Prepared for the book, "Symposium on the Semantics of Algorithmic Languages," Erwin Engeler (Ed.), Springer Verlag Lecture Notes Series

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

EXPERIENCE WITH INDUCTIVE ASSERTIONS FOR PROVING PROGRAMS CORRECT

by

Ralph L. London

Technical Report #92

May 1970

EXPERIENCE WITH INDUCTIVE ASSERTIONS FOR PROVING PROGRAMS CORRECT

by

Ralph L. London

Introduction

A most effective method for proving that many computer programs are correct is the inductive assertion method. It is described with examples in Floyd (1967), King (1969), Knuth (1968, pp. 14-20) and Naur (1966); further examples of its use may be found in Good and London (1970), London (1970 c,d) and London and Halton (1969). The basic idea of the method is that one makes assertions concerning the progress of the computation at certain points in the code. The proof consists of verifying that each assertion is true each time control reaches that assertion, under the assumption that the previously encountered assertions are true.

London (1970 b,d) discusses the advantages and feasibility of proving programs. The present paper puts forth a notation or framework for stating assertions and gives some suggestions for its effective use, including in verifying the assertions. Two running computer systems for producing correctness proofs are summarized. Previously proved programs are analyzed to gain further insight into proofs using inductive assertions. The emphasis is on obtaining practical results.

A Notation for Assertions

Two related abilities are needed in stating assertions. First, one needs simple formulas relating variables. For example, the crucial assertions needed to prove the Extended Euclidean algorithm in Knuth (1968, pp. 14-16) are

$$am + bn = d \tag{1}$$

and

$$a'm + b'n = d \tag{2}$$

while the key to proving a subtle exponential routine in King (1969, pp. 183-189) is the assertion

$$ZX^{Y} = A^{B}. \tag{3}$$

Second, one needs the ability to state the partial computation that has been accomplished or what is true over a range of values. The key assertions in proving a program in Floyd (1967, p. 20) which sums the first N elements of the array A are, using the summation operator Σ,

$$S = \sum_{j=1}^{I} A[j] \tag{4}$$

and the closely related

$$S = \sum_{j=1}^{I-1} A[j] . \tag{5}$$

For a program to sort an array M of N elements into ascending order, the proof in London (1970c) uses the important assertion

$$M[p] \leq M[p+1] \quad \text{for} \quad I+1 \leq p \leq N-1 \tag{6}$$

or, using the and operator $\wedge$,

$$\bigwedge_{p=I+1}^{N-1} M[p] \leq M[p+1]. \tag{7}$$

The second type of assertions may also be expressed using three dots ($p=I+1,\ldots,N-1$), using quantifiers ($\forall p(I+1 \leq p \leq N-1)$) or using other convenient symbolism.

All the above assertions, except (6), are members of the class of assertions defined by the pattern

$$\text{expression} \quad \text{relation} \quad \text{operator} \quad \text{expression} \qquad (8)$$

$$\text{operator} \in \left\{ \begin{array}{l} \text{bounds} \\ \text{or other} \\ \text{modifications} \end{array} \right\}$$

where expression is arithmetic or Boolean; relation is $=, >, <, \geq, \leq, \neq, \in, \notin$; and operator is $\Sigma, \Pi, \wedge, \vee$, max, min. (Certain of the elements of (8) may be missing. For example, "operator" is missing in (1) and both the first "expression" and "relation" are missing in (7).)

The class of assertions may be increased by using the same pattern but allowing additional kinds of expressions, relations and operators that are appropriate to the particular program being proved. Furthermore, a single assertion may be composed of several assertions, each in the form of (8), joined by the four logical connectives: and, or, not, imply.

Using the Notation to State Assertions

The pattern (8) is intended as an informal statement of the type of assertions that are needed and are useful. Exactly what is needed in a particular example will, of course, depend on that example. However, the idea is to assert, in sufficiently general terms using the program variables (especially the loop control variables), what holds invariantly. The idea may also be expressed as the need to write an induction hypothesis as if a proof were to be done by ordinary mathematical induction. More concretely, what is required is an assertion describing the incremental processing accomplished by the ith execution of a loop, or more generally, the

total processing accomplished by the first through the ith execution, as in (4) and (7) or even (3).

Certain necessary assertions may have to be more complex than the ones giving the final result. For example, compare (3) with the result $Z = A^B$. Assertions at an innermost loop, say, are usually more involved than the final ones.

It is difficult to be more explicit. King (1969, p. 113) states, "...the creation of predicates [assertions] remains an art learned by experience." Experience, of course, includes the study of existing proofs as well as giving proofs oneself. Naur (1966, p. 313), calling assertions by the descriptive term General Snapshots, advises, "...the values of variables given in a General Snapshot normally at best can be expressed as general, mathematical expressions or by equivalent formulations. I have to say 'at best' because in many cases we can only give certain limits on the value, and I have to admit 'equivalent formulations' because we do not always have suitable mathematical notation available."

That the notation (8) for assertions is useful in practice is seen by considering the assertions in successful proofs. The last section of this paper profiles ten inductive assertion proofs. All of the assertions in these proofs can be stated in the form of (8). While some of the assertions as originally written do not fit (8), each can be rewritten in a natural way to conform to (8).

In addition, all assertions in the twelve examples in King (1969, Appendices II and III) fit into (8) provided the quantifier notation is replaced as in (7). King's assertions are written as what he calls super Boolean expressions, a notion he precisely defines in BNF notation. They are essentially ordinary Algol Boolean

expressions augmented by $\forall$ and $\exists$ to bind simple integer variables. Super Boolean expressions cover much but not all of (8).

In writing assertions, a convenient device is to give a name to a complicated assertion and to include provision for a variable "parameter." One then uses this name instead of the actual assertion. For example, in London (1970 c) the assertion

$$\bigwedge_{k=2}^{n} M[k \div 2] \geq M[k] \qquad (9)$$

was needed. This set of inequalities was named A(s), where s is effectively a formal parameter. The actual assertions made were then $A(i_o)$, A(i), $A(i_o+1)$, A(i+1), A(n:2+1) and A(2). This device is also used extensively in London and Halton (1969) for simplified reference to rather complicated assertions.

Another device, that of introducing new names, arises because the program variables (or names) appearing in the assertions always refer to the current values of the variables at the point in the program where the assertion is made. The original starting values may be lost so sometimes there is need for additional variables to complete the proof. Thus, in the exponential routine that computes Z as A^B, the given inputs A and B are renamed X and Y; it is X and Y (but not A and B) that are altered by the program in the course of the computation. (Cf. an Algol procedure with parameters called by value.) The point is that both the original values A and B and the current values X and Y are needed in the proof as in assertions (3).

It may be necessary to create new names if the program does not do so explicitly. One convenient way is to use a subscript zero on the variable to denote the original value. In London (1970 c) this was done for a simple variable and also for the

array name M. The latter allowed the assertion that the current array M is a permutation of the original array M_o (thus showing that no array elements were lost in the sorting).

The names may be created by a suitable assertion, for example,

$$X = A \quad \text{and} \quad Y = B. \tag{10}$$

In London and Halton (1969) the proof of Algorithm (91) includes an assertion in a loop which creates the needed names by using the current value of the loop control variable as a subscript.

Unfortunately, one needs assertions beyond the key or crucial assertions (1) - (7) to complete the respective proofs. These additional assertions cover certain necessary details which are nevertheless important and all too easy to overlook. Examples include ensuring that array references are within bounds, that certain variables are always integers and that bounds on variables hold. Each such assertion of detail may well be made at several points in the program and perhaps as a separate assertion from the crucial assertions. (More than one assertion is often profitably made at a point.)

The number of assertions that appear in a proof of a given program will vary with different tastes, styles and techniques of proof. Generally one would expect an assertion at the exit(s) to state the results and an assertion at the entrance(s) giving initial assumptions, if any. Furthermore, each closed path (loop) in the program must contain at least one assertion. The same assertion may serve for more than one loop. Otherwise one is, of course, free to place additional assertions at selected points to aid in verifying conveniently all the assertions. While certain programs can

be verified easily with the minimum number of assertions necessary, this is not recommended general strategy, especially for large programs. The other extreme--with assertions at every possible point--is likewise to be avoided in general.

The usually challenging and creative job of stating assertions does not complete the proof. The significant task of verifying the assertions remains just as an ordinary induction hypothesis must be proved and not merely stated. What must be done is the subject of the next section.

Verifying the Assertions

Assume at least one assertion at each entrance and at each exit--if necessary the trivial assertion TRUE. Several assertions at one point are here considered as one conjoined assertion. Consider all paths of control between assertions that start and end with an assertion, that contain no other assertions and that contain no program statement more than once. There are a finite number of such paths since every loop contains an assertion (and there are only a finite number of assertions and statements). Each path leads to a "verification condition," a conjecture which must be proved to be a theorem. A proven verification condition verifies the assertion at the end of the path but only for that particular path. Note that an assertion is often at the end of more than one path and, accordingly, requires more than one verification. If all verification conditions are proved, then the program is correct. Failure to prove a verification condition does not necessarily mean the program is incorrect since improper assertions may have been chosen.

There are several ways to construct verification conditions. Two of the ways, forward accumulation and backward substitution, are illustrated below by an example

from King (1969, pp. 20ff.) Full details of these and other algorithms are in King (1969) and Good (1970), including the proof that each leads to valid proofs, the equivalences between them and the relative advantages and disadvantages of their use.

Suppose assertion B is to be verified along a path that begins with assertion A. Forward accumulation starts with A, accumulates terms for intermediate statements, changes notation as needed and ends at B. Backward substitution starts with B, makes substitutions directly in B and ends at A.

The construction of verification conditions is shown for the example which computes z as x * y. The assertions are enclosed in braces. Paths are identified by pairs of assertion numbers with the second number denoting the assertion to be verified.

```
        {assertion 1: true}
1.      w ← y ;
2.      z ← 0 ;
    L:  {assertion 2;  z + w  * x = x * y}
3.      if  w = 0  then {assertion 3: z = x * y} return  z;
4.      z ← x + z;
5.      w ← w - 1;
        go to L ;
```

Verification conditions (vc) by forward accumulation:

1-2: vc: true $\wedge$ w = y $\wedge$ z = 0 $\supset$ z + w * x = x * y

2-3: vc: z + w * x = x * y $\wedge$ w = 0 $\supset$ z = x * y

2-2: derived in stages (sample notation: "z" is current value, "z1" is old value)

after 3: z + w * x = x * y $\wedge$ w $\neq$ 0

after 4: z1 + w * x = x * y $\wedge$ w $\neq$ 0 $\wedge$ z = x + z1

after 5: z1 + w1 * x = x * y $\wedge$ w1 $\neq$ 0 $\wedge$ z = x + z1

$\wedge$ w = w1 - 1

vc: z1 + w1 * x = x * y $\wedge$ w1 $\neq$ 0 $\wedge$ z = x + z1

$\wedge$ w = w1 - 1 $\supset$ z + w * x = x * y

Verification conditions by backward substitution:

1-2: start with assertion 2: z + w * x = x * y

after 2: 0 + w * x = x * y

vc: true $\supset$ y * x = x * y

2-3: start with assertion 3: z = x * y

after 3: w = 0 $\supset$ z = x * y

vc: z + w * y = x * y $\supset$ (w = 0 $\supset$ z = x * y)

2-2: start with assertion 2: z + w * x = x * y

after 5: z + (w-1) * x = x * y

after 4: x + z + (w-1) * x = x * y

after 3: w $\neq$ 0 $\supset$ x + z + (w-1) * x = x * y

vc: z + w * x = x * y $\supset$

(w $\neq$ 0 $\supset$ x + z + (w-1) * x = x * y)

(All the verification conditions can be shown to be theorems. Having done so, it has been proved that if assertion 3 is reached, then $z = x * y$ as required. To show that assertion 3 is reached, i.e. the program terminates, requires the additional assumption $y \geq 0$. Termination is usually shown separately although inductive assertions can be stated and verified for termination as well.)

From this example it can be seen that for a given program, the number and complexity of the verification conditions (and the ease of their proofs) are determined by the assertions and the interconnection of statements between assertions. More points with assertions will usually give simpler verification conditions but more of them. For example, the number of verification conditions might be increased, but each one made simpler, by stating assertions at a point between two other assertions. Indeed, there may be no loss. King (1969, p. 24) gives an extreme example of a loop-free program where adding assertions decreases the number of verification conditions.

To construct and to prove the verification conditions involves several types of information: the semantic properties of the programming language in which the program is written; the properties of the expressions, relations and operators of the assertions; and the semantics of the problem domain. A human prover can usually, without undue difficulty, select from this information and organize it into verifications of the assertions. Although he often does not explicitly construct verification conditions, this concept, plus an accurate statement of all the axioms and properties used, would serve to formalize his verifications.

In his verifications he does not state the use of every axiom and elementary theorem of arithmetic, or those of symbol manipulation more generally; neither does he state

properties of the assertions and the problem domain. To do so would render already tedious proofs essentially impossible. The result is a rigorous but not formal proof and therefore a proof not subject to machine proof-checking.

An alternative to human-constructed proofs is, of course, to build an automatic program verification system. One such effort is described in the next section and is compared with a partially automatic system.

Automatic proof construction

The separate approaches of King (1969) and Good (1970) demonstrate that the computer itself can be of significant help in proving programs correct. King's prototype Program Verifier, running on an IBM 360 Model 65, uses the inductive assertion method on programs written in a simple Algol-like programming language, restricted to integers, but including one-dimensional arrays. The assertions are all written by a human as super Boolean expressions, noted earlier. Powerful formula simplification and manipulation routines are used to construct the (backward) verification conditions and, using an automatic theorem prover, to prove them. Specialized techniques for integers are also used.

Good's system, running on a Burroughs B5500, requires man-machine cooperation to complete a proof. It uses the inductive assertion method on a programming language similar to King's but without arrays or procedures and with declarations. The human uses a teletype to make all the assertions which can be any text string. Verification conditions are produced by the system; it looks for instances of program variable names in an assertion and substitutes appropriately. The human supplies the proof of the verification conditions, again as a text string, which the system accepts

unquestioningly. The system provides significant additional help to the human by keeping track of which assertions remain to be verified along which paths, by allowing proofs to be changed, by retrieving assertions or proofs, by giving the complete or partial proof at the end, etc.

King's verifier has proved some interesting but small programs. The class of programs to which the verifier is applicable is also restricted. Some needed improvements he himself cites are increasing the expressive power of the assertions, keeping the expressions generated of reasonable size, increasing the power of the simplification process, adding more sophisticated features to the programming language and providing a genuine open-ended theorem prover. King defends his claim (p. 115), "we see no major hurdle which prevents the system from being continually upgraded by removing the limitations." Doing so will be far from trivial but a good foundation does exist.

Good's system was tried on only a few small problems. The assertions given as text strings worked surprisingly well. They generated useable, intelligible, and easily provable verification conditions. This seemingly absurd way to state assertions shows real promise.

In writing a proof, however, there are problems for the human in identifying and referencing assertions, parts of the program, parts of the already completed proof and other objects, all of which he needs. These are caused partly by the fact that while the human would like to view his program as a series of linear statements, the system forces him to view it as a graph which it created from the original program. The similar graph in King's verifier causes no problem because no interaction is involved.

With some effort by the human, then, proofs can be completed. But, as expected, human-factors problems were uncovered; some suggested improvements remain to be implemented and tested. There are also problems remaining in allowing the human to modify his assertions or, even worse, to modify his program. The difficulty is to allow this without completely invalidating the partially completed proof. In short, Good has made a significant start in the realization of a man-machine program proving system as well as identifying some of the remaining problems.

Both King (1969) and Good (1970) contain numerous additional insights and suggestions for understanding and using the inductive assertion method. In particular Good has provided a formal basis for further consideration of the correctness problem that unifies some of the results obtained by others.

It is important to note that King and Good agree on the value of both approaches. King, in referring to the manual proof of Good and London (1968) says (p. 166), "Computer assistance to this tedious work appears to be as necessary for them [the authors] as human assistance is for our theorem prover. Some workable compromise between the efforts of man and machine seems to offer the most hope." Good (p. 164) suggests "combining the algebraic simplification and automatic theorem prover approach of King with the man-machine interactive approach described here."

The proposed assertion notation (8), when sufficiently formalized, may be suitable for use with mechanical theorem provers. Properties of the assertions, stated as axioms, would be used along with, say, the axioms of arithmetic and the semantics of the programming language in generating and proving the verification conditions. Both Rutledge (undated, pp. 13-14) and King (1969) suggest the need and desirability of so doing.

To illustrate, consider the simple example of finding the largest element L in the array A[1:n] where $1 \leq N$, an integer. This is a slight modification of an example given by Naur (1966).

{assertion 0: true}

1. $L \leftarrow A[1]$;

2. $I \leftarrow 2$;

B: {assertion 1: $L = \max_{j=1}^{I-1} A[j]$}

3. if $I > N$ then {assertion 3: $L = \max_{j=1}^{N} A[j]$}

return L;

4. if $A[I] > L$ then $L \leftarrow A[I]$;

{assertion 2: $L = \max_{j=1}^{I} A[j]$}

5. $I \leftarrow I + 1$;

6. go to B;

There are two properties or axioms of the max operator which are needed:

$$\max_{j=1}^{1} A[j] = A[1] \tag{11}$$

and

$$L = \max_{j=1}^{I-1} A[j] \wedge A[I] > L \supset A[I] = \max_{j=1}^{I} A[I]$$

$$\vee L = \max_{j=1}^{I-1} \wedge A[I] \leq L \supset L = \max_{j=1}^{I} A[j]. \tag{12}$$

(12) gives the two cases of the ith element being the new maximum or not. (12) would justify the induction step of an ordinary induction and (11) the basis step. Either formally or informally, assertion 1 can be verified on path 0-1 by (11) and on path 2-1 using statement 5. Assertion 2 can be verified on both paths 1-2 by (12), using one case on each path. Assertion 3 can be verified on path 1-3 by assertions not shown involving bounds on the integers I and N.

Naur's proof, not surprisingly, involves essentially the same assertions and verifications as given above. However, in place of the max operator he uses a verbal definition. In addition, his verifications use the facts expressed by (11) and (12), but in words--perfectly appropriate for his informal but nevertheless rigorous presentation.

The ability to introduce needed additional notation into assertions and to give axioms describing properties of the notation seems necessary and appropriate in proving by automatic means a wider class of programs. Many useful operators have properties analogous to (11) and (12) as basis and induction steps although not all properties are conveniently expressed in this format. Yet inductive definitions should work well with inductive assertions. Indeed, the real issue is not desirability, but feasibility. This remains an open question.

Profiles of Specific Inductive Assertion Proofs

Another way to understand how proofs are constructed by the inductive assertion method is to study proofs given by the method. Some reasonably representative inductive assertion proofs (constructed by hand) are listed in Table 1 together with some descriptors of the proofs. The result is a profile of the proofs. The entries are

only approximate since, in compiling this table, some rather arbitrary decisions had to be made. Nevertheless, the data are of interest.

All of the proofs in Table 1 work forward to verify the assertions. Verification conditions are not formally generated although, of course, the verifications could be recast into this form, often straightforwardly and directly.

Definitions of terms in Table 1:

1. Key assertions: Those assertions which express the induction hypothesis or the critical relationship needed. Assertions of detail are excluded.

2. Assumptions: A verification by initial assumption.

3. Recopy unchanged: A verification consisting of quoting a previous assertion without alteration.

4. Non-trivial calculation: A verification requiring some manipulation of assertions and statements.

5. Program logic: A verification which uses only the logic or semantics of the programming language on previous assertions.

6. Problem features: A verification which uses the features, semantics or interpretation of the problem domain.

Terms 2, 3 and 4 provide one classification of the total verifications; terms 5 and 6 are another classification.

The proofs, identified in Table 1 by letters, are as follows:

A, B and C: Variations of the exponential example of King (1969, pp. 183-189). A is his code and his assertions, B has an additional assertion and C contains one unnecessary code statement. The proofs were written by London.

D and E: TREESORT 3 is the algorithm certified correct in London (1970c). D proves the procedure siftup. E proves the body of the algorithm in which siftup is called.

F, G and H: Three of the six algorithms proved in London and Halton (1969). F is algorithm (97), G is (91) and H is (100).

I: The interval arithmetic code in Good and London (1970), an excerpt typical of Good and London (1968).

J: OUTSIDEACE, Example 3 of London (1970d).

The A and B proofs are quite similar. The extra verifications caused by the added assertion do not alter the profiles significantly and, indeed, the two proofs are essentially the same. It is strictly an individual preference whether a case analysis is expressed as an assertion, as in B, or whether it is handled in the verifications, as in A. Proof C has more assertions than either A or B, the difference being that the same assertions are repeated. This leads to shorter verifications with twice as much straight recopying but with the same breakdown between program logic and problem features.

The TREESORT proofs D and E each have a large number of assertions since that seemed the most convenient way to ensure an accurate proof. In retrospect some assertions could probably be eliminated although perhaps at some cost in the clarity of the proof. Note in D that half of the verifications are done merely by recopying an assertion. Moreover, a large fraction of the verifications follow from the program logic alone. In E, there are calls on the procedure siftup. This explains the absence of recopying. It also explains the apparent discrepancy between the number of non-trivial calculations on the one hand and the large fraction in program logic since the effect of procedure calls was considered to be program logic.

Proofs F and J are for simple programs each of which could be proved with fewer assertions than shown. Proof J is somewhat more complex than F because J's verifications involve translating the encoding of the problem domain back to the problem domain. This could be changed by using assertions directly involving the encoding. The encoding could then be translated separately once and for all at the end.

Program I contains no loop and is essentially a decision tree. Thus the assertions on the branches are all necessary and are all verified with non-trivial calculations that depend mainly on the problem features. The size of the problem statement and the size of the proof include reformulating the original problem to a form ready for implementation.

Proofs G and H are involved with loops. Program G is a single loop while program H is three simple loops, two of which are within a fourth outer loop. When the loops are separated and assertions made and verified noting this structure, the resulting profiles of G and H are similar to each other and to A and B.

Examples 1 and 2 in London (1970d) can also be done by inductive assertions. Both examples involve a linear series of N tests for bidding bridge hands. The goal is to show that each hand satisfies precisely one test from the series. Let H_i be the set of hands passing the ith test. The subgoal of at most one test can be asserted by

$$\bigwedge_{i \neq j} (H_i \cap H_j = \text{null}), \tag{13}$$

and the subgoal of at least one test by

$$\bigvee_{i=1}^{N} H_i = \text{All hands}. \tag{14}$$

The verification of assertions (13) and (14) is precisely the same case analysis proofs as in the examples. Each proof profile consists of one assertion proved by a non-trivial calculation depending heavily on the problem features--an extreme profile indeed.

The profiles may well vary with different proof styles and techniques. The proofs also probably contain additional information, and there is need for other useful descriptors. These profiles should be of value in assessing the ability to automate program proving.

Conclusion

This paper has been an informal presentation of observations and experiences using inductive assertions to prove programs correct. Practical rather than theoretical results have been stressed. If computer scientists find the suggestions useful and applicable and if provably correct programs result, then the overall aim of the paper will have been attained.

Proof / Descriptor	Exponential			TREESORT 3	
	A	B	C	D(siftup)	E(body)
Problem Statement	1 line	1 line	1 line	$\frac{1}{4}$ page	2 lines
Lines of code	9	9	10	12	10
Flowchart boxes	6	6	7	10	5
Total assertions	4	5	14	34	21
Key assertions	2-50	3-60	6-43	13-38	15-71
Size of proof (pages)	$\frac{1}{2}$	$\frac{1}{2}$	1	$2\frac{2}{3}$	$1\frac{1}{2}$
Total Verifications	7	9	21	43	28
Assumptions	1-14	1-11	1-5	2-5	0-0
Recopy unchanged	1-14	2-22	9-43	22-51	0-0
Non-trivial calculation	5-71	6-67	11-52	19-44	28-100
Program logic	5-71	6-67	14-67	36-84	17-61
Problem features	2-29	3-33	7-33	7-16	11-39

Table 1. Proof Profiles

Note: M-N means M instances and N % of the total.
For statements count as one flowchart box.

Proof / Descriptor	Asymptotic Series			Interval Arithmetic	OUT-SIDE-ACE
	F(97)	G(91)	H(100)	I	J
Problem Statement	1 line	$\frac{1}{3}$ page	$\frac{2}{3}$ page	$2\frac{1}{4}$ pages	$\frac{1}{2}$ page
Lines of code	6	11	37	19	9
Flowchart boxes	5	10	29	17	5
Total assertions	5	10	24	10	6
Key assertions	3-60	6-60	15-63	9-90	5-83
Size of proof (pages)	$\frac{1}{4}$	$\frac{1}{2}$	3	$6\frac{1}{2}$	1
Total Verifications	6	13	34	10	8
Assumptions	1-17	1-8	1-3	0-0	0-0
Recopy unchanged	3-50	3-23	12-35	0-0	1-12
Non-trivial calculation	2-33	9-69	21-62	10-100	7-88
Program logic	4-67	9-69	25-74	3-30	5-63
Problem features	2-33	4-31	9-26	7-70	3-37

Table 1. Proof Profiles (continued)

Acknowledgements

This work is supported by National Science Foundation Grant GJ-583 and the Mathematics Research Center, United States Army under Contract Number DA-31-124-ARO-D-462. Allen Newell suggested to me the idea of a proof profile and some of the descriptors.

Bibliography

Floyd, R. W. (1967), Assigning meanings to programs, Proceedings of a Symposium in Applied Mathematics, Vol. 19-- Mathematical Aspects of Computer Science, Schwartz, J. T. (ed.), American Mathematical Society, Providence, R. I., pp. 19-32.

Good, D. I. (1970), Toward a man-machine system for proving program correctness, Ph.D. Thesis, University of Wisconsin.

Good, D. I. and London, R. L. (1968), Interval arithmetic for the Burroughs B5500: Four Algol procedures and proofs of their correctness, Computer Sciences Technical Report No. 26, University of Wisconsin.

Good, D. I. and London, R. L. (1970), Computer interval arithmetic: Definition and proof of correct implementation, J. ACM, (to appear).

King, J. C. (1969), A program verifier, Ph.D. Thesis, Carnegie-Mellon University.

Knuth, D. E. (1968), The Art of Computer Programming, Vol. 1-- Fundamental Algorithms, Addison-Wesley, Reading, Mass.

London, R. L. (1970a), Bibliography on proving the correctness of computer programs, Machine Intelligence 5, Meltzer, B. and Michie, D. (eds.), Edinburgh University Press, Edinburgh, pp. 569-580.

London, R. L. (1970b), Computer programs can be proved correct, Theoretical Approaches to Non-numerical Problem Solving--Proceedings of Systems Symposium at Case Western Reserve University, Banerji, R. B. and Mesarovic, M. D. (eds.), Springer Verlag, pp. 281-303.

London, R. L. (1970c), Proof of algorithms: A new kind of certification (Certification of Algorithm 245 TREESORT 3), Comm. ACM, (to appear).

London, R. L. (1970d), Proving programs correct: Some techniques and examples, BIT, (to appear).

London, R. L. and Halton, J. H. (1969), Proofs of algorithms for asymptotic series, Computer Sciences Technical Report No. 54A, University of Wisconsin.

Naur, P. (1966), Proof of algorithms by general snapshots, BIT, 6, pp. 310-316.

Rutledge, J. D. (undated), The problem of correct programming, [1968].