A SUMMARY OF RESEARCH ON PROGRAM DERIVATION

Edmund M. Clarke*

Department of Computer Science
Duke University

October 10, 1978

CS-1978-9

*Supported by National Science Foundation Grant No. MCS-7508146

A SUMMARY OF RESEARCH ON PROGRAM DERIVATION

1. INTRODUCTION

The prospect of enhancing the reliability of computer software by mathematical proofs of program correctness has been frequently discussed in the computer science literature. Critics of this method for obtaining reliable software have pointed to two apparent obstacles: (1) a high degree of mathematical sophistication is required of programmers and (2) proofs of programs of moderate size seem excessively complex. If these obstacles are to be overcome, it is necessary to formulate simple proof systems for realistic programming languages and to develop methods of automating the construction of program proofs. While the problems associated with proofs of sequential programs are well understood, they are not well understood for recursive programs, coroutines, or concurrent programs. Our research has dealt with the development of proof techniques for these three classes of programs. Specifically, we have developed a *fixed point theory* of program invariants which applies to both recursive and concurrent programs. This fixed point theory simplifies the derivation of sound and complete proof systems for recursive programs and permits the automatic synthesis of resource invariants for concurrent programs. We have also developed techniques which do not require the use of program histories for proving the correctness of coroutines.

Section 2 contains an account of the fixed point theory of program invariants as it applies to recursive programs. A detailed account of this research [CK77b] was presented at the 18th FOCS conference in Providence, Rhode Island in November 1977 and has also been accepted for publication in *Computing*. Section 3 outlines how this fixed point theory can be modified to apply to concurrent programs, and in particular, to the synthesis of resource invariants. An account of this research will be presented at the 6th POPL conference in San Antonio, Texas in January 1978.

In Section 4 a proof system for coroutines is described which permits proofs to be constructed without the use of program histories. A full account of this proof system appears in [CK78a] and has been submitted for publication in *Acta Informatica*. The paper concludes in Section 5 with a discussion of plans for future research.

2. PROGRAM INVARIANTS AS FIXED POINTS

Proof systems for correctness of computer programs may be treated as formal logical systems. Of particular interest are deductive systems for partial correctness based on the use of *invariant assertions*. Examples of such systems are described by Floyd [FL67] and Hoare [HO69]. Formulas in *Floyd-Hoare Axiom Systems* are triples $\{P\}\ A\ \{Q\}$ where A is a statement of the programming language and P and Q are predicates in the language of first order predicate calculus (the *assertion language*). A *partial correctness formula* $\{P\}\ A\ \{Q\}$ is true iff whenever P is satisfied by the initial program state and A is executed, then either A will fail to terminate or Q will be satisfied by the final program state. An axiom or *rule of inference* is associated with each statement type in the programming language, e.g.

$$\frac{\{P \wedge b\}\ A\ \{P\}}{\{P\}\ \text{while b do A}\ \{P \wedge \sim b\}}$$

Proofs of correctness for programs are constructed by using the rules of inference for the individual statements together with a proof system for the assertion language.

Once a method of proof has been formalized, it becomes important to determine which steps in the proof process are most difficult. With Floyd-Hoare Axiom Systems experience shows that there are two main sources of difficulty: (i) choosing the correct program invariants (e.g. P in the *while* axiom above) and (ii) demonstrating the truth of formulas in the underlying assertion language. This observation is

justified by the work of Cook [CO75] on relative completeness theorems for Floyd-Hoare Axiom Systems. Cook gives an axiom system for a subset of Algol including the *while* statement and nonrecursive procedures. He proves that (a) if the assertion language satisfies a natural expressibility condition which guarantees the existence of program invariants, and (b) if there is a complete proof system for the assertion language (e.g. the set of true formulas of the assertion language), then a partial correctness formula is true iff it is provable. Extensions of Cook's work to other language features are discussed by Gorelick ([GO75], recursive procedures), Owicki ([OW76a], concurrent processes), Clarke ([CK77a], procedures with procedure parameters under various restrictions on scope of variables), and Cherniavsky ([CH77], loop languages). Incompleteness results for language features, such as call-by-name paramter passing, are given in Clarke [CK77a] and in Lipton and Snyder [LI77].

Completeness results appear to be an important tool in investigations of program correctness; however, many basic open questions remain about the derivation and interpretation of such results. Although proofs of soundness and completeness are often long and tedious, it seems that the underlying ideas are quite simple. Are there general theorems from which many different completeness results may be obtained as special cases? What is the relationship between Cook's definition of completeness and the definition of completeness used in the earlier work of Manna [MA70] and deBakker and Meertens [DE73]? Gorelick [GO75], for example, has shown that a set of three axioms gives completeness for a language with parameterless recursive procedures. DeBakker and Meertens [DE73], on the other hand, *prove* that an infinite pattern of inductive assertions is necessary to obtain completeness for the same class of programming languages. Is there a way of reconciling these apparently contradictory results?

In [CK77b] we argue that the relative soundness and completeness theorems of Cook *et al.* are really *fixed point theorems*. We give a characterization of program invariants as maximal *fixed points* of functionals which may be obtained in a natural manner from the program text. We further show that within the framework of this fixed point theory, relative soundness and completeness theorems have a very simple interpretation. Completeness of a Floyd-Hoare Axiom System is equivalent to the existence of a fixed point for an appropriate functional, and the soundness of the axiom system follows from the maximality of this fixed point.

The functionals associated with *regular recursive procedures* (i.e., flow-chartable recursions) are similar to the *predicate transformers* of Dijkstra ([DI73], [DE73], [GE75]). Let WT[A](Q) (WP[A](Q)) be the *weakest precondition for total correctness* (*partial correctness*) associated with statement A and postcondition Q. If "PROC X = τ" is a regular procedure declaration and Q is a predicate, then the *partial correctness functional* Γ associated with X and Q is defined as follows: $\Gamma(U) = G[\tau](Q)$ where

(A) $G[A_1;A_2)](R) = G[A_1](G[A_2](R))$

(B) $G[(b \rightarrow A_1, A_2)](R) = (b \wedge G[A_1](R)) \vee (\sim b \wedge G[A_2](R))$

(C) $G[A](R) = WT[A](R) = WP[A](R)$ if A is a null statement or an assignment statement

(D) $G[X](R) = U$

S will represent the set of all possible program states.

THEOREM (Regular Fixed Point Theorem): *Let* PROC X = τ *be a regular procedure declaration,* $Q \subseteq S$ *be a predicate, and* Γ *be the partial correctness functional associated with* X *and* Q. *Then the fixed points of* Γ *(invariants of* X*) from a complete lattice under the natural partial ordering on* 2^S. *The top element of the lattice (unique maximal fixed point) is* WP[X](Q) *and the bottom element of the lattice (unique minimal fixed point) is* WT[X](Q).

In order to treat non-regular recursions we must generalize the notion of a program invariant from a single predicate to a binary relation on predicates which is preserved by procedure calls. The appropriate functional in this case is a generalization of the predicate transformer concept which we call a *relational transformer*. The relational transformer maps a programming language statement into the partial correctness relation determined by that statement. Since this mapping is continuous, a fixed point theorem characterizing the partial correctness relations of recursive procedures may be obtained.

Applications of the fixed point theorems include (a) a proof of the soundness and completeness of the inductive assertion method, (b) a proof of the inadequacy of the inductive assertions method for non-regular recursion schemes, and (c) alternate derivations of the completeness results of deBakker [DE73] and Gorelick [GO75] for non-regular recursion schemes. We believe that the fixed point theory in [CK77b] may also be helpful in understanding recent incompleteness theorems for Floyd-Hoare axiom systems ([CK77], [LI77]). These theorems use classical undecidability techniques to show that for certain programming language constructs it is

impossible to obtain axiom systems which give soundness and completeness. We conjecture that for many programming language constructs it may be easier to prove the non-existence of a fixed point than to prove an undecidability result. Finally, we are exploring the question of whether the explicit formulas for program invariants given in [CK77b] may be used to automatically generate invariants of loops and recursive procedures. The work of Suzuki [SU76] and Cousot [CU77] suggests that invariants can indeed be generated in this manner.

3. SYNTHESIS OF RESOURCE INVARIANTS

Researchers in program verification have often argued that it is premature to develop methods for proving the correctness of concurrent programs. The argument usually runs as follows: "Since sequential programs are a subset of concurrent programs, how can we expect to develop methods for verifying concurrent programs when practical methods do not yet exist for the verification of sequential programs." If we view sequential programs as a subclass of concurrent programs, and if we attempt to prove the same type of correctness results for concurrent programs as for sequential programs (e.g., program P computes function F), then the above argument may be correct. On the other hand, if we abstract from a concurrent program that portion of the program which deals with synchronization (the *synchronization skeleton*), and if we restrict correctness proofs to synchronization properties (e.g., absence of deadlock), then it is possible to argue that concurrent programs are

easier to verify than sequential programs. Three reasons in support of this view are:

(1) The synchronization skeleton of a concurrent program is generally very small in comparison to the entire program. A crude count of the UNIX operating system indicates that less than 5% of the total statements are synchronization statements. Because long concurrent programs need not have long synchronization skeletons, we believe that the arguments expressed in [DE77] regarding the feasibility of correctness proofs for sequential programs are not applicable to proofs of correctness for synchronization skeletons.

(2) In many cases, it is relatively easy to isolate the synchronization skeleton of a concurrent program from the sequential part of the program. Consider, for example, the typical producer-consumer program in which two processes communicate via a message buffer; here the synchronization skeleton is independent of the sequential producer and consumer processes.

(3) A large class of synchronization problems are counting problems. Thus, the verification conditions generated in the correctness proofs can be expressed in terms of linear inequalities involving the synchronization variables, and techniques such as integer programming can be exploited to determine their validity.

What verification techniques are best suited for proving the correctness of synchronization skeletons? Hoare [HO72], and Owicki and Gries [OW76b] have developed a proof system for *conditional critical regions* in which logically related variables which must be accessed by more than one process are grouped together as *resources*. Individual

processes are allowed to access a resource only within a conditional critical region for that resource. Proofs of synchronization properties are constructed by devising predicates called *resource invariants* which describe the relationship among the variables of a resource when no process is within a critical section for the resource. Related methods for verifying concurrent programs which use resource invariants have been described by Habermann [HA72], Lauer [LU72], Keller [KE76], Lamport [LM77], and Pneuli [PN77].

In constructing correctness proofs for synchronization skeletons using the proof system of Hoare-Owicki-Gries, the programmer is required to supply the resource invariants. In [CK78b] we investigate the problem of automatically synthesizing resource invariants for synchronization skeletons of concurrent programs. The synchronization skeletons are expressed in a simple concurrent programming language (SCL) in which parallelism is introduced via "cobegin...coend" blocks and processes access shared data via conditional critical regions. We consider only *invariance* [PN77] or *safety* properties [LA76] of SCL programs. This class of properties includes mutual exclusion and absence of deadlock and is analogous to partial correctness of sequential programs. Correctness proofs of SCL programs are expressed in a proof system similar to the system of Hoare-Owicki-Gries; we will refer to such proofs as *resource invariant proofs*.

To gain insight on synthesis methods for resource invariant proofs we restrict the SCL language so that all processes are nonterminating loops of the form "cycle $S_1,\ldots,S_n$ end" and the only statements allowed in a process are P and V operations on semaphores. We call

this class of SCL programs *PV programs*. For PV programs there is a simple method for generating resource invariants, i.e., the *semaphore invariant method* of Habermann [HA72] which expresses the current value of a semaphore in terms of its initial value and the number of P and V operations which have been executed. Although the semaphore invariant is simple to state, it is quite powerful as a technique for proving PV programs. We show that the semaphore invariant method is at least as powerful as the *reduction method* of Lipton [LI75] for proving freedom from deadlock.

The semaphore invariant method, however, is not complete for proving either absence of deadlock or mutual exclusion of PV programs. We prove that it is possible to devise PV programs for which deadlock (mutual exclusion) is impossible, but the semaphore invariant method is not sufficiently powerful to establish this fact. This incompleteness result is important because it demonstrates the role of *convexity* in the generation of powerful resource invariants. We also give a characterization of the class of PV programs for which the semaphore invariant method is complete for proving absence of deadlock (mutual exclusion).

The semaphore invariant method can be generalized to the class of *linear SCL programs* in which solutions to many standard synchronization problems including the dining philosopher's problem, the reader's writers problem and the cigarette smoker's problem can be expressed. Although the generalized semaphore invariant fails to be complete for exactly the same reason as the standard semaphore invariant method, it is sufficiently powerful to permit proofs of mutual exclusion and absence of deadlock for a significant class of concurrent programs (e.g., an implementation of the readers and writers problem with writer priority).

When the generalized semaphore invariant is not sufficiently powerful to prove some desired property of an SCL program, is it possible to synthesize a stronger resource invariant? We argue that resource invariants are *fixed points*, and that by viewing them as fixed points it is possible to automatically generate invariants which are considerably stronger than the semaphore invariants previously described. Specifically we show that the resource invariants of an SCL program C are fixed points of a functional F_C which can be obtained in a natural manner from the text of program C. The *least fixed point* $\mu(F_C)$ of F_C is the "strongest" such resource invariant, e.g., if the program C is free of deadlock then this fact may be established using $\mu(F_C)$. Since the functional F_C is continuous, the least fixed point $\mu(F_C)$ may be expressed as the limit $\mu(F_C) = \vee_{j=1}^{\infty} F_C^j(\text{false})$. Because of the infinite disjunction in the formula for $\mu(F_C)$, this characterization of $\mu(F_C)$ cannot be used directly to compute $\mu(F_C)$ unless C has only a finite number of possible different states.

By using the notion of *widening* of Cousot [CU78] however, we are able to speed up the convergence of the chain $F_C^j(\text{false})$ and obtain a close approximation to $\mu(F_C)$ in a finite number of steps. The widening operator which we use exploits our earlier observation on the importance of convexity in the generation of resource invariants. Examples are given in [CK78b] to illustrate the power of this new technique for generating resource invariants. Although fixed point techniques have been previously used in the study of resource invariants ([LA76], [EN77]), we believe that this is the first research on methods for speeding up the convergence of the sequence of approximations to $\mu(F_C)$.

4. PROVING CORRECTNESS OF COROUTINES WITHOUT HISTORY VARIABLES

A number of important theoretical problems arise in the development of axiomatic proof systems for programming languages which allow coroutines. One such problem is the question of whether *history variables* are necessary in proving partial correctness of coroutines. History variables are special variables which are added to a program to facilitate its proof by recording the sequence of states reached by the program during a computation; after the proof has been completed the history variables may be deleted. The use of such variables in correctness proofs was first suggested by Clint [CL73] in a paper entitled "Program Proving: Coroutines"; subsequently, history variables have been used by Owicki [OW76] and Howard [HW76] in verifying concurrent programs and by Apt [APT77] in verifying sequential programs. Owicki and Howard have conjectured that history variables are necessary for proofs of some concurrent programs.

The obvious power of history variables in program proofs stems from the large amount of information about a program's behavior which can be obtained by examining execution sequences. This power, however, is not available without a sacrifice. Program histories are much more difficult to manipulate in partial correctness assertions than simple program identifiers. Another, less obvious, disadvantage is that it is no longer possible to construct program proofs in a top-down manner in which only the input-output behavior of a statement is used to relate the statement to the remainder of the program. Instead it is necessary to consider the entire history of the program's execution in constructing proofs. Because

of these disadvantages we believe that the use of history variables should be avoided whenever possible. Since history variables seem to be essential for correctness proofs of certain language constructs, it becomes important to identify the constructs where such variables can be avoided.

In [CK78a] we show that history variables are NOT needed in proving the correctness of simple coroutines. We give a modification of Clint's axiom system and a strategy for generating proofs in which the only auxiliary variables needed (in addition to the program identifiers) are simple program counters. Since the program counters have bounded magnitude, they may be encoded by 0, 1-valued auxiliary variables. We illustrate our method of proving coroutines with examples and give a proof of soundness and relative completeness for our axiom system. Finally, we discuss some extensions of the coroutine concept which do appear to require the use of history variables. A brief description of our axiom system and an example illustrating our technique for avoiding the use of history variables are given in the remainder of this section.

A coroutine will have the form:

coroutine Q_1, Q_2 end

Q_1 is the *main routine*; execution begins in Q_1 and also terminates in Q_1 (the requirement that execution terminate in Q_1 is not absolutely necessary but simplifies the axiom for coroutines). Otherwise Q_1 and Q_2 behave in identical manners. If an "exit" statement is encountered in Q_1, the next statement to be executed will be the statement following the last "resume" statement in Q_2. Similarly, the execution of a "resume" statement in Q_2 causes execution to be restarted following the last

"exit" statement executed in Q_1. A simple example of a coroutine is:

```
Coroutine
   while y≠z do
      y:=y+1; x:=x+y; exit;
      end,
   while true do
      y:=y-2; resume;
      y:=y+1; resume;
      end
   end
```

This example illustrates the use of "exit" and "resume" statements within while loops. Note that if x and y are 1 initially and $z \geq 1$, then the coroutine will terminate with $y = z^2$.

Four axioms are needed to adequately specify the semantics of the *coroutine* statement. These axioms are given below. Although axiom C1 is similar to Clint's coroutine axiom, our strategy for generating proofs is different from that advocated by Clint; auxiliary variables represent program counters (and therefore have bounded magnitude) rather than arbitrary stacks.

C1. (Coroutines)

$$\frac{\{P'\}\ \text{exit}\ \{R'\} \vdash \{P \wedge b\}Q_1\{R\} \qquad \{R'\}\ \text{resume}\ \{P'\} \vdash \{P' \wedge b\}Q_2\{R'\}}{\{P \wedge b\}\ \text{coroutine}\ Q_1, Q_2\ \text{end}\ \{R\}}$$

provided that no variable free in b is global to Q_1. (This axiom is a modification of the one in [CL73].)

C2. (Exit)

$$\frac{\{P'\}\ \text{exit}\ \{R'\}}{\{P' \wedge C\}\ \text{exit}\ \{R' \wedge C\}}$$

provided that C does not contain any free variables that are changed by Q_2. (Here we assume that "exit" occurs in statement Q_1 of "coroutine Q_1, Q_2 end".)

C3. (Resume)

$$\frac{\{R'\}\ \text{resume}\ \{P'\}}{\{R' \wedge C\}\ \text{resume}\ \{P' \wedge C\}}$$

provided that C does not contain any free variables that are changed in Q_1. (Here we assume that "resume" occurs in statement Q_2 of "coroutine Q_1, Q_2 end".)

C4. (Auxiliary Variables)

Let AV be a set of variables such that $x \in AV$ if and only if x appears in S' only in assignments $y:=e$ with $y \in AV$. If P and Q are assertions which do not contain any free variables from AV and if S is obtained from S' by deleting all assignments to variables in AV, then

$$\frac{\{P\}S'\{Q\}}{\{P\}\ S\ \{Q\}} .$$

(This axiom is essentially the same as the auxiliary variable axiom in [OW76].)

We illustrate the axioms with an example. We show that $\{x=1 \wedge y=1 \wedge z \geq 1\} A \{x=z^2\}$ where $A \equiv$ "coroutine Q_1, Q_2 end" is the coroutine given in Section 2. Our strategy in carrying out the proof will be to introduce auxiliary variables to distinguish the various "exit" and "resume" statements from each other and then use axiom C4 to delete these unnecessary variables as the last step of the proof. Axiom C2 enables us to adapt the general exit assumption $\{P'\}$ exit $\{R'\}$ to a specific occurrence of an exit statement in Q_1. A similar comment applies to axiom C3 for the resume statement. We prove:

$\{x=1 \wedge y=1 \wedge z \geq 1\}$

```
   i:=0; j:=0;
   coroutine
      while y≠z do
         y:=y+1; x:=x+y;
         i:=1; exit;
         end,
      while true do
         y:=y-2; j:=1; resume;
         y:=y+1; j:=0; resume;
         end
   end
```

$\{x=z^2\}$

Note that two auxiliary variables are needed (one for each routine of the coroutine). The auxiliary variable j of the second routine is assigned a different value prior to each "resume" statement and is not changed by the first routine. Thus the value of j can be used in assertions to distinguish which of the resume statements has been most recently executed. The auxiliary variable i of the first routine has a dual function. This technique of adding auxiliary variables will be formally described in [CK78a] however, the general pattern should be clear from the above example. To complete the proof we choose:

$P = \{x=1 \wedge y=1 \wedge z \geq 1 \wedge i=0 \wedge j=0\}$

$b = \{j=0\}$

$R = \{x=z^2\}$

$P' = \{(x=y^2-y+1 \wedge j=0 \wedge y \leq z) \vee (x=y^2+2y+1 \wedge j=1 \wedge y \leq z-1)\}$

$R' = \{(x=y^2+3y+3 \wedge j=1 \wedge y \leq z-2) \vee (x=y^2 \wedge j=0 \wedge y \leq z)\}$

The invariant for the while loop of the first routine is:

$$INV_1 = \{(x=y^2+3y+3 \wedge j=1 \wedge y \leq z-2) \vee (x=y^2 \wedge j=0 \wedge y \leq z)\}$$

The invariant for the while loop of the second routine is:

$$INV_2 = \{x=y^2-y+1 \wedge j=0 \wedge y \leq z\}$$

Using axioms C2-C4 together with the axioms for the assignment statement and the while statement, it is possible to prove that:

(a) $\{P'\}$ exit $\{R'\} \vdash \{P \wedge b\}Q_1\{R\}$

and

(b) $\{R'\}$ resume $\{P'\} \vdash \{P' \wedge b\}Q_2\{R'\}$

both hold. For example, to prove (b) we assume $\{R'\}$ resume $\{P'\}$ and prove $\{P' \wedge b\}Q_2\{R'\}$. In order to prove $\{P' \wedge b\}Q_2\{R'\}$ we show that

(c) $P' \wedge b \rightarrow INV_2$

(d) $\{INV_2\}$

```
while true do
   y:=y-2; j:=1; resume;
   y:=y+1; j:=0; resume;
   end
```

$\{INV_2 \wedge \sim true\}$

(e) $INV_2 \wedge \sim true \rightarrow R'$ are true.

Steps (c) and (e) are easily verified. Step (d) follows from the while axiom and the sequence of assertions below:

(d1) assignment

$\{INV_2 \wedge true\}$ y:=y-2; j=1$\{R' \wedge j=1\}$

(d2) resume

$\{R' \wedge j=1\}$ resume $\{P' \wedge j=1\}$

(d3) assignment

$\{P' \wedge j=1\}$ y:=y+1; j=0$\{R' \wedge j=0\}$

(d4) resume

$\{R' \wedge j=0\}$ resume $\{P' \wedge j=0\}$

(d5) arithmetic

$P' \wedge j=0 \rightarrow INV_2$

Once (a) and (b) have been established, the desired conclusion follows immediately by axiom C1.

5. DIRECTIONS FOR FUTURE RESEARCH

An obvious open question concerns the use of history variables in correctness proofs for concurrent programs. Owicki, for example, shows that her proof system for conditional critical regions is sound and complete with respect to an operational semantics for her language [OW76a]. The proof of completeness requires the use of history variables to record the order in which critical regions are entered. In fact, it is not difficult to construct examples of programs in which the number of history variables required to express the resource invariants is greater than the number of variables in the original program. In general the number of history variables needed appears to depend on the product of the number of processes and the number of different resources. Because of the number required and the amount of information recorded, we believe that the use of history variables in proofs of program correctness is unnatural and should be avoided. Gerhart [GE76] has shown that history variables are unnecessary in proofs of correctness for sequential programs. We conjecture that they are also unnecessary for large classes of concurrent programs. We are currently investigating ways of modifying the resource invariant axioms of Hoare and Owicki to support this conjecture.

Finally the author is currently designing an automatic verification system for concurrent programs based on the ideas in Section 4. This system will extract the synchronization skeleton of a concurrent program and use the techniques of [CK78b] to generate the appropriate resource invariants. If this system or some similar system is successful in generating correctness proofs for synchronization skeletons, then a

particularly intriguing idea for verifying large concurrent programs would be to combine formal proofs of correctness for the synchronization part of the program with other techniques such as testing or symbolic execution for the sequential part of the program.

[APT77] Apt, K. R., Bergstra, J. A., and Meertens, L. G. L. T. Recursive assertions are not enough--or are they? Mathematical Centre IW 92/77.

[CH77] Cherniavsky, J. and S. Kamin. A complete and consistent Hoare axiomatics for a simple programming language. Proceedings of the 4th POPL, 1977.

[CK77a] Clarke, E. M. Programming language constructs for which it is impossible to obtain good Hoare-like axiom systems. Proceedings of the 4th POPL, 1977.

[CK77b] Clarke, E. M. Program invariants as fixed points. Proceedings of the 18th FOCS, 1977.

[CK78a] Clarke, E. M. Proving correctness of coroutines without history variables. Technical Report No. CS-1978-4, Department of Computer Science, Duke University, 1978.

[CK78b] Clarke, E. M. Synthesis of resource invariants. To be presented at the 6th POPL conference in San Antonia, Texas in January 1979.

[CL73] Clint, M. Program proving: Coroutines. Acta Informatica, 2:50-63, 1973.

[CO75] Cook, S. A. Axiomatic and interpretative semantics for an algol fragment. Technical Report 79, Department of Computer Science, University of Toronto, 1975.

[CU77] Cousot, P. and R. Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. Proceedings of the 4th POPL, 1977.

[DE73] deBakker, J. W. and Meertens, L. G. L. T. On the completeness of the inductive assertion method. Mathematical Centre, December 1973.

[DI73] Dijkstra, E. E. A simple axiomatic basis for programming language constructs. Lecture notes from the Internation Summer School on Structured Programming and Programmed Structures, Munich, Germany 1973.

[FL67] Floyd, R. W. Assigning meaning to programs. In Schwarts, J. T., Ed., Mathematical Aspects of Computer Science Proc. Symposia in Applied Mathematics 19, pp. 19-32, Amer. Math. Soc., 1967.

[FN77] Flon, L. and Suzuki, N. Nondeterminism and the correctness of parallel programs. Carnegie Mellon University, Department of Computer Science, May 1977.

[GE75] Gerhart, S. L. Proof theory of partial correctness verification systems. SIAM J. Comput., Vol. 5, No. 3, September 1975.

[GO75] Gorelick, G. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report No. 75, Department of Computer Science, University of Toronto, January 1975.

[HA72] Habermann, A. N. Synchronization of communicating processes. Comm. ACM, 15, 3(March 1972), 171-176.

[HO69] Hoare, C. A. R. An axiomatic approach to computer programming, CACM 12, 10(October 1969), pp. 322-329.

[HO72] Hoare, C. A. R. Towards a theory of parallel programming. Operating Systems Techniques, C. A. R. Hoare, R. H. Perrot, Editors, Academic Press, 1972.

[HW76] Howard, J. H. Proving monitors. COMM ACM, 19(5):273-279, May 1976.

[KE76] Keller, R. M. Formal verification of parallel programs. CACM, 19(7), 1976.

[LA76] van Lamsveerde, A. and Sintzoff, M. Formal derivation of strongly correct parallel programs. MBLE Research Report, Brussels, Belgium 1976.

[LI75] Lipton, R. J. Reduction: A new method of proving properties of systems of processes. Proceeding of 2nd ACM Symposium on Principles of Programming Languages, 78-86, 1975.

[LM77] Lamport, L. Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering, 3(2), 1977, 125-143.

[MA70] Manna, Z. and Pnueli, A. Formalization of properties of functional programs. JACM 17(3):555-569, 1970.

[OW76a] Owicki, S. A consistent and complete deductive system for the verification of parallel programs. 8th Annual Symposium on Theory of Computing, 1976.

[OW76b] Owicki, S. and Gries, D. Verifying properties of parallel programs: An axiomatic approach. CACM 19(5):279-284, 1976.

[PN77] Pnueli, A. The temporal logic of programs. 18th Annual Symposium on Foundations of Computer Science, November 1977.

[SU77] Suzuki, N. and Ishihata, K. Implementation of an Array Bound Checker. Proceedings of the 4th POPL, 1977.