mechanism for specifying scheduling rules in a provable way.

Acknowledgments. I thank Ralph London for suggesting the use of history variables, and the referees and C.A.R. Hoare for their thoughtful and helpful comments on earlier drafts of this paper.

References

1. Dijkstra, E.W. Hierarchical ordering of sequential processes. In *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrott (Eds.), Academic Press, 1972, pp. 72–93.
2. Brinch Hansen, P. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, N.J., 1973.
3. Hoare, C.A.R. Monitors: an operating system structuring concept. *Comm. ACM 17*, 10 (Oct. 1974), 549–557. Corrigendum, *Comm. ACM 18*, 2 (Feb. 1975), 95.
4. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576–580, 583.
5. Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica 1* (1972), 271–281.
6. Robinson, L., Levitt, K.N., Neumann, P.G., and Saxena, A.R. On attaining reliable software for a secure operating system. 1975 International Conf. on Reliable Software, Los Angeles, pp. 267–284.
7. Clint, M. Program proving: coroutines. *Acta Informatica 2* (1973), 50–63.
8. Hopcroft, J.E., and Ullman, J.D. *Formal Languages and their Relation to Automata.* Addison-Wesley, Reading, Mass., 1969.
9. Howard, J.H., and Alexander, W.P. Analyzing sequences of operations performed by programs. *Program Test Methods*, W.C. Hetzel (Ed.), Prentice-Hall, 1973, pp. 239–254.
10. Alexander, W.P. Analysis of sequencing in computer programs and systems. Ph.D. Diss., U. of Texas at Austin, 1974.
11. Zilles, S.N. Abstract specification for data types. Working paper, IBM Research, San Jose, Calif., 1975. Extracted from 1974 Project Mac Progress Report, MIT, Cambridge, Mass., 1975.
12. Liskov, B.H., and Zilles, S.N. Specification techniques for data abstractions. 1975 International Conf. on Reliable Software, Los Angeles, pp. 72–87.
13. Parnas, D.L. A technique for software module specification with examples. *Comm. ACM 15*, 5 (May 1972), 330–336.
14. Habermann, A.N. Synchronization of communicating processes. Third Symp. on Operating Systems Principles, Palo Alto, 1971, pp. 80–85.
15. Sintzoff, M., and van Lamsweerde, A. Constructing correct and efficient concurrent programs. 1975 International Conf. on Reliable Software, Los Angeles, pp. 319–326.
16. Teory, T.J., and Pinkerton, T.B. A comparative analysis of disk scheduling policies. Third Symp. on Operating Systems Principles, Palo Alto, 1971, pp. 114–121.

Operating Systems — R.S. Gaines, Editor

Verifying Properties of Parallel Programs: An Axiomatic Approach

Susan Owicki and David Gries
Cornell University

An axiomatic method for proving a number of properties of parallel programs is presented. Hoare has given a set of axioms for partial correctness, but they are not strong enough in most cases. This paper defines a more powerful deductive system which is in some sense complete for partial correctness. A crucial axiom provides for the use of auxiliary variables, which are added to a parallel program as an aid to proving it correct. The information in a partial correctness proof can be used to prove such properties as mutual exclusion, freedom from deadlock, and program termination. Techniques for verifying these properties are presented and illustrated by application to the dining philosophers problem.

Key Words and Phrases: structured multiprogramming, correctness proofs, program verification, concurrent processes, synchronization, mutual exclusion, deadlock

CR Categories: 4.32, 4.35, 5.21, 5.24

A version of this paper was presented at the Fifth ACM Symposium on Operating Systems Principles, The University of Texas at Austin, November 19–21, 1975.

This research was partially supported by National Science Foundation grant GJ-42512. The authors are grateful for the possibility of presenting and discussing these ideas in a preliminary stage at the IFIP WG2.3 (on programming methodology) meeting in December 1974. Authors' address: Computer Science Department, Cornell University, Ithaca, NY 14853.

1. Introduction

The importance of correctness proofs for sequential programs is widely recognized; with parallel programs the need is even greater. When several processes are executed in parallel, their results can depend on the unpredictable order in which actions from different processes are executed. Such complexity greatly increases the probability that the programmer will make mistakes. Even worse, the mistakes may not be detected during program testing, since the particular interactions in which the errors are visible may not occur. It is important to structure parallel programs in a way which eliminates some of this complexity, and to verify their correctness with proofs as well as by program testing.

The techniques for program proofs given here are based on Hoare's syntax and axioms for parallel programs [6]. We find Hoare's language attractive because it restricts the interactions between parallel processes in a way which leads to intellectually manageable programs. His axiomatic method gives a sound basis for formal program proofs, but it also can be used informally and is more reliable than most informal methods.

But Hoare's axioms for parallel programs have certain weaknesses. They are intended only for proofs of partial correctness (a program is partially correct if it either produces the desired results or fails to terminate), and there are many other correctness criteria for parallel programs. Also, they are too weak to prove even partial correctness for many simple programs. In this paper we present a stronger set of axioms which greatly increase the power of the deductive system, and are in some sense complete. We also show how to apply axiomatic techniques to some other properties of parallel programs: mutual exclusion, blocking, and termination. Our intent is to describe the proof methods in an informal manner, relying on the reader's intuitive understanding of program execution. A more thorough formal presentation can be found in [9].

2. The Language

The parallel programming language we use is derived from Algol 60. It contains the usual assignment, conditional, **while**, **for**, compound, and null statements, plus two statements which are designed for parallel processing. Parallel execution is initiated by a statement of the form

resource r_1(*variable list*), . . . , r_m(*variable list*):
  cobegin S_1 // . . . // S_n **coend**

Here a resource r_i is a set of logically connected shared variables, and $S_1 \dots S_n$ are statements to be executed in parallel, i.e. parallel processes. No assumption is made about the way parallel execution is implemented, or about the relative speeds of the parallel processes.

The second statement, called a critical section, provides for synchronization and protection of shared variables. A statement of the form:

with r **when** B **do** S

has the following interpretation: r is a resource, B is a Boolean expression, and S is a statement which uses the variables of r. When a process attempts to execute such a statement it is delayed until the condition B is true and r is not being used by another process. When the process has control of r and B is true, S is executed. Upon termination r is free for further use by other processes. When several processes are competing for a particular resource we make no assumptions about the order in which they receive it. Critical section statements can only appear inside parallel processes, and critical sections for the same resource cannot be nested.

Much of the complexity of parallel programs stems from the way processes can interfere with each other as they use shared variables. The critical section statement reduces these problems by guaranteeing that only one process at a time has access to the variables in a resource. The following syntax restrictions ensure that all variables which could cause conflict are protected by critical sections.

1. If variable x belongs to resource r, it cannot appear in a parallel process except in a critical section for r.
2. If variable x is changed in process S, it cannot appear in $S_j (i \neq j)$ unless it belongs to a resource.

These restrictions can easily be enforced by a compiler. They greatly reduce the complexity of parallel programs and their correctness proofs.

Even with these restrictions, the results of executing a parallel program still depend on the relative speeds of the parallel processes. We introduce the term computation to correspond to one particular instance of program execution. In most cases there are many different computations for a given parallel program, and each one may result in different values for the program variables. Since we are interested in intermediate stages in program execution as well as the final result, we allow computations which represent only partial execution of a program.

In general a parallel program may have any number of **cobegin** statements and resources. In the interests of clarity we will restrict our attention to simple programs with just one resource and **cobegin** statement. Our results are valid for more complex programs, but they are easier to state and prove for the restricted case.

3. The Axioms

The axioms defined by Hoare [4] give the meaning of program statements in terms of *assertions* about variables in the program. The notation {P}S{Q} ex-

Fig. 1. Assertions for {x = 0} *add1* {x = 2}.

```
{x=0}
add1: begin y := 0;  z := 0;
  {y=0 ∧ z=0 ∧ I(r)}
  resource r(x, y, z): cobegin
     {y=0}
     with r when true do
       {y=0 ∧ I(r)}
       begin x := x + 1;  y := 1 end
       {y=1 ∧ I(r)}
     {y=1}
   //
     {z=0}
     with r when true do
       {z=0 ∧ I(r)}
       begin x := x + 1;  z := 1 end
       {z=1 ∧ I(r)}
     {z=1}
  coend
  {y=1 ∧ z=1 ∧ I(r)}
end
{x=2}
I(r) = {x=y+z}
```

Fig. 2. The program *add2*.

```
add2: resource r(x): cobegin
          with r when true do x := x + 1
        //
          with r when true do x := x + 1
        coend
```

presses the partial correctness of statement S with respect to assertions P and Q: i.e. if P is true before executing S, and S halts, then Q is true after executing S. P is called the *precondition*, Q the *postcondition* of S.

Hoare [4, 6] gives a set of axioms and inference rules for formal proofs of partial correctness formulas. Since we are mainly concerned with parallel programs here, we will be informal about sequential statements, and provide formal rules only for parallel statements. We will rely on an intuitive understanding of sequential programs to write formulas like

$\{z=x^y\}$ **begin** $y := y + 1;\ \ z := x*z$ **end** $\{z=x^y\}$

The axioms for parallel programs require an assertion I(r), the invariant for resource r, which describes the "reasonable" states of the resource. I(r) must be true when parallel execution begins, and remains true at all times outside critical sections for r. The axioms for **cobegin** and **with-when** statements make use of this invariant.

Parallel Execution Axiom. If {P1} *S1* {Q1} and {P2} *S2* {Q2} and . . . {Pn} *Sn* {Qn} and no variable free in Pi or Qi is changed in *Sj* ($i \neq j$) and all variables in I(r) belong to resource r, then {P1 ∧ . . . ∧ Pn ∧ I(r)} **resource** r: **cobegin** *S1*// . . . //*Sn* **coend** {Q1 ∧ . . . ∧ Qn ∧ I(r)}.

Critical Section Axiom. If {I(r) ∧ P ∧ B} S {I(r) ∧ Q}, and I(r) is the invariant from the **cobegin** statement, and no variable free in P or Q is changed in another process, then {P} **with** r **when** B **do** S {Q}.

Note that we cannot assume that I(r) is still true after the critical section statement is finished, since another process may have control of the resource and I(r) may be (temporarily) false. These two axioms are similar to ones given by Hoare, but are more powerful because they allow a more flexible use of variables in assertions.

Figure 1 shows an example of an informal proof of partial correctness based on the axioms. Note that the preconditions and postconditions are set off by braces { } and interspersed with the program statements.

Suppose we have a proof of {P} S {Q}, and S' is a statement in program S. We will write pre(S') and post(S') to denote the preconditions and postconditions which appear with S' in the proof. Likewise, if r is a resource of S, we will write I(r) for the invariant used with r. These assertions will be used extensively in our proofs of mutual exclusion and other properties. Their value derives from the fact that in any computation which starts with P true,

1. pre(S') is true whenever S' is ready to execute;
2. post(S') is true whenever S' finishes;
3. I(r) is true whenever no critical section for r is being executed.

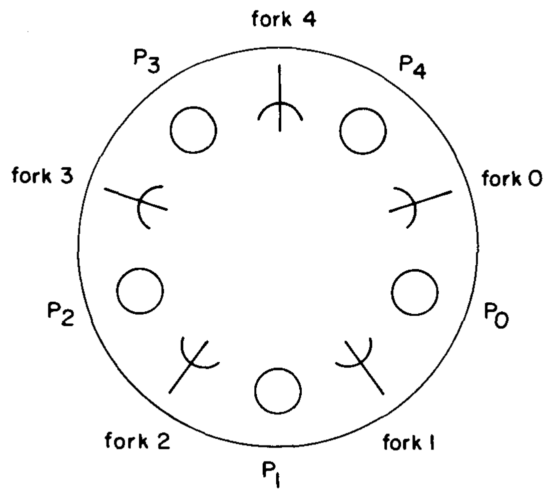
4. Auxiliary Variables

Unfortunately the axioms given above are inadequate for many simple programs. Figure 2 shows the program *add2*, for which {x = 0} *add2* {x = 2} is certainly true. However this cannot be proved using the axioms given so far: we cannot even prove that {0 ≤ x ≤ 2} is a valid invariant for resource r. Now consider the program *add1* in Figure 1. It has the same effect on x as *add2*, but it is possible to prove {x = 0} *add1* {x = 2} because of the extra variables y and z. The program *add1* has essentially the same behavior as *add2*, in spite of the fact that it contains statements and variables which do not appear in *add2*. This is because the additional variables, and the statements using them, do not affect the flow of control or the values assigned to x. Variables which are used in this way in a program will be called *auxiliary variables.* The need for auxiliary variables in proofs of parallel programs has been recognized by Brinch Hansen [1] and Lauer [7].

We would like to be able to conclude from the proof of {x = 0} *add1* {x = 2} that {x = 0} *add2* {x = 2} is also true. In order to do this we need an axiom which allows us to use auxiliary variables.

Definition. Let AV be a set of variables of program S such that $x \in \text{AV} \Rightarrow x$ appears in S only in assign-

Fig. 3. The dining philosophers.



ment statements of the form $x := E$ where any variable may be used in E. Then AV is an *auxiliary variable* set for S.

Auxiliary Variable Axiom. If AV is an auxiliary variable set for S, let S' be obtained from S by deleting all assignments to variables in AV (and possibly some redundant **begin end** brackets). Then if $\{P\}\ S\ \{Q\}$ is true and P and Q do not refer to any variables from AV, $\{P\}\ S'\ \{Q\}$ is also true. (The rules for deleting statements are defined more formally in [9]).

This axiom can be applied to $\{x = 0\}$ *add1* $\{x = 2\}$ to yield a proof of $\{x = 0\}$ *add2* $\{x = 2\}$. Auxiliary variables can be a very powerful aid in program proofs. Starting with a program such as *add2*, new variables and statements using them can be added to give a program like *add1* for which a proof is possible. Then the auxiliary variable axiom can be applied to yield a proof for the original program.

In [9] we show that the auxiliary variable rule makes the axioms for parallel programs given here complete in the following sense: any true formula $\{P\}\ S\ \{Q\}$ can be proved from these axioms, given sufficient knowledge about the data types of S. Thus the axioms capture all the information about program execution that is required for partial correctness proofs. Cook [2] gives a formal presentation of this kind of completeness for sequential programs.

5. Mutual Exclusion

Two statements are mutually exclusive if they cannot be executed at the same time. The critical section statement is designed to provide mutual exclusion for statements which operate on shared variables. However, there are times when the programmer must control the scheduling of resources directly, and must provide his own code for mutual exclusion. In such cases mutual exclusion can be verified using the assertions from a partial correctness proof.

As an example, consider a standard synchronization problem, the five dining philosophers. Five philosophers sit around a circular table (see Figure 3), alternately thinking and eating spaghetti. The spaghetti is so long and tangled that a philosopher needs two forks to eat it, but unfortunately there are only five forks on the table. The only forks which a philosopher can use are the ones to his immediate right and left. Obviously two neighbors cannot eat at the same time. The problem is to write a program for each philosopher to provide this synchronization. Hoare's solution [6] is given in Figure 4. The array *af*[*0:4*] indicates the number of forks available to each philosopher. In order to eat, a philosopher must wait until two forks are available; he then takes the forks and reduces the number available to each of his neighbors. Figure 5 shows some pre and post assertions for the dining philosophers program. Note the use of an auxiliary variable array, *eating*[*0:4*]. These assertions are derived from a formal proof, but they should also be intuitively valid.

Now we would like to use these assertions to prove that mutual exclusion is accomplished, i.e. that two neighbors do not get to eat at the same time. We will do this by assuming that it is possible for two neighbors to be eating at the same time and deriving a contradiction. Suppose program execution can take place in such a way that philosophers i and $i \oplus 1$ are ready to eat at the same time. At this point $eating[i] = 1$ and $eating[i \oplus 1] = 1$, from the preconditions for "*eat i*" and "*eat i* $\oplus$ *1*". If I(*forks*) is also true at this point we have the desired contradiction, for

$$(eating[i] = 1 \wedge eating[i \oplus 1] = 1 \wedge \mathrm{I}(forks)) \Rightarrow$$
$$(af[i] = 2 \wedge af[i] < 2) \Rightarrow \text{false}$$

Unfortunately I(*forks*) is not necessarily true, since some other philosopher may be in the midst of executing a critical section. Nevertheless, the following theorem shows that

$$(eating[i] = 1 \wedge eating[i \oplus 1] = 1 \wedge \mathrm{I}(forks)) \Rightarrow \text{false}$$

is a sufficient condition for guaranteeing that the required mutual exclusion holds.

THEOREM. *Suppose S_1 and S_2 are statements in different parallel processes of a program S, and neither S_1 nor S_2 belongs to a critical section for resource r. Let* P_1 *and* P_2 *be assertions such that*

pre $(S_1') \Rightarrow \mathrm{P}_1$ *for all statements S_1' in S_1,*
pre $(S_2') \Rightarrow \mathrm{P}_2$ *for all statements S_2' in S_2,*

where pre(S_1') *and* pre(S_2') *are derived from a proof of* $\{P\}$ S $\{Q\}$; *i.e.* P_i *is true throughout the execution of S_i. Then if*

$$(\mathrm{P}_1 \wedge \mathrm{P}_2 \wedge \mathrm{I}(r)) \Rightarrow \text{false},$$

S_1 and S_2 are mutually exclusive if P *is true when execution of S begins.*

Fig. 4. Dining philosophers program.

```
dining philosophers: begin
  comment af[i] is the number of forks available to philosopher i;
  af := 2;
  resource forks (af): cobegin DP0 // ... // DP4 coend
end

DPi: for j := 1 step 1 until Ni do
  begin
    getforks i: with forks when af[i] = 2 do
      begin af[i⊖1] := af[i⊖1] − 1;
            af[i⊕1] := af[i⊕1] − 1;
      end
    eat i: "eat";
    releaseforks i: with forks do
      begin af[i⊖1] := af[i⊖1] + 1;
            af[i⊕1] := af[i⊕1] + 1;
      end
    think i: "think";
  end
```

⊕ and ⊖ indicate arithmetic modulo 5

Fig. 5. Assertions for the dining philosophers.

```
{true}
dining philosophers: begin
  comment eating[i] is an auxiliary variable,
          eating[i] = 1 when philosopher i is eating, 0 otherwise;
  af := 2; eating := 0;
  {I(forks) ∧ eating[i] = 0, 0 ≤ i ≤ 4}
  resource forks(af, eating): cobegin DP0 // ... // DP4 coend
  {I(forks) ∧ eating[i] = 0, 0 ≤ i ≤ 4}
end

{eating[i] = 0}
DPi: for j := 1 step 1 until Ni do
  begin {eating[i] = 0}
      getforks i: with forks when af[i] = 2 do
        {eating[i] = 0 ∧ af[i] = 2 ∧ I(forks)}
        begin af[i⊖1] := af[i⊖1] − 1;  af[i⊕1] := af[i⊕1] − 1;
              eating[i] := 1
        end
        {eating[i] = 1 ∧ I(forks)};
      {eating[i] = 1}
      eat i: "eat";
      {eating[i] = 1}
      release forks i: with forks do
        {eating[i] = 1 ∧ I(forks)}
        begin af[i⊖1] := af[i⊖1] + 1;  af[i⊕1] := af[i⊕1] + 1;
              eating[i] := 0
        end
        {eating[i] = 0 ∧ I(forks)};
      {eating[i] = 0}
      think i: "think";
      {eating[i] = 0}
  end
{eating[i] = 0}
I(forks) = {[0 ≤ eating[i] ≤ 1 ∧ (eating[i] = 1 ⟹ af[i] = 2) ∧
           af[i] = 2 − (eating[i⊖1] + eating[i⊕1])] 0 ≤ i ≤ 4}
```

PROOF. Suppose not. If S_1 and S_2 are not mutually exclusive there is a computation C for S which starts with P true and reaches a point at which S_1 and S_2 are both in execution. P_1 and P_2 must be true after C, since they hold throughout execution of S_1 and S_2 respectively. Now if I(r) is also true after C we have a contradiction, since

$$P_1 \wedge P_2 \wedge I(r) \Rightarrow \text{false}.$$

But it is possible that some third process S' is in the midst of executing a critical section statement for r, so that I(r) does not hold after C. In this case there is another computation C′ for S which has S_1 and S_2 in execution and I(r) true.

To derive C′, let execution proceed as in C until the time when S' is ready to start the critical section mentioned above. In the original computation C, S' begins this statement but does not finish it. So from this point on in C, no process except S' makes any reference to the variables in resource r. C′ is obtained by stopping process S' at this point and allowing the other processes to continue exactly as before. Stopping S' does not affect the behavior of the other processes; because of the restrictions on shared variables, S' cannot change any variables used in other processes except those in resource r, and the other processes do not have access to r in the final part of the computation.

Now C′ still has S_1 and S_2 in execution, but no critical section for r is in execution. Then $P_1 \wedge P_2 \wedge I(r)$ holds after C′. Since this is impossible, the original assumption was wrong, and S_1 and S_2 are mutually exclusive. (A more formal proof of this theorem, based on a precise definition of "computation," is given in [9]).

Returning to the dining philosophers problem, we now can prove that two neighbors cannot eat at the same time. Let

S_1 = "*eat i*" S_2 = "*eat i ⊕ 1*"
P_1 = {eating[i] = 1} P_2 = {eating[i⊕1] = 1}

Since $(P_1 \wedge P_2 \wedge I(r)) \Rightarrow$ false, mutual exclusion is guaranteed.

6. Blocking

Another problem which is peculiar to parallel processes is the possibility that a program can be forced to stop before it has accomplished its purpose. This can happen in our parallel language because of the **with-when** statements. We say that a parallel process S_i is *blocked* if it is stopped at the statement **with** r **when** B **do** S because B is false or because another process is using the resource r. A program containing parallel processes is blocked if at least one of its processes is blocked, and the others are either finished or blocked.

In most cases process blocking is harmless: a process may be blocked and then unblocked many times during program execution. However if an entire program is blocked there is no way to recover. In this section we describe a way of proving this cannot occur in a given program, i.e. that the program is deadlock-free. Once again the method is based on assertions obtained from a partial-correctness proof.

THEOREM. *Suppose program S contains the statement*

$S' =$ **resource** r; **cobegin** $S_1 // \ldots // S_n$ **coend**.

Let the **with-when** *statements of process* S_k *be*

$S_k{}^j =$ **with** $r_k{}^j$ **when** $B_k{}^j$ **do** $T_k{}^j$, $1 \leq j \leq n_k$.

Let pre$(S_k{}^j)$, *and* I(r) *be assertions derived from a proof of* $\{P\}\ S\ \{Q\}$. *Let*

$$D_1 = \bigwedge_k (\text{post}(S_k) \vee (\bigvee_j(\neg B_k{}^j \wedge \text{pre}(S_k{}^j))))$$

$$D_2 = \bigvee_k \bigvee_j(\neg B_k{}^j \wedge \text{pre}(S_k{}^j))$$

Then if $D_1 \wedge D_2 \wedge I(r) \Rightarrow$ false, *S cannot be blocked if* P *is true when execution begins.*

PROOF. Suppose S is blocked for some computation C which starts with P true. Since S can only be blocked at **with-when** statements in S', C has begun parallel execution of the S_i. For each process S_i, either C has finished S_i or S_i is blocked at one of the $S_i{}^j$. In either case, no critical sections are in execution, so I(r) holds. Also, if C has finished S_i, post(S_i) holds, and if S_i is blocked at $S_i{}^j$, pre$(S_i{}^j) \wedge \neg B_i{}^j$ holds ($S_i{}^j$ must be blocked because $B_i{}^j$ is false, since no critical sections are in execution). Thus D_1 must hold after C. Since at least one of the S_i is blocked, D_2 must hold after C. This means that $D_1 \wedge D_2 \wedge I(r)$ holds after C, but this is impossible since $D_1 \wedge D_2 \wedge I(r) \Rightarrow$ false. So no such C exists, and S cannot be blocked.

Applying this theorem to the dining philosophers problem we have

$$\begin{aligned} D_1 &= \bigwedge_i [\text{post}(DPi) \vee (\text{pre}(\textit{getforks } i) \wedge af[i] \neq 2) \vee \\ &\qquad (\text{pre}(\textit{releaseforks } i) \wedge \neg \textit{true})] \\ &= \bigwedge_i (\textit{eating}[i] = 0 \vee (\textit{eating}[i] = 0 \wedge af[i] \neq 2)) \\ &\Rightarrow \bigwedge_i \textit{eating}[i] = 0 \end{aligned}$$

$$D_1 \wedge I(\textit{forks}) \Rightarrow \forall i\ (af[i] = 2)$$

$$\begin{aligned} D_2 &= \bigvee_i [\text{pre}(\textit{getforks } i) \wedge af[i] \neq 2) \vee \\ &\qquad (\text{pre}(\textit{releaseforks } i) \wedge \neg \textit{true})] \\ &\Rightarrow \exists i (af[i] \neq 2) \end{aligned}$$

So $D_1 \wedge D_2 \wedge I(\textit{forks}) \Rightarrow$ false, and the dining philosophers program cannot be blocked.

7. Termination

Program termination is an important property for both parallel and sequential programs, although there are correct parallel programs which do not terminate. Various techniques have been suggested for proving termination of sequential programs (Hoare [4], Manna [8]), and the same methods can often be applied to parallel programs. A sequential program can fail to terminate for two reasons: an infinite loop or the execution of an illegal operation such as dividing by zero. With parallel programs there is an additional possibility: the program can be blocked. (It is even possible that a program can be blocked for one computation and loop infinitely for another.) But if a program cannot be blocked, termination can be proved just as it would be for a sequential program.

One approach to proving termination is to show that each statement terminates provided that its components terminate. We will not attempt to present general rules for doing this, but will give sufficient conditions for proving that a parallel statement terminates.

Definition. A statement T *terminates conditionally* if it can be proved to terminate under the assumption that it does not become blocked.

THEOREM. *If T is a* **cobegin** *statement in a program S which cannot be blocked, T terminates if each of its parallel processes terminates conditionally.*

PROOF. Suppose T does not terminate. None of its processes can loop indefinitely, since they terminate conditionally. So after a finite time each one either finishes or is blocked. At that point S is blocked. Since this is impossible, T must terminate.

As an example, consider once again the dining philosophers program in Figure 5. We have already proved that it cannot be blocked, so we need only show that each philosopher process terminates conditionally. Assuming that the operations involved in "eating" do not stop execution, philosopher i must either become blocked or perform Ni iterations of the loop and terminate. So the process terminates conditionally. The termination theorem implies that the **cobegin** statement, and thus the whole dining philosophers program, must terminate.

8. Conclusion

The theorems and examples presented here have showed how to prove various properties of parallel programs. These techniques have been applied successfully to a number of standard problems from the parallel programming literature, e.g. readers and writers, communication via a bounded buffer, etc. They can also be modified to apply to programs which use other synchronization operations (e.g. semaphores, events) instead of **with-when** (see [9] for a theoretical discussion, and [3] for an application of these techniques to the verification of a concurrent garbage collector). However the proof process becomes much longer in languages which do not restrict the use of shared variables.

There are many important correctness properties for parallel programs besides the ones treated here; priority assignments, progress for each process, blocking of some subset of the processes in a program, etc. Many of these properties are difficult to define in a uniform way, while others require a language in which there are definite rules for scheduling competing processes. We are working to broaden the range of properties which can be proved with axiomatic methods.

The proof techniques we have discussed can be profitably applied at three levels. First, they provide a sound basis for formal proofs of program correctness. Although formal proofs are generally too long to be reasonably done by hand, the axiomatic method would be well suited for an interactive program verifier, in which the programmer provides the resource invariants and some of the pre and post assertions, and the program verifier checks that these satisfy the axioms.

A second possibility is informal proofs, like the ones given in this paper. The techniques are easy to use, and are relatively reliable. Although mistakes are possible in any informal proof, the structure of the axioms reduces the probability of error. Once the programmer has defined his resource invariants, the reasoning involved in the proofs is strictly sequential, and thus easy to do. In contrast, many informal proofs involve arguments about the order in which statements can be executed—in these it is dangerously easy to overlook the one case in which the program performs incorrectly.

Finally, the language and the axioms give guides for the construction of correct and comprehensible programs. The use of resources isolates the areas in which programs can interfere with each other, and the resource invariant states explicitly what each process can assume about the variables it shares with other processes. The programmer who takes the time to define a resource invariant and check that it is preserved in each critical section is using a valuable tool for producing correct programs.

References
1. Brinch Hansen, P. Concurrent programming concepts. *Computing Surveys 5*, 4 (Dec. 1973), 223–245.
2. Cook, S. A. Axiomatic and interpretive semantics for an Algol fragment. Tech. Rep. 79, Dep. of Computer Sci., U. of Toronto, 1975.
3. Gries, D. An exercise in proving properties of parallel programs. Lecture notes, Technical U. Munich.
4. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576–580.
5. Hoare, C.A.R. Monitors: an operating system structuring concept. *Comm. ACM 17*, 10 (Oct. 1974), 548–557.
6. Hoare, C.A.R. Towards a theory of parallel programming. In *Operating Systems Techniques*, Hoare and Perott (Eds.), Academic Press, New York, 1972.
7. Lauer, H.C. Correctness in operating systems. Ph.D. Th., Carnegie-Mellon U., 1973.
8. Manna, Z. and Pnueli, A. Axiomatic approach to total correctness of programs. *Acta Informatica 3* (1974), 243–263.
9. Owicki, S. Axiomatic proof techniques for parallel programs. Ph.D. Th., Cornell U., 1975.

Operating Systems — R.S. Gaines, Editor

Characteristics of Program Localities

A. Wayne Madison and Alan P. Batson
University of Virginia

The term "locality" has been used to denote that subset of a program's segments which are referenced during a particular phase of its execution. A program's behavior can be characterized in terms of its residence in localities of various sizes and lifetimes, and the transitions between these localities. In this paper the concept of a locality is made more explicit through a formal definition of what constitutes a phase of localized reference behavior, and by a corresponding mechanism for the detection of localities in actual reference strings. This definition provides for the existence of a hierarchy of localities at any given time, and the reasonableness of the definition is supported by examples taken from actual programs. Empirical data from a sample of production Algol 60 programs is used to display distributions of locality sizes and lifetimes, and these results are discussed in terms of their implications for the modeling of program behavior and memory management in virtual memory systems.

Key Words and Phrases: program behavior, memory management, locality

CR Categories: 4.22, 4.35, 4.6, 4.9, 6.21

A version of this paper was presented at the Fifth ACM Symposium on Operating Systems Principles, The University of Texas at Austin, November 19–21, 1975.

This research was supported by NSF Grant GJ-1005. Authors' address: Department of Applied Mathematics and Computer Science, University of Virginia, Charlottesville, VA 22901.