Research Directions in Programming Language Semantics and Formal Program Verification

Edmund M. Clarke, Jr.

TR-22-81

Center for Research in Computing Technology
Harvard University

Research Directions in Programming Language Semantics and Formal Program Verification

1. Introduction

Program correctness is one of the most serious problems in the construction of large software systems. Between one-third and one-half of the effort that goes into the development of a large software system is spent on program testing and debugging [Ko76]. If the indirect costs of program errors (e.g. loss of critical information in a data base management system) are taken into account, the problem becomes even more disturbing. Although correctness is not the only desirable property of a software system, it is definitely the most basic. When a program contains an error, issues such as efficiency and hardware fault tolerance may become meaningless.

One method for obtaining more reliable software is to *prove* programs correct in the same way that we prove mathematical theorems. We should be able to have as much confidence in the correctness of a sorting program as we have in the Pythagorean Theorem. Critics of this method for obtaining reliable software point to two apparent obstacles: the high degree of mathematical sophistication required by programmers and the complexity of proofs needed for large programs. If these obstacles are to be overcome, programming languages must be designed for which the proof systems are simple and easy to use. In addition, methods for automatically constructing program proofs will be needed. In this paper we describe some of the recent research at Harvard on these two aspects of formal program verification. In section 2 we characterize the class of programming languages for which an axiomatic semantics is appropriate, and in section 3 we describe techniques that can be used to automate the construction of correctness proofs for concurrent programs.

2. Hoare Axioms and the Semantics of Control Structures

A key trend in program verification has been the use of axioms and rules of inference to specify the meanings of programming language constructs. This approach was first suggested by C.A.R. Hoare in 1969 [Ho69]. Although the most complicated control structure in Hoare's original paper was the **while** statement, there has been considerable success in extending his method to other language features. Axioms have been proposed for the **goto** statement, functions, recursive procedures with value and reference parameter passing, simple coroutines, and concurrent programs. Research by Clarke [Cl79] has shown, however, that there are natural programming language control structures which are impossible to describe adequately be means of Hoare axioms. Specifically, Clarke has shown that there are control structures for which it is impossible to obtain axiom systems which are sound and complete in the sense of Cook [Co78]. These constructs include procedures with procedure parameters under standard

Algol 60 scope rules, recursive procedures with call by name parameter passing, and coroutines in a language with parameterless recursive procedures.

In this section we outline how the incompleteness results are obtained in the case of procedure parameters and suggest ways of modifying Algol 60 scope rules to obtain good axiom systems. We also discuss the significance of these results. We argue that the incompleteness theorems provide additional evidence for the importance of programming languages with simple, clean control structures.

2.1. Background

The formulas in a ***Hoare axiom system*** are triples {P} S {Q} where S is a statement of the programming language and P and Q are predicates describing the initial and final states of the program S. The logical system in which the predicates P and Q are expressed is called the ***assertion language*** (AL) and is an applied version of first order predicate calculus. The triple {P} S {Q} is true iff whenever P holds for the initial program state and S is executed, then either S will fail to terminate or Q will be satisfied by the final program state. We call such triples ***partial correctness formulas.***

The control structures of the programming language are specified by axioms and rules of inference for the partial correctness formulas. A typical rule of inference is

$$\frac{\{P \wedge b\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{P \wedge \sim b\}}.$$

The predicate P is the ***invariant*** of the **while** loop. Proofs of correctness for programs are constructed by using the axioms together with a proof system T for the assertion language. We write $\vdash_{H,T}\{P\}\ S\ \{Q\}$ if the partial correctness formula {P} S {Q} is provable using the Hoare axiom system H and the proof system T for the assertion language AL.

To discuss whether a particular Hoare axiom system adequately describes the programming language PL, it is necessary to have a definition of ***truth*** for partial correctness formulas which is independent of the axiom system H. The definition of truth requires two steps. First, we give an interpretation I for the assertion language AL. The interpretation I specifies the primitive data objects of our programming language; it consists of a set D (the domain of the interpretation) and an assignment of predicates and functions on D to the predicate and function symbols of AL. Typical interpretations might be the integers with the standard functions and predicates of arithmetic, or linear lists with the list processing functions car, cdr, etc.

Second, we provide an interpreter for the statements of the programming language. There are many ways such an interpreter may be specified—in terms of computation sequences or as the least fixed point of a continuous functional (denotational semantics). The net result

is a function M[S](s) = s′ which associates with each statement S and state s a new state s′. Once the meaning function M has been specified a formal definition may be given for partial correctness.

> **2.1.1 Definition:** The partial correctness formula {P} S {Q} is ***true with respect to interpretation*** I ($\models_I$ {P} S {Q}) iff for all states s and s′, if predicate P holds for state s under interpretation I and M[S](s) = s′, then Q must hold for s′ under I also.

2.2. Soundness and Completeness

When can we be satisfied that a Hoare axiom system H adequately describes the programming language PL? There are two possible ways a Hoare axiom system may be inadequate. First, some theorem {P} S {Q} which can be proven in the axiom system may fail to hold for actual executions of the program S, i.e., there is a terminating computation of S such that the initial state satisfies P but the final state fails to satisfy Q. A way of preventing this source of error is to adopt an operational or denotational semantics for the programming language which is close to the way statements are actually executed. We then show that every theorem which can be proven using the axiom system will be true in the model of program execution that we have adopted. In the notation defined above we prove that for all P, Q, S, if $\vdash_{H,T}$ {P} S {Q} then $\models_I$ {P} S {Q}. Logicians call this property ***soundness*** or ***consistency***.

A second source of inadequacy is that the axioms for the programming language may not be sufficiently powerful to handle all combinations of the control structures of the language. The question of when it is safe to stop looking for new axioms is much more difficult to answer than the question of soundness. One solution is to prove a completeness theorem for the Hoare axiom system. We can attempt to prove that every partial correctness formula which is true of the execution model of the programming language is provable in the axiom system. In general it is impossible to prove such completeness theorems; the proof system for the assertion language may itself fail to be complete. For example, when dealing with the integers for any consistent axiomatizable proof system, there will be predicates which are true of the integers but not provable within the system. Also the assertion language may not be powerful enough to express the invariants of loops. This difficulty occurs if the assertion language is Presburger arithmetic (integer arithmetic without multiplication). Note that both of the above difficulties are faults of the underlying assertion language and not of the Hoare axiom system.

How can we talk about the completeness of a Hoare axiom system independently of its assertion language? Cook [Co78] gives a Hoare axiom system for a subset of Algol including the while statement and nonrecursive procedures. He then proves that if there is a complete proof system for the assertion language (e.g. all true statements of the assertion language) and if the assertion language satisfies a certain natural expressibility condition, then every partial correctness assertion will be provable.

2.2.1 Definition: A Hoare axiom system H for a programming language PL is ***sound*** and ***complete*** (in the sense of Cook) iff for all AL, T, and I such that (a) AL is expressive with respect to I, and (b) T is a complete proof system for AL with respect to I,

$$\models_I \{P\}\ S\ \{Q\} \Leftrightarrow \vdash_{H,T} \{P\}\ S\ \{Q\}$$

2.3. Incompleteness Results

We next consider the problem of obtaining a sound and complete axiom system for an Algol-like language which allows procedures as parameters of procedure calls.

2.3.1 Theorem: It is impossible to obtain a system of Hoare-like axioms H which is sound and complete in the sense of Cook for a programming language PL which allows:

(i) procedures as parameters of procedure calls
(ii) recursion
(iii) static scope
(iv) global variables
(v) internal procedures as parameters of procedure calls

All of the features (i) - (v) are found in Algol 60 and in Pascal. In [Cl79] we show that a sound and complete axiom system can be obtained by modifying any one of the five features of the language PL. Thus if we change from *static scope* to *dynamic scope*, a complete set of axioms may be obtained for (i) procedures with procedure parameters, (ii) recursion, (iv) global variables, and (v) internal procedures as parameters; or if we disallow internal procedures as parameters, a complete system may be obtained for (i) procedures with procedure parameters, (ii) recursion, (iii) static scope, and (iv) global variables.

The incompleteness results are established by observing that if a programming language P has a sound and relatively complete proof system for all expressive interpretations, then the halting problem for P must be decidable for finite interpretations. Lipton [Li77] considered a form of converse: If P is an *acceptable* programming language and the halting problem is decidable for finite interpretations, then P has a sound and relatively complete Hoare logic for expressive and effectively presented interpretations. The acceptability of the programming language is a mild technical assumption which ensures that the language is closed under certain reasonable programming constructs, and that given a program, it is possible to effectively ascertain its step-by-step computation in interpretation I by asking some quantifier-free questions about I.

Lipton actually proved a partial form of the converse. He showed that given a program P and the effective presentation of I, it is possible to enumerate all the partial correctness assertions of the form $\{true\}\ S\ \{false\}$ which are true in I. From this it easily follows that we can enumerate all true quantifier-free partial correctness assertions, since we can encode

quantifier-free tests into the programs. But it does *not* follow that we can enumerate all first-order partial correctness assertions, since an acceptable programming language will not in general allow first-order tests.

In [Cl81b] we consider acceptable programming languages which permit recursive procedure calls. We also require, for technical reasons, that every element of the domain of I correspond to some term in the assertion language. (These requirements seem quite reasonable.) Under these assumptions we are able to significantly extend the results of [Cl79] and [Li77]:

1. We are able to eliminate the requirement that pre- and post-conditions be quantifier-free and that the interpretation be effectively presented. Under the assumption that the halting problem for P is decidable for finite interpretations, we show that for all expressive interpretations P has a sound and relatively complete Hoare axiom system for partial correctness assertions with arbitrary first-order pre- and post-conditions.

2. We show, in fact, that the set of partial correctness assertions true in I is actually (uniformly) decidable in the theory of I (Th(I)) provided that the halting problem for P is decidable for finite interpretations. Lipton's proof, on the other hand, produces an enumeration procedure for partial correctness assertions and, thus, shows only that the set of true partial correctness assertions is r.e. in Th(I).

3. We extend the decidability result to termination assertions (which coincide with total correctness assertions for deterministic programming languages). Here even stronger results can be obtained. The set of true termination assertions is (uniformly) decidable in Th(I) iff the halting problem for P is decidable for finite interpretations. Moreover, the set of true termination assertions is (uniformly) r.e. in Th(I) even if the halting problem for P is not decidable for finite interpretations.

This last result unexpectedly suggests that good axiom systems for total correctness may exist for a wider spectrum of languages than is the case for partial correctness. In particular, it may be possible to find a sound and relatively complete total correctness proof system for a language with call by name parameter passing, recursive procedures, functions, and global variables, even though no corresponding partial correctness proof system can exist.

3. Verification of Concurrent Systems

The rapid development of computer networks and multiprocessor systems has increased the need for methods of constructing reliable concurrent systems. Traditional techniques such as hierarchical development and exhaustive testing that have been successfully used in constructing large sequential programs are no longer adequate because of the high degree of nondeterminism that is inherent in parallel computation. In this section we describe two techniques that can be used to automate the construction of correctness proofs for parallel programs.

3.1. Techniques for Finding Resource Invariants

Parallel processes which operate on disjoint sets of variables are called ***disjoint*** or ***noninteracting*** processes. With disjoint processes it is theoretically possible to achieve the full power of parallelism, since it is never necessary for one process to wait for another. Disjoint processes can therefore be analyzed as a collection of independent sequential programs. Unfortunately, the usefulness of disjoint processes is limited; in most applications, processes must access and change common variables. In order to exploit the full power of parallelism in these cases, it is important to understand and be able to control process interactions.

Hoare [Ho72] and Brinch Hansen [BH73] have proposed ***conditional critical regions*** as a language primitive for controlling process interactions. With this primitive, logically related variables which must be accessed by more than one process are grouped together as ***resources.*** Individual processes are allowed access to a resource R only in a critical region of the form "**with** R **when** b **do** A **od**" where b is a boolean expression and A is a statement whose execution may change the values of the shared variables in R. When execution of a process reaches the conditional critical region, the process is delayed until no other process is using resource R and condition b is satisfied. The statement A is then executed as an indivisible operation.

Owicki and Gries [Ow76] have developed a proof system for conditional critical regions. Proofs of synchronization properties are constructed by devising predicates called ***resource invariants.*** These predicates describe relationships among the variables of a resource when no process is in a critical region for the resource.

In constructing proofs using the system of Hoare-Owicki-Gries, the programmer is required to supply the resource invariants. In [Cl80a] we investigate the possibility of *automatically* synthesizing resource invariants for a simple concurrent programming language (SCL) in which processes access shared data via conditional critical regions. We consider only *invariance* or *safety* properties of SCL programs. This class of properties includes mutual exclusion and absence of deadlock and is analogous to partial correctness for sequential programs. Correctness proofs of SCL programs are expressed in a proof system similar to that of Hoare-Owicki-Gries.

To gain insight on the synthesis of resource invariants we restrict the SCL language so that all processes are nonterminating loops, 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 [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. The semaphore invariant method, however, is not complete for proving either absence of deadlock or mutual exclusion of PV programs. We show in [Cl80c] that there exist PV programs for which deadlock is

impossible, but the semaphore invariant method is insufficiently 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 is generalized to the class of *linear SCL programs* in which solutions to many synchronization problems can be expressed. Although the generalized semaphore invariant also fails to be complete, it is sufficiently powerful to permit proofs of mutual exclusion and absence of deadlock for a significant class of concurrent programs (e.g. the readers and writers problem).

When the generalized semaphore invariant is insufficiently 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 *fixedpoints* of continuous functionals, and that by viewing them as fixedpoints it is possible to generate invariants which are stronger than the semaphore invariants previously described. We show that the resource invariants of an SCL program C are fixedpoints of a functional F which can be obtained from the text of program C and that the ***least fixedpoint*** μF of F is the "strongest" such resource invariant. Since the functional F is continuous, the least fixedpoint μF may be expressed as the limit

$$\mu F = \bigcup_{j \geq 0} F^j(\text{false}).$$

Because of the infinite disjunction, this characterization of F cannot be used to compute μF directly unless C has only a finite number of different states or unless a good initial approximation is available for μF. By using the notion ***widening*** of Cousot [Cou76], however, we are able to speed up the convergence of the chain $F^j(\text{false})$ and obtain a close approximation of μF in a finite number of steps. The widening operator which we use exploits our observation on the importance of convexity in the generation of resource invariants and is described in [Cl80c]. Although fixpoint techniques have been previously used in the study of resource invariants [Si76], we believe that this is the first research on methods for speeding up the convergence of the sequence of approximations to μF. We have developed at Harvard an EL1 program [Cl80c] which uses the above ideas to find resource invariants of nontrivial parallel programs.

3.2. Synthesis of Parallel Programs from Temporal Logic Specifications

Another direction of research focuses on the use of Temporal Logics to specify properties of parallel programs. While these logics differ in their notation and in their expressive power, most have operators such as

$\Diamond$P which means that in every possible future, P *sometimes* holds

and

$\Box P$ which means that for every possible future, P *always* holds.

$\Box P$ is useful in defining *safety* properties which insure that "nothing bad happens". For example, to indicate that two processes P_1 and P_2 satisfy the requirement that they are never in their respective critical sections at the same time (mutual exclusion) we write

$$\Box(\sim CS_1 \wedge \sim CS_2)$$

where CS_i indicates that P_i is in its critical section. Freedom from deadlock is another safety property. $\Diamond P$ is useful in describing *liveness* properties such as absence of starvation and inevitability which insure that "something good does happen". For example, we can write

$$TRY_i \rightarrow \Diamond CS_i$$

to indicate that if process P_i is started in its trying region, it inevitably enters its critical section at some time in the future.

We are currently investigating a method of constructing concurrent programs in which the ***synchronization skeleton*** of the program is automatically synthesized from a high-level Temporal Logic specification. The synchronization skeleton is an abstraction of the actual program where detail irrelevant to synchronization is suppressed. For example, in the synchronization skeleton for a critical section problem each process's critical section may be viewed as an atomic step since the internal structure of the critical section is unimportant. Most solutions to synchronization problems in the literature are in fact given as synchronization skeletons. Because synchronization skeletons are in general finite state, the propositional version of Temporal Logic suffices for specification.

Our synthesis method is based on the ***finite model property*** for an appropriate Propositional Temporal Logic, which asserts that *if a formula of Propositional Temporal Logic is satisfiable, it is satisfiable in a finite model.* Decision procedures have been devised which, given a Temporal Logic formula F, will decide whether F is satisfiable or unsatisfiable. If F is satisfiable, a finite model of F is constructed. In our application unsatisfiability of the formula F means that the specification F is inconsistent (and must be reformulated). If the formula F is satisfiable, then the specification expressed by F is consistent. A model for F with a finite number of states is constructed by the decision procedure. The synchronization skeleton of a program meeting the specification can be read from this model.

The finite model property insures that any program whose synchronization properties can be expressed in Temporal Logic can be realized by a system of concurrently running processes, each of which is in fact a finite state machine. We remark that an inspection of the literature reveals that many programs to solve synchronization problems are finite state. Initially, the synchronization skeletons we synthesize will be for concurrent programs running in a shared-memory environment. However, we also plan to investigate the possibility of

synthesizing distributed programs. We believe, for instance, that network communication protocols can be specified in Propositional Temporal Logic and then automatically synthesized. A preliminary account of our research is given in [Cl81a].

4. Conclusion

Formal program verification is in its infancy. It is unlikely that program proof techniques will be developed to the point where *all* new software systems are routinely verified. However, it is inevitable that certain crucial parts of ultra-reliable software systems will be verified (e.g. the software that controls emergency shutdown of nuclear reactors and the software used in automated air traffic control systems). The only alternative to formal verification is testing and as Dijkstra [Di76] has wisely observed: "In software systems testing can only establish the presence of errors never their absence." We believe that the research described in this paper will significantly contribute to an understanding of the relationship between proof methods and language design and to the development of techniques for automating the verification of parallel programs.

References

[BH73] Brinch Hansen, P. *Operating System Principles.* Prentice-Hall, 1973.

[Cl79] Clarke, E.M. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *JACM 26*, 1 (January 1979).

[Cl80a] Clarke, E.M. Program invarints as fixed points. *Computing 21*, 4 (1980), 273-294.

[Cl80b] Clarke, E.M. Proving correctness of coroutines without history variables. *Acta Informatica 13* (1980), 169-188.

[Cl80c] Clarke, E.M. Synthesis of resource invariants. *TOPLAS 2*, 3 (July 1980), 338-358.

[Cl81a] Clarke, E.M. and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. IBM Conference on Logics of Programs, May 4-6, 1981; to appear in *Springer Lecture Notes in Computer Science.*

[Cl81b] Clarke, E.M., with S.M. German and J.Y. Halperin. On effective axiomatizations of Hoare logics. Accepted for presentation at the Ninth Annual Symposium on Principles of Programming Languages, January 1982.

[Co78] Cook, S.A. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput. 7*, 1 (February 1978), 70-90.

[Cou76] Cousot, P. and R. Cousot. Static determination of dynamic properties of programs. Proc. 2d Int. Symp. on Programming, Paris, April, 1976.

[Di76] Dijkstra, E.W. *A Discipline of Programming.* Prentice-Hall, 1976.

[Ha72] Habermann, A.N. Synchronization of communicating processes. *CACM 15*, 3 (1972), 171-176.

[Ho69] Hoare, C.A.R. An axiomatic approach to computer programming. *CACM 12*, 10 (1969), 322-329.

[Ho72] Hoare, C.A.R. Towards a theory of parallel programming. In Hoare and Perrot, eds., *Operating Systems Techniques.* Academic Press, 1972.

[Ko76] Kopetz, H. *Software Reliability.* Springer-Verlag Inc., 1976.

[Li77] Lipton, R.J. A necessary and sufficient condition for the existence of Hoare logics. 18th IEEE Symposium on the Foundations of Computer Science, October, 1977, pp. 1-6.

[Ow76] Owicki, S.D. and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *CACM 19*, 5 (1976), 279-289.

[Si76] Sintzoff, M. and A. Van Lamsveerde. Formal derivation of strongly correct parallel programs. MBLE Research Report, Brussels, 1976.