 'Structured Programming' are much wider and [DDH72, Dij68a, Dij71, Wir73, Wir76] have transformed thinking about the programming task.

If more than one form of semantics is used then they must be consistent. (The impact of this observation on the assumptions under which it is possible to claim that a proof provides knowledge about program execution is returned to in Section 3.3.) The issue of proving that inference rules like Hoare's so-called 'axioms' are consistent with a model-oriented semantics is considered in [Lau71, HL74, Don75]. (Peter Lauer was a member of the IBM Laboratory in Vienna and went to Belfast to do a PhD under Hoare's supervision.)

2.3 Post-Hoare

Hoare's 1969 paper not only linked the ideas of van Wijngaarden and Floyd, it also gave the topic of program verification the boost of making a theoretically possible idea into a tractable technique.[22] This section reviews some detailed issues which remain to be finally resolved; the next major step in the history is related in Section 3.

Program Verification ideas –at least in the narrow sense of this section– have made the transition from research papers, via post-graduate textbooks

[20] [Hoa71a, CH72, ACH76, dR74, dR76a, Apt81]

[21] It has often been argued that this goal was over-ambitious. In Hoare's presentation at the April 1969 meeting of IFIP Working Group W.G. 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!

[22] A detailed technical assessment which focuses on the questions of consistency and completeness is given in [Apt81, Apt84]; another important study is [dB80].

like [Dij76], to undergraduate texts.[23] Leaving aside the plethora of notational details, there are at least two substantive issues which distinguish approaches even within this simplest form of program verification.

In most publications, assertions –be they pre-conditions, invariants or post-conditions– are predicates of one state. This seemingly innocent decision causes problems when writing meaningful specifications. For example, the specification of the division task discussed above 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 which is specifically intended to change a large data structure (e.g. in-place sorting or matrix inversion) it is not possible to get by with post-conditions of the final state alone. There are a number of techniques, such as using auxiliary variables to record initial values, but these should be compared to the effect of making the step to post-conditions (and other assertions) of two states. This was done as early as [Man68] and it is something which has distinguished VDM (see Section 3.3) from most other approaches since [Jon73]. Seeing this as the natural way to present specifications, a set of proof rules were tolerated in [Jon80] which were clumsy in comparison to those in [Hoa69]. Fortunately, Peter Henry George Aczel (b1941) wrote an extensive, but unpublished, note in January 1982 which shows how an elegant set of rules can be given for proofs using predicates of two states.[24] In fact, most of the rules are extensions of those for the single state case.[25] Among other changes, Aczel showed that it is more convenient in a post-condition to use $\overleftarrow{x}/x$ rather than x/x' for the old/new values of state variables.[26]

The other significant issue to which there still appear to be divergent approaches is the way in which 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. Non-denoting terms can arise from recursive definitions and from the basic operators of data types like sequences. The need to prove results about logical formulae which include terms using partial functions casts doubt on the applicability of classical logic. (McCarthy –in [McC63a]– was one of the first to recognise this problem.) Approaches to handling partial functions which 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 non-classical logic. The most frequently used non-classical logic adopts non-strict (but also non-commutative) conditional versions of the propositional operators; alternatively one can use commutative, non-strict, and/or operators as defined

[23] For example [Gri81, Bac86].

[24] Aczel writes of the specifications in [Hoa69] '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 [Jon80] ... His rules appear elaborate and unmemorable compared with the original rules for partial correctness of Hoare.'

[25] The resulting rules are used in more recent VDM publications [Jon86]. Others including [Heh84] use similar specifications; [CdRZ91] gives a clear and concise comparison of the one and two state approach; [Tar85] goes so far as to use pre-conditions which are predicates of two states.

[26] This led to some embarrassment for the current author when at a seminar in Newcastle-upon-Tyne in 1982, Hoare gave a talk which accepted post-conditions 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 $\overleftarrow{x}, x$.

in Kleene's 'strong truth-tables'. A consistent and complete proof system for such a logic has been used in VDM. This 'Logic of Partial Functions' is presented in [BCJ84] which also analyzes approaches to reasoning about partial functions.[27]

Another area of research which should be mentioned here is various notions of induction. Key results by David Michael Ritchie Park[28] (1935–90) [Par69], Hans Bekič (1936–82) and Dana Scott (b1932) 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 airing here. Rodney Martineau Burstall (b1934) showed the usefulness of this rule in his talk at the 1967 Yorktown Heights conference and it was published as [Bur69] with connections to 'recursion induction' which were 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.

2.4 Other work

As can be seen by the citations, the Floyd-Hoare contribution traced above shows the main flow of development of the research; it is not necessarily a value judgement on the work to collect together in this section some of the papers which have had less influence sometimes just because of an accident of timing etc.

Ralph London produced a string of early papers [Lon64, Lon70c, Lon70a, Lon70b, Lon70d, Lon71] on verification topics. One early avenue of research into programs involved proofs about the equivalence of 'schemas' which are like flowcharts with uninterpreted operations in their boxes. The majority of the results are about (un)decidability. Early papers by Russian (e.g. [Yan58]) and Japanese (e.g. [Iga64]) scientists are important here.[29] In November 1967 at a conference on 'Mathematical Theory of Computation' at IBM's Yorktown Heights research laboratory, Manna's methods were enthusiastically commended by John McCarthy who used them as one argument for a move away from the work on schemas as represented in [LPP70]. In as much as papers no longer appear on this topic, time has shown McCarthy to be right, but the audience witnessed a robust defence of their work by Mike Paterson (b1942) who had just completed his Cambridge (U.K.) PhD [Pat67] under the supervision of David Park. Paterson's research interests moved to focus on complexity theory which was a serious loss for verification research. Work on 'algebraic semantics' (see below) is an interesting echo of the schema research.

David Cooper has been mentioned above in connection with his sabbatical at the Carnegie Institute of Technology. He wrote a paper which became available at Carnegie in 1965 –but was published as [Coo66]– which considers the equivalence of iterative and recursive forms of programs. This paper had an influence on the work on 'transformational development' methods. This is an alternative approach to the formal development of programs. The basic idea is to begin with a very simple program as a specification. If this program is

[27] See also [Luk20, Sco67, MM69, Bla80, Che86, Hoo87, Bli88, Avr91, KTB88, CJ91].

[28] Normally just David Park. In order to avoid confusion, all references show only one initial.

[29] Useful reviews are contained in [Rut64, dB69, Ers71] and, more recently, [Ers90, pp267–274].

written in a sufficiently constrained language its correctness should be obvious. The reason one cannot stop there is that such a program is likely –in for example some purely functional language– to be hopelessly inefficient; but it can be used as the starting point for a series of transformations which yield a usable implementation. The first major piece of work in this area appears to be by John Darlington (b1947) whose PhD research (his thesis is [Dar72])[30] was supervised by Burstall. Cooper also presented a paper (published in the proceedings [Coo67]) at the first Annual Machine Intelligence Workshop in 1965 which lists a series of 'things it would be nice to find out about programs': these include 'Does the program solve a given problem', 'Is the program equivalent to another' and 'Given two equivalent programs which one is more efficient'. The paper also cites early work by London [Lon64] and Evans [Eva65][31] on proofs about recognisers. Cooper became professor of computing at Swansea University in 1967; one of his first research assistants was Robin Milner (from October 1968 to December 1970)!

Zohar Manna (b1939) in his thesis [Man68] written at Carnegie Institute of Technology (later Carnegie Mellon University) expresses his debt for guidance in his research to Floyd (his supervisor) and Perlis. Printed in April 1968, this two-part report makes clear the role of Predicate Calculus in reasoning about 'abstract programs'. In particular, Manna focuses on the problem of termination and shows it to be semi-decidable. Manna's has perehaps had more impact than the citations to [Man68] suggest.[32]

Rod Burstall has consistently produced seminal contributions and one of his proposals is the idea of 'intermittent assertions' which for some tasks produce clearer proofs than Floyd's assertions which are linked to the flow of control. His invited talk at IFIP'74 [Bur74] additionally broke the mould of using predicate calculus alone by introducing the use of 'temporal logic' (see Section 4.1).

3 Formal development methods

A crucial test for program development ideas is whether they scale – that is, 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 which are now applied in industry.

3.1 Stepwise design

Given methods –like those in [Hoa69]– for reasoning about programs, it was natural to attempt 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 construct a proof of a complete program was not an adequate response to the challenge of creating non-trivial programs. In fact, one of the first people to whom this became evident was Hoare himself. One of his earliest publications

[30] See also [DB76, BD77]. The most sophisticated system for transformational development is 'CIP' ('computer-aided intuition-guided programming') from the group at the Technische Universität in Munich [BW82, CIP85, CIP87]. Current references on this topic include [SA89, Par90].

[31] [ed.] These citations taken from secondary material.

[32] His text book [Man74] was more influential as has been his subsequent research on Temporal Logics.

had concerned efficient programs related to sorting: he christened his famous sorting program 'Quicksort' [Hoa61]. After publishing [Hoa69] he attempted to use the same method of presenting a program and then reasoning about it to justify the 'FIND' program which 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 baulked 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 –published as [Hoa71b]– relies on the same rules as [Hoa69] but uses them to justify one stage of development at a time.[33] Although the specifications of sub-operations were not completely formalized, a crucial step towards development methods (like those discussed in Section 3.3) had been made. A *post facto* proof that a program satisfies its specification is only possible for a correct program; moreover, it is unlikely that attempts to construct such proofs for other than trivial programs provides a cost-effective way of detecting errors. The alternative of making one step of development and justifying it before further work is done 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, it too can only be applied once code has been developed.) Once the move to a formal stepwise process has been made, the potential is there 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 which are only detected in testing.

In order 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 post-conditions which 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 pre-condition is true the implementation both terminates and yields a state for which the post-condition is true. The same notion of satisfaction can be used in discussing specifications of any sub-components. 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 sub-tasks $S1$ 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 which satisfy specifications $S1$ and $S2$ can be combined to yield an implementation which satisfies S. In other words, the specifications of the sub-components tell their developers all that they need to know about their task.[34] (Not only is this easy to illustrate for sequential programs, it is also not difficult to achieve – Section 4.1 explains why the issue of compositionality is a significant complication in the development of concurrent programs.)

[33] Further applications are in [FH71, Hoa72a].

[34] The researchers working on Gypsy used the suggestive term 'Independence Principle'.

Being able to justify stages of design was a significant step forward 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 which 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 above; Dijkstra proposed regarding all program constructs as ways of transforming a (required) post-condition into a 'weakest pre-condition' which would guarantee successful termination [Dij75]. 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 pre-condition under which $r := r - y;\ q := 1 + q$ will terminate and deliver a state for which $x = r + y * q$ is true. The weakest pre-condition is just $x = r + y * q \wedge y \leq r$ but the change of viewpoint has some interesting consequences. Firstly, there is a nice calculus of such predicate transformers. Writing $wp(S, R)$ as the weakest pre-condition for statement S to achieve a state satisfying R,

$$wp((S1; S2), R) = wp(S1, wp(s2, R))$$

Dijkstra's claim in [Dij75] to have 'tightened things up a bit' (over [Hoa69]) is more than justified: he handles total correctness (including termination) where Hoare's axioms only covered partial correctness; Dijkstra's *guarded commands* embody and prompt a clear handling of non-determinacy; there is a strong anti-operational flavour to Dijkstra's explanations.[35]

To return to the point about helping a programmer discover valid steps of development, Dijkstra's own monograph [Dij76] goes a long way to teaching how one can discover correct designs.

In addition to providing predicate transformer semantics for a specific programming language which contains the main 'Structured Programming' constructs, [Dij75] also presents *healthiness conditions* which all predicate transformers should satisfy.[36]

3.2 Richer data structures

For a specification to be a useful starting point for development, it ought be shorter and clearer than a program which satisfies it. Post-conditions 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 of the preceding sections have used variables whose values are numbers; but many computing problems can only be discussed

[35] Not only did the Blanchland (October 1973) meeting of IFIP W.G. 2.3 have a part to play in the development (cf. Mike Woodger's nice history in [Gri78, pp1–5]), but the firm rejection of arguments presented in an unnecessarily operational way has remained a hallmark of these gatherings.

[36] A full discussion of predicate transformers can be found in [DS90]; see also [dR76b]. The so-called 'refinement calculus' is a development (described [Bac80, Mor90]) whose impact goes beyond the 1990 cutoff for this paper.

in terms of values which have more elaborate structure. Finding ways of representing such values in linearly addressable store 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 one can easily determine its properties. It became clear in the 1960s that specification and design methods would have to be able to cope with richer data structures than numbers. In particular, the necessity was seen to write specifications in terms of some abstract class of states and then find ways of relating 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 in order 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 (see Section 4.2). In their cooperation with IBM's UK Laboratory, two different ways of modelling access to local variables had arisen. There was concern as to whether the two 'definitions' were equivalent. Peter Lucas (b1935) undertook a proof [Luc68] that the two models defined the same language using a 'twin machine' which combined two different representations of the underlying state. A predicate which defined valid elements of this cross product essentially fixed a relation between the two representations. This prompted Clifford Bryn Jones (b1944) to write another Vienna report [Jon70] which pointed out the advantage of a functional relationship between the representation and abstraction (i.e. there can be many possible representations for the same abstract value). Arthur John Robin Gorell Milner (b1934)[37] had worked on this problem in Swansea from 1969 (see for example [Mil69, Mil70, Mil71b] and he presented a careful algebraic view of simulation in a Stanford technical report [Mil71a]. Hoare's paper on the topic [Hoa72b] is one of the most common citations in this area. He also uses a function from representation to abstraction as the basis of his set of proof obligations. Hoare acknowledges Robin Milner's influence but the value of historiography in this area is diminished when one knows that Christopher Strachey (1916–75) described the idea to Burstall in the mid-60s but considered it so obvious as to be not worth writing down.[38] Some indication of how Strachey used the idea of developing programs via abstract data objects can be seen in [Str66] where he outlines the design of a program to play draughts (checkers). In view of the date of this contribution, it is 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

[37] Normally just Robin Milner. In order to avoid confusion, all cited references show only one initial.

[38] Robin Milner also pointed out (private communication February, 1992) that his submission to the Journal of the 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!'

> 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.

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 latter to the former. These functions are called *abstraction* (or *retrieve*) functions and they can be used to formulate correctness conditions for program development steps which 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[39] give prominence to this topic.

There are a number of technicalities which have become clearer over time. The approach described above is often called *model-oriented* (in contrast to the property-oriented approach described at the end of this section). There is a risk that a model can be chosen which is overly concrete in the sense that it conserves non-essential information. One can make this notion of *bias* quite precise [Jon77]. The purpose of a specification is to fix a behaviour which 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 which 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 non-determinacy (which need not be present in an implementation) and whose states contain information which can also be dropped when designing implementations. Proof rules which resurrect the idea of relations between abstraction and representation have been developed in [Nip86b, HHS86].[40] In a sense made precise in the papers, these rules are complete.

Here, model-oriented specifications are described first because it fits 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 above that a model-oriented

[39] Exceptions include [Jon80, Rey81, Mor90]; a summary of the VDM research on data reification is given in [Jon89].

[40] See also [Nip86a].

specification can contain unnecessary detail led researchers to seek specifications which 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 to the approach 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 behaviour 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 [Gut75, Zil74, GTWW75]. (See also [Mor73b] and [Mor73a].)[41] In fact [Luc72] (there was also an internal paper by Lucas and Walk in 1969) 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 above but it is also interesting to note 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 (the most accessible source for this material, with references to the 1969 version of the PL/I definition, is [BW71]); [Jon72] also uses a correctness notion which 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 which 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 which are nondeterministic are issues on which full agreement is absent even now.[42]

It is not useful to view the model/property-oriented split as a schism: they are concepts each of which has its domain of applicability; there are data types which are naturally viewed as collections of (immutable) values and there are others whose description is best given in terms of a state. There are, however, some interesting technical questions about the inter-relationship of the two approaches. 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 generators of the type. Viewed from the property-oriented world, one could say that model-oriented specifications are ones which use a particular set of hidden functions (the operators of the underlying types such as sets and mappings).

3.3 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

[41] A Useful historical sketch can be found in [GTWW77, p69].

[42] Text books in this area include [EM85, EM90].

a coherent notation and set of 'proof obligations'. The so-called 'Vienna Development Method' (VDM) was one of the earliest attempts to create such coherence. The history of the IBM Vienna Laboratory and its contribution to work on formal methods cannot be fitted into this paper; 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 [BJ78, Jon80] (the former is concerned with denotational semantics and its use in the development of compilers; the latter relates to general program development).[43] VDM specifications are model-oriented; a module defines a class of states and a series of operations which change and depend on those states; precise notions of operation decomposition and data reification are part of the development method. VDM has been used in organisations far beyond that which originated it.[44] Another specification language which is closely related to VDM is known as 'Z'. Jean-Raymond Abrial (b1938) was the originator of Z but little was published other than [AS79, ASM80][45] in its early form. Researchers who continued to develop and apply Z at Oxford University have published a number of books.[46] 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 [GY91].

One key technical problem is the way in which large specifications can be built up from components. 'Clear' [BG81] provides a valuable starting point but the problem is still an area of active research (and disagreement).

3.4 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: [DLP79], for example, asserts that the concept of proof in mathematics has been misunderstood in work on program verification. Acceptance of proofs in mathematics –it is argued– is a social process with putative proofs being subjected to public scrutiny; the lengthy and detailed logical derivations used to justify some (steps of development of) programs are quite unlike proofs in mathematics. As with most criticism one can discern some truth in this argument although the debate which was sparked off by this 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 residual errors. There is also a message hidden here in the need to structure knowledge about computing science. Ole-Johan Dahl (b1931) was one of the first people to appreciate that formal development of programs could only make progress if knowledge were accumulated from one

[43] More recent publications are [BJ82, Jon86, Jon87].

[44] See, for example, proceedings of the VDM Symposia [BJMN87, BMJ88, BHL90, PT91a, PT91b].

[45] A reference that has not been located is his paper which was presented at the conference whose proceedings are published as [BPT79].

[46] Treatments of what Z has evolved into include [Hay86, Spi88, Spi89].

application to another in the form of theories. Papers such as [Dah78, Jon79][47] 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 80s, a violent debate in the correspondence pages of CACM was caused by [Fet88] 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 one 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. There does of course remain the danger of physical failure of the hardware but –at least as far as undetected errors are concerned– micro-electronic devices are now far more reliable than mechanical systems. If the point is simply to make everyone aware that absolute reliability is unattainable, then one can only hope the message is understood by those who devise weapon systems etc.

In fact, there is a much more serious issue which has probably received too little attention in spite of it 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 to purpose'; one cannot repeat often enough 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 which is not susceptible to mathematical argument.

4 Other issues

As indicated in the introduction, there are many issues which relate to program verification but which are not explored in as much detail as the basic theory of sequential programs above. This section touches on four such issues; here, secondary literature is cited in order to provide pointers to the fields.

4.1 Concurrent programs

Sequential programs can be considered to run in isolation. In reality this is rarely the case on modern computers: the resources of the machine 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 a very interesting class of programs where concurrency is an issue. This class of programs contains many which are very important to society. Programs which handle airline seat reservations might have to handle messages from many agents who are negotiating for the same resource of places on flights. Another area of

[47] More recently: [GHW85, Möl91].

difficulty comes from concurrent programs which have to respond at particular times to stimuli which come from physical sensors. To stay in the same field, programs which 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 paper 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 itself 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. So formal methods for verifying concurrent programs are vitally important. Unfortunately no consensus has evolved on how best to handle concurrency. In part, this is almost certainly due to the diverse nature of the tasks to be covered but it must also be seen as a symptom of a field which is still trying to find tractable methods. Here, we can do no more than point out some key references.

Edsger 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 is contained in [Dij68b] which uses 'semaphores' in order to control concurrent processes. Dijkstra's privately circulates a series of 'EWD' notes which 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 towards making concurrent programs more tractable can be seen in the work of Tony Hoare. 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' [Hoa72c] and 'monitors' [Hoa74] are progressively richer constructs whose structure rules out certain sorts of potential error. (Hoare's concept of monitor 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) [Hoa78][48] 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 – neither he nor the current author would pretend that this work was done in a vacuum. In fact, the development

[48] A more recent description of CSP can be found in [Hoa85].

of the monitor idea is closely paralleled in the work of Per Brinch Hansen – see [Bri74b, Bri75].[49] Robin Milner's CCS (Calculus of Communicating Systems) was presented in [Mil80][50] and 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 focussed on notions like observation equivalence of 'agents' (processes). Park proposed the important concept of 'bisimulation' [Par81].

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 to see whether they can form 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 sub-components which are to be executed in isolation, interfering programs are very difficult to specify in a way which facilitates separate development.

In [AM71] (first available as a Stanford report in February 1970) the fact that two processes can interfere with each other's states is reflected by constructing an equivalent non-deterministic (sequential) program which 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 (b1947) wrote her thesis under the supervision of David Gries (b1939). Owicki's thesis [Owi75] broke the task of verifying concurrent programs into two stages: the first treats them as sequential programs and the second checks whether the proof of process A can be interfered with by statements in process B – and *vice versa*.[51] 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 which 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: [FP78] specifies 'interference' but does not present a development method in the sense suggested here; [Jon81][52] uses rely and guarantee conditions to provide a partial specification of interference; a rely-guarantee approach was published independently in [MC81]. A significant

[49] See also [Bri74a, Bri78].
[50] Revised and rewritten as [Mil89].
[51] See also [OG76]; a related method is described in [Lam77].
[52] This is developed in [Jon83, Stø90].

literature has developed on compositional methods and a good overview is given in [dR85].[53]

The proof methods discussed so far have nearly all been based on classical logic (First-Order Predicate Calculus). As mentioned above, Burstall suggested [Bur74] that some form of Temporal Logic could be used in the development of programs. It would appear that Amir Pnueli (b1941) was the first person to suggest that Temporal Logic be used in specifying and developing concurrent programs [Pnu77][54]. Within the temporal logic framework, the issue of compositionality is tackled in [BKP84].

A major contributor to formalisms for concurrency has been Leslie Lamport (b1941): amongst other contributions he outlined the issues of 'safety' and 'liveness' in [Lam77] and proposed a novel 'Temporal Logic of Actions' in [Lam90].[55] An early attempt to use formalism on operating systems is [Lau72].

A number of putative development methods have evolved for concurrent programs. The Unity method of [CM88] is perhaps the most widely discussed.

As mentioned at the beginning of this section, the work on parallel systems is quite diverse. One contentious issue that has emerged is the contrast between interleaving and 'true concurrency'. The latter position is taken by Carl Adam Petri (b1926) and others working on Petri-nets [Pet62, Pet76, Pet77].[56]

4.2 Language semantics

A separate historical paper on 'Language Semantics' needs to be written but an outline of the major issues is given in this section.

The study of languages can be partitioned into 'syntax', 'semantics' and 'pragmatics'; Heinz Zemanek (b1920) applied this terminology to programming languages in [Zem66]. The valid strings of a language and some suggestion of their structure are defined by its syntax. The history of this area has little to do with the main topic of the current paper; suffice it to say that an adequate method for defining the syntax of programming languages (Backus-Naur-Form) was employed in the description of Algol 60 in [BBG+63]. Finding ways of defining the semantics, or meaning, of strings is much more challenging.[57] (The topic of pragmatics is not pursued here.) 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 which runs on a computer and interprets statements so as 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

[53] See also [HdR86, Zwi88].
[54] Later published in journal form as [Pnu81].
[55] See also [Lam80, Lam88].
[56] A useful overview is contained in [Roz85].
[57] For an early view, see [Gor61].

mathematical functions about which it is possible to reason. One of the earliest papers was by McCarthy: [McC66] describes only a tiny programming language but, together with ideas from Calvin C. Elgot (1922–80) and Peter John Landin (b1930), 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 [Ste66] capture the key discussions. Many of Zemanek's Vienna group[58] were involved and they went on to tackle the semantics of a very large and complex language: PL/I.[59] The Vienna group called their descriptive style 'Universal Language Definition' but their notation became known as 'Vienna Definition Language' (VDL).[60] 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 as was shown in [MP66] and for larger languages in papers from the Vienna group such as [Luc68].

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, the meaning of a program can be discussed independently of particular input values. This can make the task of proving implementations or other general results easier. Strachey and Landin [Lan64, Lan65] made major early steps towards denotational semantics; problems of program components whose denotations are functions of a type whose domain includes its own type posed foundational problems which were solved by Dana Scott. Scott actually discovered his models of 'reflexive domains' whilst in Oxford and the Programming Research Group –led by Strachey– made denotational semantics into a major research topic [Sco69a, Sco69b, Sco73, Sco70, SS70, Sco76] with [Sto77] being a classic of exposition. Interestingly, the Vienna group went on to adopt the denotational approach and develop VDM [BBH+74, BJ78, Bek84].

Here again there has been controversy. Peter Naur made an attack on formalization in [Nau82]. Since the current author's work is cited, it is not fair to enter into further discussion of this controversy here. Suffice it to say that Naur's position ever since his own early contributions appears to have been that formalism should be one tool and should 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 finding appropriate denotations. Furthermore, the difficulties in reasoning about operational definitions can be greatly reduced if care is exercised to avoid unnecessary mechanistic features. Gordon David Plotkin (b1946) made proposals for 'structured operational semantics' [Plo81] which provide a widely used notation for describing programming languages.

[58] For background on the IBM Laboratory in Vienna see [Luc81, Luc87, Zem74] and Zemanek's personal recollections in [Chr85, pp13–41]. A fuller account of the Vienna history should be given in the planned paper on 'Language Semantics'.

[59] 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 [Wex81, pp551–600]).

[60] Secondary material includes [LW69].

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, Section 2 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 which avoids the need for models is to describe semantics by giving sets of equivalences over texts of the language. Examples of this approach include [Iga64, dB68, Bac78, RH86, HHH+87]. 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.

4.3 Machine supported verification

The concept of a formal proof is definable in terms of symbol manipulation operations and it is therefore not surprising that theorem proving was an early task considered for machine support: key papers include [NS56, McC59, Wan60, Rob65, Goo70, dB70].[61] There is a whole spectrum of approaches ranging from proof checkers to attempts to write programs which discover proofs. Proof checkers are easy to write but require that a user generate a complete (and hideously detailed) proof before they can be of use. Automatic theorem provers are –for non-trivial theories– impossible to build in general and have become a major challenge in 'Artificial Intelligence'. The best known development in the use of heuristics is known as the Boyer-Moore theorem prover [BM79] but even this appears to be difficult to control.

Many researchers have sought to find a balance where a user controls the strategy and the machine performs symbol manipulation.[62] The most influential work in this area has been LCF [Mil72, GMW79] on which many subsequent systems have been based.[63] Of particular interest is the series of PRL systems – see [C+86].

4.4 Novel languages

The store 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 towards 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 store were computed by a compiler. This is just one example of many where the symbol manipulation powers of computers have been used to aid the task of using those same machines. After the first small steps were made away from the raw interface of the machine, people began to investigate the question of whether radically different languages could overcome the problem of designing programs which solve problems.

Languages which reflect the idea that the store of a computer is changed by some updating operation are known as 'imperative' and in most such languages

[61] See also [Luc67, Kin69] – all citations taken from secondary material [ed.].
[62] Useful survey articles are [Cra85, Kem86, Lin88].
[63] For a recent book, see [Pau87].

the updating is performed by assignment statements. Assignment statements have the undesirable property that the so-called variables of a program contain different values at different points in the execution of a program. (A problem compounded by languages like FORTRAN where the updating operation was written as = resulting in statements like $X = X + 1$ which 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 seen above, 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 which are purely functional. The hope is that programs in such languages would be much easier to reason about – a point of view strongly presented in [Bac78]. This hope is perhaps easier to realize than the ensuing challenge of finding purely functional languages which can be efficiently translated to machine code. At the time of writing, 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 in [Kow79] 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 Programme' was to make machines more usable by exploiting languages like 'Prolog'.[64] 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 (non-procedural) 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? It is clear from the above discussion that one reaction to finding a (class of) language(s) intractable is to change the language – 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 choosing new abstractions which are more tractable than their predecessors and then developing theories which match the new reality which 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

[64] A recent survey is contained in [Rob92]; Alan Robinson also wrote the key paper [Rob65].

communication based concurrency. One of the first steps away from shared variable concurrency was in the languages which provided queued message passing (e.g. [KM77, Mac79]); some hint of the history of process algebras is discussed above in Section 4.1; 'occam' [INM88, RH86] 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 those which are called 'object-oriented'. Their genesis is Dahl and Nygaard's Simula'67 language [DMN68]. Languages like Smalltalk-80 [GR83] and Eiffel [Mey88] are being recognised as having potential to increase the productivity of programmers; there is now progress in studying concurrency in this framework [Ame89].

Acknowledgements

This text is an extended version of a paper which was to have been published by *Instituto della Enciclopedia Italiana* as *La verifica dei programmi* (*Program Verification*) in the Volume *Logica* of *Storia del XX Secolo* (*History of the Twentieth Century*).

I am grateful to Richard Giordano, Don Good, Ian Hayes, JAN Lee, Peter Lindsay, Robin Milner, José Nuno Olivera, Brian Randell and Mike Woodger, for comments on drafts of this paper; 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 JAN 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

[ACH76] E. A. Ashcroft, M. Clint, and C. A. R. Hoare. Remarks on "Program proving: Jumps and functions". *Acta Informatica*, 6:317–318, 1976.

[AM71] E. A. Ashcroft and Z. Manna. Formalization of properties of parallel programs. In B. Meltzer and D. Michie, editors, *Machine Intelligence, 6*, pages 17–41. Edinburgh University Press, 1971.

[Ame89] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1:366–411, 1989.

[Apt81] K. R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3:431–483, 1981.

[Apt84] K. R. Apt. Ten years of Hoare's logic: A survey – part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.

[AS79] J.-R. Abrial and S. A. Schuman. Non-deterministic system specification. In *[Kah79]*, pages 34–50, 1979.

[ASM80] J.-R. Abrial, S. A. Schuman, and B. Meyer. Specification language. In R. M. McKeag and A. M. Macnaghten, editors, *On the Construction of Programs*, pages 343–410. Cambridge, 1980.

[Avr91] A. Avron. Natural 3-valued logics – characterization and proof theory. *Journal of Symbolic Logic*, 56(1):276–294, March 1991.

[Bac78] J. Backus. Can programming be liberated from the von Neuman style?: a functional style and its algebra of programs. *Communications of the ACM*, 21:613–641, 1978.

[Bac80] R. J. R. Back. Correctness preserving program refinements: Proof theory and applications. Technical report, Mathematisch Centrum Tract, 131, 1980.

[Bac86] R. C. Backhouse. *Program Construction and Verification*. Prentice-Hall International, 1986.

[BBG+63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[BBH+74] H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A formal definition of a PL/I subset. Technical Report 25.139, IBM Laboratory Vienna, December 1974.

[BCJ84] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[Bek84] H. Bekič. *Programming Languages and Their Definition*, volume 177 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[BG81] R. M. Burstall and J. A. Goguen. An informal introduction to specifications using CLEAR. In *[BM81]*, pages 185–214. 1981.

[BHL90] D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors. *VDM'90: VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[BJ78] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[BJ82] D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982.

[BJMN87] D. Bjørner, C. B. Jones, M. Mac an Airchinnigh, and E. J. Neuhold, editors. *VDM'87 – A Formal Definition at Work*, volume 252 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you can compose temporal logic specification. In *Proceedings of 16th ACM STOC*, pages 51–63, Washington, April–May 1984.

[Bla80] S. R. Blamey. *Partial Valued Logic*. PhD thesis, Oxford University, 1980.

[Bli88] A. Blikle. Three-valued predicates for software specification and validation. In *[BMJ88]*, pages 243–266, 1988.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[BM81] R. S. Boyer and J. S. Moore. *The Correctness Problem in Computer Science*. International Lecture Series in Computer Science. Academic Press, London, 1981.

[BMJ88] R. Bloomfield, L. S. Marshall, and R. B. Jones, editors. *VDM'88: VDM – The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[BPT79] E. K. Blum, M. Paul, and S. Takasu, editors. *Mathematical Studies of Information Processing*, volume 75 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Bri74a] P. Brinch Hansen. Concurrent Pascal – a programming language for operating system design. Technical Report 10, Information Science, Cal. Tech., April 1974.

[Bri74b] P. Brinch Hansen. A programming methodology for operating system design. In J. L. Rosenfeld, editor, *Information Processing 74*, pages 394–397, 1974. Proceedings of IFIP'74.

[Bri75] P. Brinch Hansen. The programming language concurrent Pascal. *IEEE Transactions on Software Engineering*, 1:199–207, June 1975.

[Bri78] P. Brinch Hansen. Distributed processes: A concurrent programming concept. *Communications of the ACM*, 21:934–941, 1978.

[Bur69] R. M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12:41–48, 1969. Earlier available as Experimental Programming Report, No. 17, DMIP, Edinburgh, 1968.

[Bur74] R. M. Burstall. Program proving as hand simulation with a little induction. In J. L. Rosenfeld, editor, *Information Processing 74*, pages 308–312, 1974. Proceedings of IFIP'74.

[BW71] H. Bekič and K. Walk. Formalization of storage properties. In E. Engeler, editor, *[Eng71]*, pages 28–61. 1971.

[BW82] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer-Verlag, 1982.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[CdRZ91] J. Coenen, W.-P. de Roever, and J. Zwiers. Assertional data reification proofs: Survey and perspective. In J. M. Morris and R. Shaw, editors, *4th Refinement Workshop*, pages 97–114. Springer-Verlag, 1991.

[CH72] M. Clint and C. A. R. Hoare. Program proving: Jumps and functions. *Acta Informatica*, 1:214–224, 1972.

[Che86] J. H. Cheng. *A Logic for Partial Functions.* PhD thesis, University of Manchester, 1986.

[Chr85] G. Chroust, editor. *Heinz Zemanek – Ein Computerpionier.* R. Oldenbourg, 1985.

[CIP85] CIP Language Group. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[CIP87] CIP System Group. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[CJ91] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[Coo66] D. C. Cooper. The equivalence of certain computations. *BCS, Computer Journal*, 9:45–52, 1966.

[Coo67] D. C. Cooper. Mathematical proofs about computer programs. In N. L. Collins and D. Michie, editors, *Machine Intelligence, 1*, pages 17–28. Olliver and Boyd, 1967.

[Cra85] D. Craigen. A technical review of four verification systems: Gypsy, Affirm, FDM and Revised Special. Technical Report FR-85-5401-01, I. P. Sharp Associates, August 1985.

[Dah78] O-J. Dahl. Can program proving be made practical? In M. Amirchahy and D. Néel, editors, *EEC-Crest Course on Programming Foundations*, pages 57–114. IRIA, 1978. Also printed as Technical Report 33 of Institute of Informatics, University of Oslo.

[Dar72] J. Darlington. *A Semantic Approach to Automatic Program Improvement.* PhD thesis, University of Edinburgh, 1972.

[Dav65] M. Davis. *The Undecidable*. Raven Press, 1965.

[dB68] J. W. de Bakker. Axiomatics of simple assignment statements. Technical Report 94, Mathematisch Centrum, Amsterdam, June 1968.

[dB69] J. W. de Bakker. Semantics of programming languages. In J. T. Tou, editor, *Advances in Information Systems Science*, volume 2, pages 173–227. Plenum Press, 1969.

[dB70] N. G. de Bruijn. The mathematical language AUTOMATH – its usage and some of its extensions. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.

[DB76] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[dB80] J. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall International, 1980.

[dBS69] J. W. de Bakker and D. Scott. A theory of programs. Manuscript notes for IBM Seminar, Vienna, August 1969.

[DDH72] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press, 1972.

[Dij68a] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, 1968.

[Dij68b] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[Dij68c] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[Dij71] E. W. Dijkstra. A short introduction to the art of programming. Technisch Hogeschool Eindhoven, EWD-316, 1971.

[Dij72] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15:859–866, 1972.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DLP79] R. A. DeMillo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22:271–280, May 1979.

[DMN68] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. SIMULA 67 common base language. Technical Report S-2, Norwegian Computing Center, Oslo, 1968.

[Don75] J. E. Donahue. Complementary definitions of programming language semantics. Technical Report CSRG-62, University of Toronto, Canada, November 1975.

[dR74] W.-P. de Roever. Recursion and parameter mechanisms: An axiomatic approach. In J. Loeckx, editor, *Automata Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*. Springer-Verlag, 1974.

[dR76a] W.-P. de Roever. Call-by-value versus call-by-name: A proof theoretic comparison. Technical Report IW 23/76, Mathematical Center, Amsterdam, September 1976.

[dR76b] W.-P. de Roever. Dijkstra's predicate transformer, nondeterminism, recursion and termination. Technical Report 37, I.R.I.S.A., University of Rennes, 1976.

[dR85] 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. In *[NC85]*, pages 181–205, 1985.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1990.

[Eng71] E. Engeler. *Symposium on Semantics of Algorithmic Languages*. Number 188 in Lecture Notes in Mathematics. Springer-Verlag, 1971.

[Ers71] A. P. Ershov. Theory of program schemata. In C. V. Freiman, editor, *Information Processing 71*, volume 1, pages 28–46. North-Holland, 1971. Proceedings of IFIP'71.

[Ers90] A. P. Ershov. *Origins of Programming: Discourses on Methodology*. Springer-Verlag, 1990. Original Russian in 1977.

[Eva65] A. Evans. *Syntax Analysis by a Production Language*. PhD thesis, Carnegie Institute of Technology, 1965.

[Fet88] J. H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31:1048–1063, 1988.

[FH71] M. Foley and C. A. R. Hoare. Proof of a recursive program: Quicksort. *BCS, Computer Journal*, 14:391–395, November 1971.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[FP78] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978.

[GCH+78] D. I. Good, R. M. Cohen, C. G. Hoch, L. W. Hunter, and D. F. Hare. Report on the language Gypsy, version 2.0. Technical Report ICSCA-CMP-10, University of Texas at Austin, September 1978.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, DEC, SRC, July 1985.

[GMW79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1979.

[Goo70] D. I. Good. *Toward a Realization of a Program Proving System.* PhD thesis, University of Wisconsin, 1970.

[Gor59] S. Gorn. Common programming language task: Final report No. AD59UR1. 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.

[Gor61] S. Gorn. Specification languages for mechanical languages and their processors – a baker's dozen. *Communications of the ACM*, 4:532–542, 1961.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Gri78] D. Gries, editor. *Programming Methodology: A Collection of Articles by Members of IFIP W.G. 2.3.* Springer-Verlag, 1978.

[Gri81] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[GTWW75] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics. Technical Report RC 5243, IBM Yorktown Heights, January 30th 1975.

[GTWW77] J. A. Goguen, J. W. Thatcher, E. W. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24:68–95, 1977.

[Gut75] J. V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* PhD thesis, University of Toronto, Computer Systems Research Group, September 1975. CSRG-59.

[GvN47] 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 U.S. Army Ord. Dept.; also published as pages 80–151 of [Tau63].

[GY91] D. I. Good and W. D. Young. Mathematical methods for digital systems development. In *[PT91b]*, pages 406–430, 1991.

[Hay86] I. Hayes, editor. *Specification Case Studies*. Prentice-Hall International, 1986.

[HdR86] J. Hooman and W.-P. de Roever. The quest goes on: A survey of proof systems for partial correctness of CSP. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[Heh84] E. C. R. Hehner. *The Logic of Programming*. Prentice-Hall International, 1984.

[HES72] T. E. Hull, W. H. Enright, and A. E. Sedgwick. The correctness of numerical algorithms. *ACM SIGPLAN Notices*, 7(1):66–73, January 1972.

[HHH+87] C. A. R. Hoare, I. J. Hayes, J. He, C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. The laws of programming. *Communications of the ACM*, 30:672–687, 1987. see Corrigenda in *ibid* 30:770.

[HHS86] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined: Resumé. In B. Robinet and R. Wilhelm, editors, *ESOP'86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.

[HJ89] C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science*. Prentice Hall International, 1989.

[HL74] C. A. R. Hoare and P. E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3:135–153, 1974.

[Hoa61] C. A. R. Hoare. Algorithm 63, Partition; Algorithm 64, Quicksort; Algorithm 65, Find. *Communications of the ACM*, 4(7):321–322, July 1961.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 583, October 1969.

[Hoa71a] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *[Eng71]*, pages 102–116. 1971.

[Hoa71b] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14:39–45, January 1971.

[Hoa72a] C. A. R. Hoare. Proof of a structured program: 'the sieve of Eratosthenes'. *Computer Journal*, 15:321–325, November 1972.

[Hoa72b] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hoa72c] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. Perrot, editors, *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17:549–557, October 1974.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hod83] A. Hodges. *Alan Turing: The Enigma*. Burnett Books, 1983. Vintage edition, 1992.

[Hoo87] A. Hoogewijs. Partial-predicate logic in computer science. *Acta Informatica*, 24:381–393, 1987.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[Iga64] S. Igarashi. *An Axiomatic Approach to the Equivalence Problems of Algorithms with Applications*. PhD thesis, University of Tokyo, 1964. Reprinted as Report of the Computer Centre University of Tokyo, No. 1, in 1968.

[INM88] INMOS. *occam 2: Reference Manual*. Prentice Hall, 1988.

[Jon70] C. B. Jones. A technique for showing that two functions preserve a relation between their domains. Technical Report LR 25.3.067, IBM Laboratory, Vienna, April 1970.

[Jon72] C. B. Jones. Formal development of correct algorithms: an example based on Earley's recogniser. *ACM SIGPLAN Notices*, 7(1):150–169, January 1972.

[Jon73] C. B. Jones. Formal development of programs. Technical Report 12.117, IBM Laboratory Hursley, April 1973.

[Jon77] C. B. Jones. Implementation bias in constructive specification of abstract objects. unpublished manuscript, September 1977.

[Jon79] C. B. Jones. Constructing a theory of a data structure as an aid to program development. *Acta Informatica*, 11:119–137, 1979.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as Technical Monograph No. PRG-25.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *Information Processing'83*, pages 321–332. North-Holland, 1983.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, 1986.

[Jon87] C. B. Jones. VDM proof obligations and their justification. In *[BJMN87]*, pages 260–286. Springer-Verlag, 1987.

[Jon89] C. B. Jones. Data reification. In J. A. McDermid, editor, *The Theory and Practice of Refinement*, pages 79–89. Butterworths, 1989.

[Kah79] G. Kahn, editor. *Semantics of Concurrent Computation: Proceedings, Evian, France 1979*, volume 70 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1979.

[Kem86] R. A. Kemmerer. Verification assessment study: Final report, volume 1 overview, conclusions and future directions. Technical Report C3-CR01-86, Library No. S-228,204, National Computer Security Center, Maryland, USA, March 1986.

[Kin69] J. C. King. *A Program Verifier*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1969.

[KM77] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing'77*, pages 993–998, 1977.

[Kow79] R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424–436, 1979.

[KTB88] B. Konikowska, A. Tarlecki, and A. Blikle. A three-valued logic for software specification and validation *tertium tamen datur*. In *[BMJ88]*, pages 218–242, 1988.

[Lam77] L. Lamport. Proving the correctness of mutiprocess programs. *IEEE Transactions on Software Engineering*, 3:125–143, 1977.

[Lam80] L. Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

[Lam88] L. Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, 10:267–281, April 1988.

[Lam90] L. Lamport. A temporal logic of actions. Technical Report 57, DEC, SRC, 1990.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Lan65] P. J. Landin. A correspondence between ALGOL-60 and Church's lambda-notation. Parts I and II. *Communications of the ACM*, 8:89–101, 158–165, 1965.

[Lau71] P. E. Lauer. *Consistent Formal Theories of the Semantics of Programming Languages*. PhD thesis, Queen's University of Belfast, 1971. Printed as TR 25.121, IBM Lab. Vienna.

[Lau72] H. C. Lauer. *Correctness in Operating Systems.* PhD thesis, Carnegie-Mellon University, 1972.

[LGH+78] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules for the programming language Euclid. *Acta Informatica*, 10:1–26, 1978.

[Lin88] P. A. Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal*, 3:3–27, January 1988.

[Lon64] R. L. London. *A Computer Program for Discovering and Proving Sequential Recognition Rules for Well-formed Formulas Defined by a Backus Normal Form Grammar.* PhD thesis, Carnegie Institute of Technology, 1964.

[Lon70a] R. L. London. A correctness proof of the Fisher-Galler algorithm using inductive assertions. Technical Report 102, The University of Wisconsin, Computer Sciences Dept, October 1970.

[Lon70b] R. L. London. Experience with inductive assertions for proving programs correct. Technical Report 92, The University of Wisconsin, Computer Sciences Dept, May 1970.

[Lon70c] R. L. London. Proof of algorithms – a new kind of certification. *Communications of the ACM*, 13(6):371–373, 1970.

[Lon70d] R. L. London. Proving programs correct: Some techniques and examples. *BIT*, 10:168–182, 1970.

[Lon71] R. L. London. Correctness of two compilers for a Lisp subset. Technical Report CS240, Computer Science Dept, Stanford University, October 1971.

[LPP70] D. C. Luckham, D. M. R. Park, and M. S. Paterson. On formalised computer programs. *Journal of Computer and System Sciences*, 4:220–249, 1970.

[Luc67] D. Luckham. The resolution principal in theorem proving. In N. L. Collins and D. Michie, editors, *Machine Intelligence, 1*, pages 47–61. Olliver and Boyd, 1967.

[Luc68] P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory Vienna, June 1968.

[Luc72] P. Lucas. On the semantics of programming languages and software devices. In *[Rus72]*, pages 41–57. 1972.

[Luc81] P. Lucas. Formal semantics of programming languages: VDL. *IBM Journal of Research and Development*, 25(5):549–561, September 1981.

[Luc87] P. Lucas. VDM: Origins, Hopes, and Achievements. In *[BJMN87]*, pages 1–18, 1987.

[Łuk20] J. Łukasiewicz. O logice trójwartościowej (on three-valued logic). *Ruch Filozoficzny*, 5:169–171, 1920.

[LW69] P. Lucas and K. Walk. *On The Formal Description of PL/I*, volume 6, Part 3 of *Annual Review in Automatic Programming*. Pergamon Press, 1969.

[Mac79] D. B. MacQueen. Models for distributed computing. Technical Report 351, INRIA, France, April 1979.

[Man68] Z. Manna. *Termination of Algorithms*. PhD thesis, Carnegie-Mellon University, April 1968.

[Man74] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[MC81] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.

[McC59] J. McCarthy. Programs with common sense. In *Teddington Conference on the Mechanization of Thought Processes*, 1959.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3:184–195, April 1960.

[McC63a] J. McCarthy. A basis for a mathematical theory for computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland Publishing Company, 1963. (A slightly extended and corrected version of a talk given at the May 1961 Western Joint Computer Conference).

[McC63b] J. McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, editor, *Information Processing'62*, pages 21–28. North-Holland, 1963.

[McC66] J. McCarthy. A formal description of a subset of ALGOL. In *[Ste66]*, pages 1–12, 1966.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[Mil69] R. Milner. The difficulty of verifying a program with unnatural data representation. Technical Report 3, Computation Services Dept., University College of Swansea, January 1969.

[Mil70] R. Milner. A formal notion of simulation between programs. Technical Report 14, Department of Computer Science, University College of Swansea, October 1970.

[Mil71a] R. Milner. An algebraic definition of simulation between programs. Technical Report CS-205, Computer Science Dept, Stanford University, February 1971.

[Mil71b] R. Milner. Program simulation: An extended formal notion. Technical Report 17, University College of Swansea, April 1971.

[Mil72] R. Milner. Logic for computable functions description of a machine implementation. Technical Report STAN-CS-72-288, Computer Science Department, Stanford University, May 1972.

[Mil80] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MJ84] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6:139–143, April 1984.

[MM69] Z. Manna and J. McCarthy. Properties of programs and partial function logic. In B. Meltzer and D. Michie, editors, *Machine Intelligence, 5*, pages 27–37. Edinburgh University Press, 1969.

[Möl91] B. Möller. Formal derivation of pointer algorithms. In M. Broy, editor, *Informatik und Mathematik*, pages 419–440. Springer-Verlag, 1991.

[Mor73a] F. L. Morris. Advice on stucturing compilers and proving them correct. In *ACM Symposium on Principles of Programming Languages*, pages 144–152. ACM, 1973.

[Mor73b] J. H. Morris. Types are not sets. In *ACM Symposium on Principles of Programming Languages*, pages 120–124. ACM, October 1973.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[MP66] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. Technical Report CS38, Computer Science Department, Stanford University, April 1966. See also pages 33–41 Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science, American Mathematical Society, 1967.

[Nau66] P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[Nau69] P. Naur. Programming by action clusters. *BIT*, 9:250–258, 1969.

[Nau74] P. Naur. *Concise Survey of Computer Methods*. Studentlitteratur, Lund, 1974.

[Nau82] P. Naur. Formalization in program development. *BIT*, 22:437–453, 1982.

[NC85] E. J. Neuhold and G. Chroust. *Formal Models in Programming*. North-Holland, 1985. Proceedings of the IFIP TC2 Working Conference on The Role of Abstract Models in Information Processing. Vienna, Austria, 30 January – 1 February 1985.

[Nip86a] T. Nipkow. *Behavioural Implementation Concepts for Nondeterministic Data Types*. PhD thesis, University of Manchester, 1986. Reprinted as UMCS-87-5-3, May 1987.

[Nip86b] T. Nipkow. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22:629–661, 1986.

[NS56] A. Newell and H. A. Simon. The logic theory machine. In *IRE Transactions on Information Theory IT-2*, pages 61–79, 1956.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[Owi75] S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975. Published as technical report 75-251.

[Par69] D. Park. Fixpoint induction and proofs of program properties. In B. Meltzer and D. Michie, editors, *Machine Intelligence, 5*, pages 59–78. Edinburgh University Press, 1969.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, March, 1981*, number 104 in Lecture Notes in Computer Science, pages 167–183. Springer-Verlag, 1981.

[Par90] H. A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer-Verlag, 1990.

[Pat67] M. S. Paterson. *Equivalence Problems in a Model of Computation*. PhD thesis, University of Cambridge, 1967.

[Pau87] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Darmstadt, 1962.

[Pet76] C. A. Petri. Nichtsequentielle prozesse. Technical Report ISF-76-6, GMD, Bonn, 1976.

[Pet77] C. A. Petri. Non-sequential processes. Technical Report ISF-77-05, GMD, Bonn, 1977. Translation of ISF-76-6.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[Pnu77] A. Pnueli. The temporal semantics of concurrent programs, 1977. Tel-Aviv University.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[PT91a] S. Prehn and W. J. Toetenel, editors. *VDM'91 – Formal Software Development Methods. Vol.1: Conference Contributions*, volume 551 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[PT91b] S. Prehn and W. J. Toetenel, editors. *VDM'91 – Formal Software Development Methods. Vol.2: Tutorials*, volume 552 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Ran75] B. Randell. *The Origins of Digital Computers: Selected Papers*. Springer-Verlag, second edition, 1975.

[Rey81] J. C. Reynolds. *The Craft of Programming*. Prentice Hall International, 1981.

[RH86] A. W. Roscoe and C. A. R. Hoare. Laws of occam programming. Technical Report PRG-53, Oxford University Computing Laboratory, Programming Research Group, 1986.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Rob92] J. A. Robinson. Logic and logic programming. *Communications of the ACM*, 35(3):40–65, March 1992.

[Roz85] G. Rozenberg. *Advances in Petri-nets*, volume 188 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[Rus72] R. Rustin. *Formal Semantics of Programming Languages*. Prentice-Hall, 1972. Courant Computer Science Symposium 2, September 14-16, 1970.

[Rut64] J. D. Rutledge. On Ianov's program schemata. *Journal of the ACM*, 11:1–9, January 1964.

[SA89] D. M. Steier and A. P. Anderson. *Algorithm Synthesis: A Comparative Study*. Springer-Verlag, 1989.

[Sco67] D. Scott. Existence and description in formal logic. In R. Schoenman, editor, *Bertrand Russell, Philosopher of the Century*, pages 181–200. Allen and Unwin, 1967.

[Sco69a] D. Scott. A construction of a model for the λ calculus. Manuscript, November 1969.

[Sco69b] D. Scott. Models for the λ calculus. Manuscript – Draft, December 1969.

[Sco70] D. Scott. Outline of a mathematical theory of computation. Technical Report PRG-2, Oxford University Computing Laboratory, Programming Research Group, November 1970.

[Sco73] D. Scott. A simplified construction for λ calculus models. Manuscript, April 1973.

[Sco76] D. Scott. Data types as lattices. Technical Report PRG-5, Oxford University Programming Research Group, September 1976. Reprinted from the SIAM Journal on Computing, Volume 5, 1976, pp. 522–587; manuscript version dated 1972.

[Spi88] J. M. Spivey. *Understanding Z – A Specification Language and its Formal Semantics.* Cambridge Tracts in Computer Science 3. Cambridge University Press, 1988.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, 1989.

[SS70] C. Strachey and D. Scott. Mathematical semantics for two simple languages, August 1970. Paper read at Princeton.

[Ste66] T. B. Steel. *Formal Language Description Languages for Computer Programming.* North-Holland, 1966.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures.* PhD thesis, Manchester University, 1990. Also published as technical report UMCS-91-1-1.

[Str66] C. Strachey. Systems analysis and programming. *Scientific American*, 215(3):112–124, September 1966.

[Tar85] A. Tarlecki. A language of specified programs. *Science of Computer Programming*, 5:59–81, 1985.

[Tau63] A. H. Taub, editor. *John von Neumann: Collected Works*, volume V: Design of Computers, Theory of Automata and Numerical Analysis. Pergamon Press, 1963.

[Tur36] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, 1936. Correction published: ibid, 43:544–546, 1937.

[Tur49] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, Cambridge, June 1949.

[vW66] A. van Wijngaarden. Numerical analysis as an independent science. *BIT*, 6:66–81, 1966. (Text of 1964 talk).

[Wan60] H. Wang. Towards mechanical mathematics. *IBM Journal of Research and Development*, 4:2–22, 1960.

[Wex81] R. L. Wexelblat, editor. *History of Programming Languages.* Academic Press, 1981.

[Wil85] M. V. Wilkes. *Memoirs of a Computer Pioneer.* MIT Press, 1985.

[Wir73] N. Wirth. *Systematic Programming: An Introduction.* Prentice-Hall, 1973.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.

[Yan58] Y I. Yanov. On logical algorithm schemata. *Problems of Cybernetics*, 1:75–127, 1958.

[Zem66] H. Zemanek. Semiotics and programming languages. *Communications of the ACM*, 9:139–143, 1966.

[Zem74] H. Zemanek. Formalization, history, present and future, 1974. Paper for Newcastle IBM Seminar.

[Zil74] S. N. Zilles. Abstract specifications for data types. Technical Report 11, M.I.T. Progress Report, 1974.

[Zus84] K. Zuse. *Der Computer: Mein Lebenswerk*. Springer-Verlag, 1984.

[Zwi88] J. Zwiers. *Compositionality, Concurrency and Partial Correctness: Proof theories for networks of processes, and their relationship*. PhD thesis, Technical University Eindhoven, 1988. Available as LNCS 321, Springer-Verlag.