Acta Informatica 6, 319—340 (1976)

An Axiomatic Proof Technique for Parallel Programs I *

Susan Owicki and David Gries

Received November 6, 1975

Summary. A language for parallel programming, with a primitive construct for synchronization and mutual exclusion, is presented. Hoare's deductive system for proving partial correctness of sequential programs is extended to include the parallelism described by the language. The proof method lends insight into how one should understand and present parallel programs. Examples are given using several of the standard problems in the literature. Methods for proving termination and the absence of deadlock are also given.

1. Introduction

The importance of correctness proofs for sequential programs has long been recognized. Advocates of structured programming have argued that a well structured program should be easy to prove correct, and that programs should be written with a correctness proof in mind. In this connection, Hoare's deductive system [9], using axioms, inference rules and assertions, has been the most influential. Not only has Hoare shown us how to prove programs correct, his deductive system has shown us how to understand programs in an informal manner, and has given us insight into how to write better programs.

The need for correctness proofs for parallel programs is even greater. When several processes can be executed in parallel, the results can depend on the unpredictable order in which actions from different processes are executed, resulting in a complexity too great to handle informally. Even worse, program testing will rarely uncover all mistakes since the particular interactions in which errors are visible may not occur. A proof method is required which teaches us how to handle parallelism in a simple, understandable manner.

A number of methods have been used in proofs for parallel programs. The most common is reliance on informal arguments—a risky business given the complexity of parallel program interactions. More formal approaches have included application of Scott's mathematical semantics (Cadiou and Levy [3]), Lipton's reduction method [14], and Rosen's Church-Rosser approach [17].

This paper, based on the PhD thesis of the first author, extends Hoare's attempt [10] to include parallelism in this deductive system. We feel it is intuitive enough to be used as a basis for reliable proof outlines, and it has given us insight into how to understand parallel programs. Other approaches related to our work are contained in Ashcroft and Manna [1], Ashcroft [2], Lauer [12] and Newton [15].

* This research was partially supported by National Science Foundation grant GJ-42512.

Any parallel programming language must contain statements for describing cooperation between processes — synchronization, mutual exclusion, and the like. We provide a flexible but primitive tool, so primitive that other methods for synchronization such as semaphores and events can be easily described using it. This means that the deductive system can be used to prove correctness of programs using other methods as well. It can also be used to prove correctness for programs of such a fine degree of interleaving that the only mutual exclusion need be the memory reference. This has been done for Dijkstra's on-the-fly garbage collector [6], with fairly good results given the complexity of this algorithm, in [7].

The paper is organized as follows. In section 2 we describe Hoare's work briefly. In Section 3 we introduce the parallel language and extend his system to include it. In Section 4 we give several examples of proofs of partial correctness, while in Section 5 we show how to describe semaphores in the language and give examples. Sections 6 and 7 are devoted to discussions of proofs of other important properties of parallel programs: the absence of deadlock and termination. We summarize our work in Section 8.

Thanks go to Charles Moore for many valuable discussions about parallel processing, and also to Robert Constable and Marvin Solomon. We are grateful to the members of IFIP working group 2.3 on programming methodology, especially to Tony Hoare and Edsger W. Dijkstra, for the opportunity to present and discuss this material in its various stages at working group meetings. The observation that the memory reference must have "reasonable" properties, as discussed in Section 3, was made by John Reynolds.

2. Proofs of Properties of Sequential Programs

Let P and Q be assertions about variables and S a statement. Informally, the notation

$$\{P\}\; S\; \{Q\}$$

means: if P is true before execution of S, then Q is true after execution of S. Nothing is said of termination; Q holds *provided* S terminates. The notation

$$\frac{a}{b}$$

means: if a is true, then b is also true. Using such notation, Hoare [9] describes a deductive system for proving properties of sequential programs. Let P, P_i represent assertions, x a variable, E an expression, B a Boolean expression and S, S_i statements, then the axioms for the five kinds of statements allowed are:

(2.1) null $\{P\}$ **skip** $\{P\}$

(2.2) assignment $\{P_E^x\}\; x := E\; \{P\}$ where P_E^x is the assertion formed by replacing every occurrence of x in P by E.

(2.3) alternation $$\frac{\{P \wedge B\}\; S_1\; \{Q\},\; \{P \wedge \neg B\}\; S_2\; \{Q\}}{\{P\}\; \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\; \{Q\}}$$

(2.4) iteration
$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{P \wedge \neg B\}}$$

(2.5) composition
$$\frac{\{P_1\}\ S_1\ \{P_2\},\ \{P_2\}\ S_2\ \{P_3\},\ \ldots,\ \{P_n\}\ S_n\ \{P_{n+1}\}}{\{P_1\}\ \textbf{begin}\ S_1;\ S_2;\ \ldots;\ S_n\ \textbf{end}\ \{P_{n+1}\}}$$

In addition, we have the following rule of consequence:

(2.6) consequence
$$\frac{\{P_1\}\ S\ \{Q_1\},\ P \vdash P_1,\ Q_1 \vdash Q}{\{P\}\ S\ \{Q\}}$$

The notation $P \vdash Q$ means it is possible to prove Q using P as an assumption. The deductive system to be used in proving Q from P is not given; it could be any system which is valid for the data types and operations used in the programming language.

Note that declarations have been omitted, purely for the sake of simplicity. Hence all variable are globally defined. We also choose not to give the syntax of expressions or assertions. In general, we use an ALGOL-like syntax for expressions, while assertions will be given in a mixture of mathematical notation and English.

Now let us briefly discuss proofs of properties of sequential programs. When we write $\{P\}\ S\ \{Q\}$, this implies the existence of a proof of $\{P\}\ S\ \{Q\}$, using axioms (2.1)–(2.6). For example, suppose we have

$$S \equiv \textbf{begin}\ x := a;\ \textbf{if}\ e\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{end}$$

and suppose we already have proofs

$$\{P1 \wedge e\}\ S1\ \{Q1\} \quad \text{and} \quad \{P1 \wedge \neg e\}\ S2\ \{Q1\}.$$

Then a proof of $\{P\}\ S\ \{Q\}$ might be:

(2.7) (1) $\{P1^x_a\}\ x := a\ \{P1\}$ — assignment

(2) $$\frac{\{P1^x_a\}\ x := a\ \{P1\},\ P \vdash P1^x_a}{\{P\}\ x := a\ \{P1\}}$$ rule of consequence

(3) $$\frac{\{P1 \wedge e\}\ S1\ \{Q1\},\ \{P1 \wedge \neg e\}\ S2\ \{Q1\}}{\{P1\}\ \textbf{if}\ e\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q1\}}$$ alternation

(4) $$\frac{\{P1\}\ \textbf{if}\ e\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q1\},\ Q1 \vdash Q}{\{P1\}\ \textbf{if}\ e\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}$$ rule of consequence

(5) $$\frac{\{P\}\ x := a\ \{P1\},\ \{P1\}\ \textbf{if}\ e\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}{\{P\}\ \textbf{begin}\ x := a;\ \textbf{if}\ e\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{end}\ \{Q\}}$$ composition

This proof is made much more understandable by giving a *proof outline*, in which the program is given with assertions interleaved at appropriate places, as in (2.8). In such a proof outline, two adjacent assertions $\{P1\}$ $\{P2\}$ denote a use of the rule of consequence, where $P1 \vdash P2$.

(2.8)
```
{P}
begin {P}
      {P1^x_a}
      x := a;
      {P1}
      if e then {P1 ∧ e}
                S1
                {Q1}
           else {P1 ∧ ¬e}
                S2
                {Q1}
      {Q1}
      {Q}
end
{Q}
```

Most of our proofs will be presented in this style. If $P1 \vdash P2$ can be understood easily, we will sometimes only write $P1$, or $P2$. Thus, we might have written

(2.9) **begin** $\{P\}\ x := a; \{P1\} \ldots$

leaving out the assertion $\{P1^x_a\}$ in (2.8). However, each statement S is always preceded directly by one assertion, called its *precondition*, written *pre*(S). In (2.8), *pre*$(x{:=}a) = P1^x_a$ while in (2.9) *pre*$(x{:=}a) = P$. This notion of a precondition of a statement is important for our work. Similarly, the *postcondition post*(S) is the assertion following statement S.

We may also leave out assertions entirely for a sequence of assignments or simple conditionals, since the necessary weakest precondition of the sequence can always be derived from the postcondition — from the result assertion of the sequence. However, as we shall see, in the parallel case this can sometimes lead to our inability to develop a proof; this situation can sometimes be remedied by explicitly stating stronger preconditions. Proofs of correctness in the face of parallelism require much more care then the simple sequential case.

We will later discuss proofs of properties of parallel programs, such as termination and the absence of deadlock. These are actually properties of the *execution* of a program, and in order to discuss them we should introduce an operational model of the language and show that the deductive system is consistent with it. This has been done for the sequential system by Hoare and Lauer [11] and Cook [5], and for the parallel system by Owicki [16]. The systems have also been shown to be complete in a restricted sense by Cook [5] and Owicki [16]; informally this means that every program you would expect to be able to prove partially correct, can indeed be proved in this system.

We will not introduce an operational model here, but will rely on the reader's knowledge that this can be done and his knowledge about execution of programs. We should however discuss assertions somewhat.

An assertion P is a Boolean function defined over the possible values of all the variables of the program. Let the state m of the machine denote the set of values of all variables at any moment during execution. By the phrase "P is true at that moment", we mean that $P[m]=$ **true**. By $P=$ **true** we mean that $P[m]=$ **true** for all possible states m.

Our informal proof outlines and proofs of properties of execution rely on the following property, which must be true if the deductive system is to be consistent with the operation model:

(2.10) Let S be a statement in a program T, and $pre(S)$ the precondition of S in a proof outline of $\{P\}\ T\ \{Q\}$. Suppose execution of T begins with P true and reaches a point where S is about to begin execution, with the variables in state m. Then $pre(S)[m]=$ **true**.

3. Proof of Correctness of Parallel Programs

We introduce parallelism by extending the sequential language with two new statements—one to initiate parallel processing, the other to coordinate processes to be executed in parallel.

Let $S1, S2, \ldots, Sn$ be statements. Then execution of the **cobegin** statement

cobegin $S1\ //\ S2\ //\ \ldots\ //\ Sn$ **coend**

causes the statements Si to be executed in parallel. Execution of the **cobegin** statement terminates when execution of all of the processes Si have terminated. There are no restrictions on the way in which parallel execution is implemented; in particular, nothing is assumed about the relative speeds of the processes.

We do require that each assignment statement and each expression be executed or evaluated as an individual, indivisible action. However this restriction can be lifted if programs adhere to the following simple convention (which we follow in this paper):

(3.1) Each expression E may refer to at most one variable y which can be changed by another process while E is being evaluated, and E may refer to y at most once. A similar restriction holds for assignment statements $x := E$.

With this convention, the only indivisible action need be the memory reference. That is, suppose process Si references variable (location) c while a different process Sj is changing c. We require that the value received by Si for c be the value of c either before or after the assignment to c, but it may not be some spurious value caused by the fluctuation of the value of c during assignment. Thus, our parallel language can be used to model parallel execution on any reasonable machine.

The second statement has the form

await B **then** S

where B is a Boolean expression and S a statement not containing a **cobegin** or another **await** statement. When a process attempts to execute an **await**, it is delayed until the condition B is true. Then the statement S is executed as an indivisible action. Upon termination of S, parallel processing continues. If two or more processes are waiting for the same condition B, any one of them may be allowed to proceed when B becomes true, while the others continue waiting. In some applications it is necessary to specify the order in which waiting processes are scheduled, but for our purposes any scheduling rule is acceptable. Note that evaluation of B is part of the indivisible action of the **await** statement; another process may not change variables so as to make B false after B has been evaluated but before S begins execution.

The **await** statement can be used to turn any statement S into an indivisible action:

await true then S

or it may be used purely as a means of synchronization:

await "some condition" **then skip**

Note that the **await** is not proposed as a new synchronization statement to be inserted in the next programming language; it is too powerful to be implemented efficiently. Rather, it is provided as a means of representing a number of standard synchronization primitives such as semaphores. Thus to verify a program which uses semaphores, one first expresses the semaphore operations as **await**s, and then applies the techniques given here.

We now turn to formal definitions of these statements, in (3.2) and (3.3). The definition of the **await** is straightforward, but (3.3) will require an explanation, along with a definition of "interference-free":

(3.2) **await** $$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \textbf{await}\ B\ \textbf{then}\ S\ \{Q\}}$$

(3.3) **cobegin** $$\frac{\{P1\}\ S1\ \{Q1\}, \ldots, \{Pn\}\ Sn\ \{Qn\} \text{ are interference-free}}{\{P1 \wedge \ldots \wedge Pn\}\ \textbf{cobegin}\ S1\ /\!/ \ldots /\!/\ Sn\ \textbf{coend}\ \{Q1 \wedge \ldots \wedge Qn\}}$$

Definition (3.3) says that the effect of executing $S1, \ldots, Sn$ in parallel is the same as executing each one by itself, provided the processes don't "interfere" with each other. The key word is of course "interfere". One possibility to obtain non-interference is not to allow shared variables, but this is too restrictive. A more useful rule is to require that certain assertions used in the proof $\{Pi\}$ Si $\{Qi\}$ of each process are left invariantly true under parallel execution of the other processes. For if these assertions are not falsified, then the proof $\{Pi\}$ Si $\{Qi\}$ will still hold and consequently Qi will still be true upon termination! For example, the assertion $\{x \geqq y\}$ remains true under execution of $x := x+1$, while the assertion $\{x=y\}$ does not. The invariance of an assertion P under execution of a statement S is explained by the formula

$$\{P \wedge pre(S)\}\ S\ \{P\}$$

We now give the definition of "interference-free".

(3.4) **Definition.** Given a proof $\{P\}$ S $\{Q\}$ and a statement T with precondition $pre(T)$, we say that T *does not interfere with* $\{P\}$ S $\{Q\}$ if the following two conditions hold:

(a) $\{Q \wedge pre(T)\}\ T\ \{Q\}$,

(b) Let S' be any statement within S but not within an **await**. Then $\{pre(S') \wedge pre(T)\}\ T\ \{pre(S')\}$.

(3.5) **Definition.** $\{P1\}\ S1\ \{Q1\}, \ldots, \{Pn\}\ Sn\ \{Qn\}$ are ***interference-free*** if the following holds. Let T be an **await** or assignment statement (which does not appear in an **await**) of process Si. Then for all $j, j \neq i$, T does not interfere with $\{Pj\}\ Sj\ \{Qj\}$.

We will from time to time make program transformations which obviously don't affect correctness, such as replacing **begin** S **end** by S, and replacing **await true then** $x := E$ by $x := E$ provided the assignment satisfies (3.1). One transformation that is necessary in proving correctness of parallel programs is the addition (or deletion) of assignments to so-called *auxiliary variables*. These auxiliary variables are needed only for the proof of correctness and other properties, and not in the program itself. Typically, they record the history of execution or indicate which part of a program is currently executing. The need for such variables has been independently recognized by many; the first reference we have found to them is Clint [4]. We define:

(3.6) **Definition.** Let AV be a set of variables which appear in S only in assignments $x := E$, where x is in AV. Then AV is an ***auxiliary variable set*** for S.

(3.7) *Auxiliary variable transformation*: Let AV be an auxiliary variable set for S', and P and Q assertions which do not contain free variables from AV. Let S be obtained from S' by deleting all assignments to the variables in AV. Then

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

We shall give examples of the use of the deductive system (2.1)—(2.6), (3.2), (3.3), (3.7) in the next section. But first let us discuss it. Rule (3.3) teaches us to understand parallel processes in two steps. First, understand each process Si, that is study its proof, as an independent, sequential program, disregarding parallel execution completely. *Then* show that execution of each other process does not interfere with the *proof* of Si.

The conventional way of showing non-interference has been to see whether execution of a process Sj interferes with the *execution* of Si. Thus we find phrases like "Suppose Sj does so and so, and then Si executes this and does that". This interleaving of two dynamic objects, the execution of Si and Sj, is very difficult if not impossible to understand for many parallel processes, and it is too easy to miss an argument somewhere.

By concentrating on whether Sj can affect the *proof* of Si's correctness, we turn our attention to a static object which is easier to deal with. Showing non-interference is quite mechanical; make up a list of Si's preconditions, a second list of Sj's assignments and **await**s, and show that each element of the second list does not disturb the truth of each assertion in the first.

If a statement T of Sj does interfere with a precondition P of Si, then either the program is incorrect or else Si's proof is inadequate. Often the proof $\{Pi\}\ Si$ $\{Qi\}$ can be adjusted—assertions can be weakened, keeping the proof still valid, until Sj no longer interferes with them. In any case, the possibility of the programmer missing a particular case is quite low as long as he is careful and persists; this is not the case with earlier informal reasoning.

4. Examples of Proof Outlines of Partial Correctness

Example 1. A proof outline for a very simple program is given in (4.1). It is obvious that the program "works", as long as $S1$ and $S2$ are interference-free. This requires verification of 4 formulas:

1. $\{pre(S1) \wedge pre(S2)\}\ S2\ \{pre(S1)\}$:

```
    {(x=0 ∨ x=2) ∧ (x=0 ∨ x=1)}
    {x=0}
    await true then {x=0}
                    x := x+2
                    {x=2}
    {x=2}
    {x=0 ∨ x=2}
```

2. $\{Q1 \wedge pre(S2)\}\ S2\ \{Q1\}$ (verification left to the reader)
3. $\{pre(S2) \wedge pre(S1)\}\ S1\ \{pre(S2)\}$ (left to the reader)
4. $\{Q2 \wedge pre(S1)\}\ S1\ \{Q2\}$ (left to the reader)

```
(4.1) {x=0}
      S: cobegin {x=0}
                 {x=0 ∨ x=2}
                 S1: await true then x := x+1
                 {Q1: x=1 ∨ x=3}
          //
                 {x=0}
                 {x=0 ∨ x=1}
                 S2: await true then x := x+2
                 {Q2: x=2 ∨ x=3}
         coend
      {(x=1 ∨ x=3) ∧ (x=2 ∨ x=3)}
      {x=3}
```

Suppose we replace $S1$ by the single assignment statement $x := x+1$. Then the program does not follow convention (3.1). Hence the proof method could not be used to prove this program correct for execution in an environment where the grain of interleaving is finer than the assignment statement. In fact, execution of the program (with this change) could result in the value 2 or 3 for x.

Example 2. Consider the more realistic problem of finding the first component $x(k)$ of an array $x(1:M)$, if there is one, which is greater than zero. Program *Findpos* (4.2), given by Rosen [17], does this using two parallel processes to check the even and odd subscripted array elements separately. In (4.3) we present a proof outline, except for the interference-free check. Note that *Findpos* uses no **await** statement.

(4.2) *Findpos*: **begin**

```
initialize: i := 2; j := 1; eventop := M+1; oddtop := M+1;
search: cobegin
          Evensearch: while i < min(oddtop, eventop) do
                          if x(i) > 0 then eventop := i
                                      else i := i+2
          //
          Oddsearch:  while j < min(oddtop, eventop) do
                          if x(j) > 0 then oddtop := j
                                      else j := j+2
        coend;
k := min(eventop, oddtop)
end
```

(4.3) $\{ES \wedge OS\}$

```
search: cobegin {ES}
            Evensearch: while i < min(oddtop, eventop) do
                {ES ∧ i < eventop ∧ i < M+1}
                if x(i) > 0
                then {ES ∧ i < M+1 ∧ x(i) > 0 ∧ i < eventop}
                     eventop := i
                     {ES}
                else {ES ∧ i < eventop ∧ x(i) ≦ 0}
                     i := i+2
                     {ES}
                {ES}
            {ES ∧ i ≧ min(oddtop, eventop)}
        //
            {OS}
            Oddsearch: while j < min(oddtop, eventop) do
                {OS ∧ i < oddtop ∧ j < M+1}
                if x(j) > 0
                then {OS ∧ j < M+1 ∧ x(j) > 0 ∧ j < oddtop}
                     oddtop := j
                     {OS}
                else {OS ∧ j < oddtop ∧ x(j) ≦ 0}
                     j := j+2
                     {OS}
                {OS}
            {OS ∧ j ≧ min(oddtop, eventop)}
    coend
{OS ∧ ES ∧ i ≧ min(oddtop, eventop) ∧ j ≧ min(oddtop, eventop)}
k := min(oddtop, eventop)
{k ≦ M+1 ∧ ∀l(0 < l < k ⇒ x(l) ≦ 0) ∧ (k ≦ M ⇒ x(k) > 0)}
```

$$\text{where } ES = \begin{cases} eventop \leqq M+1 \wedge \forall l((l \text{ even} \wedge 0<l<i) \Rightarrow x(l) \leqq 0) \wedge i \text{ even} \\ \wedge\ (eventop \leqq M \Rightarrow x(eventop) > 0) \end{cases}$$

$$OS = \begin{cases} oddtop \leqq M+1 \wedge \forall l((l \text{ odd} \wedge 0<l<j) \Rightarrow x(l) \leqq 0) \wedge j \text{ odd} \\ \wedge\ (oddtop \leqq M \Rightarrow x(oddtop) > 0) \end{cases}$$

While studying (4.3) do not worry about interaction between *Evensearch* and *Oddsearch*; look upon them as sequential, independent programs. To verify the interference-free property, we must show that each assignment in *Oddsearch* leaves invariantly true each precondition and the final assertion of *Evensearch*. (The argument that *Evensearch* does not interfere with *Oddsearch* is symmetric.) The only assignment in *Oddsearch* that changes a variable in one of *Evensearch*'s assertions is $oddtop := j$, and the only clause in *Evensearch*'s assertions which references *oddtop* is $i \geqq min(eventop, oddtop)$. Thus we must show that

(4.4) $\{i \geqq min(eventop, oddtop) \wedge pre(oddtop := j)\}$
$oddtop := j$
$\{i \geqq min(eventop, oddtop)\}$

Since $pre\ (oddtop := j) \Rightarrow j < oddtop$, (4.4) is certainly true. Thus, for this program, establishing the interference-free property was quite simple.

Example 3. We consider a standard problem from the literature of parallel programming. A producer process generates a stream of values for a consumer process. Since the producer and consumer proceed at a variable but roughly equal pace, it is profitable to interpose a buffer between the two processes, but since storage is limited, the buffer can only contain N values. The description of the buffer is:

(4.5) $buffer[0:N-1]$ is the shared buffer;
in = number of elements added to the buffer;
out = number of elements removed from the buffer;
the buffer contains *in-out* values. These are in order, in
$buffer[out \textbf{ mod } N], \ldots, buffer[(out+in-out-1) \textbf{ mod } N]$.

In (4.6) we show a solution to the problem in a general environment. In (4.7), we consider a program using this solution which copies an array of values $A\ [1:M]$ into an array $B\,[1:M]$. (4.8) gives a proof outline for the main program; (4.9) and (4.10) proof outlines for the separate processes. To show the interference-free property, first note that assertion I is invariant throughout both processes. The only assignment in the consumer which might invalidate an assertion of the producer is $out := out+1$. The only assertion of the producer which it could possibly invalidate is $in\text{-}out < N$, but clearly increasing *out* leaves this true. Hence the consumer does not interfere with the producer; similar reasoning shows that the producer does not interfere with the consumer.

(4.6) **begin comment** See (4.5) for description of buffer;
$in := 0$; $out := 0$;
cobegin producer: ...

```
                  await in-out < N then skip;
                  add: buffer (in mod N) := next value;
                  markin: in := in + 1;
                        ...
       //
                consumer: ...
                  await in-out > 0 then skip;
                  remove: this value := buffer [out mod N];
                  markout: out := out + 1;
                        ...
         coend
       end
```

(4.7)
```
fg1: begin comment See (4.5) for description of buffer;
  in := 0; out := 0; i := 1; j := 1;
  cobegin producer: while i ≦ M do
                      begin x := A [i];
                            await in-out < N then skip;
                            add: buffer [in mod N] := x;
                            markin: in := in + 1;
                            i := i + 1
                      end
       //
           consumer: while j ≦ M do
                      begin await in-out > 0 then skip;
                            remove: y := buffer [out mod N];
                            markout: out := out + 1;
                            B [j] := y;
                            j := j + 1
                      end
  coend
end
```

(4.8) Proof outline for $fg1$ (main program)

$\{M \geqq 0\}$

$fg1$: begin $in := 0;\ out := 0;\ i := 1;\ j := 1;$

$\{I \wedge i = in + 1 = 1 \wedge j = out + 1 = 1\}$

$fg1'$: **cobegin**

$\{I \wedge i = in + 1 = 1\}$ *producer* $\{I \wedge i = in + 1 = M + 1\}$

$//\ \{I \wedge j = out + 1 = 1\}$ *consumer* $\{I \wedge (B[k] = A[k],\ 1 \leqq k \leqq M)\}$

coend

end

$\{B[k] = A[k],\ 1 \leqq k \leqq M\}$

$$\text{where } I = \left\{\begin{array}{l} buffer\,[(k-1) \mathbf{\ mod\ } N] = A[k],\ out < k \leqq in) \\ \wedge\ 0 \leqq in\text{-}out \leqq N \\ \wedge\ 1 \leqq i \leqq M + 1 \\ \wedge\ 1 \leqq j \leqq M + 1 \end{array}\right\}$$

(4.9) Proof outline for $fg1$ (*producer*). Invariant I is as in (4.8).

$\{I \wedge i=in+1\}$
producer: **while** $i \leqq M$ **do**
  begin $\{I \wedge i=in+1 \wedge i \leqq M\}$
    $x := A[i]$;
    $\{I \wedge i=in+1 \wedge i \leqq M \wedge x=A[i]\}$
    await $in\text{-}out < N$ **then skip**;
    $\{I \wedge i=in+1 \wedge i \leqq M \wedge x=A[i] \wedge in\text{-}out < N\}$
    add: $buffer[in \textbf{ mod } N] := x$;
    $\{I \wedge i=in+1 \wedge i \leqq M \wedge buffer[in \textbf{ mod } N]=A[i] \wedge in\text{-}out < N\}$
    markin: $in := in+1$;
    $\{I \wedge i=in \wedge i \leqq M\}$
    $i := i+1$
    $\{I \wedge i=in+1\}$
  end
$\{I \wedge i=in+1=M+1\}$

(4.10) Proof outline for $fg1$ (*consumer*). Invariant I is as in (4.8).

$\{I \wedge IC \wedge j=out+1\}$
consumer: **while** $j \leqq M$ **do**
  begin $\{I \wedge IC \wedge j=out+1 \wedge j \leqq M\}$
    await $in\text{-}out > 0$ **then skip**;
    $\{I \wedge IC \wedge j=out+1 \wedge j \leqq M \wedge in\text{-}out > 0\}$
    remove: $y := buffer[out \textbf{ mod } N]$;
    $\{I \wedge IC \wedge j=out+1 \wedge j \leqq M \wedge in\text{-}out > 0 \wedge y=A[j]\}$
    markout: $out := out+1$;
    $\{I \wedge IC \wedge j=out \wedge j \leqq M \wedge y=A[j]\}$
    $B[j] := y$;
    $\{I \wedge IC \wedge j=out \wedge j \leqq M \wedge B[j]=A[j]\}$
    $j := j+1$
    $\{I \wedge IC \wedge j=out+1 \wedge j \leqq M+1\}$
  end
$\{I \wedge IC \wedge j=out+1=M+1\}$
$\{I \wedge (B[k]=A[k], 1 \leqq k \leqq M)\}$
where $IC = \{B[k]=A[k], 1 \leqq k < j\}$

5. Implementing Semaphores

A semaphore *sem* is an integer variable which can only accessed by two operations, P and V.

$P(sem)$: if $sem \leqq 0$, $sem := sem - 1$; otherwise suspend the process until $sem > 0$.

$V(sem)$: $sem := sem + 1$.

The P and V operations are indivisible. They can be represented by synchronization statements as follows.

$P(sem)$: **await** $sem > 0$ **then** $sem := sem - 1$;
$V(sem)$: **await true then** $sem := sem + 1$

Semaphores, as first defined by Dijkstra [18] were slightly different:

$P'(sem)$: $sem := sem - 1$; if $sem < 0$ then the process is suspended on a queue associated with *sem*.

$V'(sem)$: $sem := sem + 1$; if $sem \leqq 0$, awaken one of the processes on the semaphore's queue.

A possible implementation of these operations uses a Boolean array *waiting*, with one element for each process. Initially $waiting[i] =$ **false**, and $waiting[i] =$ **true** implies that i is on the queue.

```
P'(sem): await true then
             begin sem := sem - 1;
                   if sem < 0 then waiting[this process] := true;
             end;
         await ¬waiting[this process] then skip
V'(sem): await true then
             begin sem := sem + 1;
                   if sem ≦ 0 then
                      begin choose i such that waiting[i];
                            waiting[i] := false
                      end
             end
```

In some cases the effects of the operations P and V are different from those of P' and V', but for the properties we discuss—partial correctness, absence of deadlock, and termination—these differences are irrelevant. See Lipton [13] for a comparison of the two kinds of semaphore operations. We leave it to the reader to define semaphores P'' and V'', like P' and V', except that the longest waiting process always gets served next.

Given a program with semaphores, the semaphore operations can be replaced by the corresponding **await**s. The result is an equivalent program which can be proved correct using the methods presented in this paper. A number of other synchronization primitives can also be modelled using **await**.

Consider a second version of the producer-consumer program, *fg*2 (5.1), which uses semaphores *full* and *empty* to synchronize access to the buffer. In (5.2) we show the translation of the semaphores into **await**s; (5.2) also uses auxiliary variables needed for a proof of partial correctness. In (5.3) we give a proof outline for the main program; in (5.4) the proof outline for the consumer (the producer is omitted, since it is similar). The proof is essentially the same as for the earlier version *fg*1 of the program. Using inference rule (3.7), the auxiliary variables can be removed to yield a proof of $\{M \geqq 0\}$ *fg*2 $\{B[k] = A[k],\ 1 \leqq k \leqq M\}$. The producer does not interfere with the proof of the consumer because the assertions in this proof include only I (which is invariantly true in both processes) and variables not changed by the producer. Likewise, the consumer does not interfere with the proof of the producer.

Habermann (8) presents this solution to the producer-consumer problem and provides an informal proof of correctness. He uses special functions which count the number of P and V operations on each semaphore; these play the same role as our auxiliary variables.

(5.1)
```
fg2: begin comment buffer[0:N-1] is the shared buffer,
                   full = number of full places in buffer (semaphore),
                   empty = number of empty places (semaphore);
     full := 0; empty := N; i := 1; j := 1;
     cobegin producer: while i ≦ M do
                         begin x := A[i];
                               P(empty);
                               buffer[i mod N] := x;
                               V(full);
                               i := i + 1
                         end
        //
             consumer: while j ≦ M do
                         begin P(full);
                               y := buffer[j mod N];
                               V(empty);
                               B[j] := y;
                               j := j + 1
                         end
     coend
    end
```

(5.2)
```
fg2': begin comment Pempty, Vempty, Pfull, Vfull are
                    auxiliary variables;
    full := 0; empty := N; i := 1; j := 1;
    Pfull, Vfull, Pempty, Vempty := 0, 0, 0, 0;
    cobegin producer: while i ≦ M do
            begin x := A[i];
                  await empty > 0 then
                    begin empty := empty - 1;
                          Pempty := Pempty + 1 end;
                  buffer[i mod N] := x;
                  await true then
                    begin full := full + 1; Vfull := Vfull + 1 end;
                  i := i + 1
            end
      //
          consumer: while j ≦ M do
            begin await full > 0 then
                    begin full := full - 1; Pfull := Pfull + 1 end;
                  y := buffer[j mod N];
                  await true then
                    begin empty := empty + 1;
                          Vempty := Vempty + 1 end;
                  B[j] := y;
                  j := j + 1
            end
    coend
end
```

(5.3) Proof outline of $fg2'$ (main program)

$fg2'$: **begin**

$full := 0;\ empty := N;\ i := 1;\ j := 1;$

$Pfull, Vfull, Pempty, Vempty := 0, 0, 0, 0;$

$\{I \wedge Vfull = Pempty \wedge i = Vfull + 1 \wedge Vempty = Pfull$

$\wedge\ j = Vempty + 1\}$

cobegin

$\{I \wedge Vfull = Pempty \wedge i = Vfull + 1\}$

producer

$\{I\}$

//

$\{I \wedge Vempty = Pfull \wedge j = Vempty + 1\}$

consumer

$\{I \wedge (B[k] = A[k], 1 \leqq k \leqq M)\}$

coend

end

$\{B[k] = A[k], 1 \leqq k \leqq M)\}$

where $I = (buffer[k \ \mathbf{mod}\ N] = A[k], Vempty < k \leqq Vfull)$

$\wedge\ full = Vfull - Pfull$

$\wedge\ empty = N + Vempty - Pempty$

$\wedge\ 1 \leqq i \leqq M + 1$

$\wedge\ 1 \leqq j \leqq M + 1$

(5.4) Proof outline for $fg2'$ (*consumer*). Invariant I is given in (5.3)

$\{I \wedge IC \wedge Vempty = Pfull \wedge j = Vempty + 1\}$

consumer: **while** $j \leqq M$ **do**

begin $\{I \wedge IC \wedge Vempty = Pfull \wedge j = Vempty + 1 \wedge j \leqq M\}$

await $full > 0$ **then**

begin $full := full - 1;\ Pfull := Pfull + 1$ **end**;

$\{I \wedge IC \wedge Vempty = Pfull - 1 \wedge j = Vempty + 1 \wedge j \leqq M\}$

$y := buffer[j \ \mathbf{mod}\ N];$

$\{I \wedge IC > Vempty = Pfull - 1 \wedge j = Vempty + 1$

$\wedge\ j \leqq M \wedge y = A[j]\}$

await true then

begin $empty := empty + 1;$

$Vempty := Vempty + 1$ **end**;

$\{I \wedge IC \wedge Vempty = Pfull \wedge j = Vempty \wedge j \leqq M \wedge y = A[j]\}$

$B[j] := y;$

$\{I \wedge IC \wedge Vempty = Pfull \wedge j = Vempty \wedge j \leqq M \wedge B[j] = A[j]\}$

$j := j + 1$

$\{I \wedge IC \wedge Vempty = Pfull \wedge j = Vempty + 1 \wedge j \leqq M + 1\}$

end

$\{I \wedge IC \wedge j = M + 1\}$

$\{I \wedge (B[k] = A[k], 1 \leqq k \leqq M\}$

where $IC = (B[k] = A[k], 1 \leqq k < j)$

6. Blocking and Deadlock

Because of the **await** statements, a process may be delayed, or "blocked" at an **await**, until its condition B is true.

(6.1) **Definition.** Suppose a statement S is being executed. S is *blocked* if it has not terminated, but no progress in its execution is possible because it (or all of its subprocesses that have not yet terminated) are delayed at an **await**.

Blocking by itself is harmless; processes may become blocked and unblocked many times during execution. However, if the whole program becomes blocked, this is serious because it can never be unblocked and thus the program cannot terminate.

(6.2) **Definition.** Execution of a program *ends in deadlock* if it is blocked.

(6.3) **Definition.** A program S with proof $\{P\}\ S\ \{Q\}$ is *free from deadlock* if no execution of S which begins with P true ends in deadlock.

We wish to derive sufficient conditions under which a program is free from deadlock. First of all, a proof of correctness of a program S includes a proof of correctness of a program S', together with several applications of the auxiliary variable rule (3.7) which reduce S' to S. Since the reduction consists of deleting assignments to auxiliary variables, we take as obvious the following theorem (a proof with respect to a particular execution model appears in Owicki [16]).

(6.4) **Theorem.** Suppose program S' is free from deadlock, and suppose S is derived from S' by application of inference rule (3.7). Then S is also free from deadlock.

We are now in a position to give sufficient conditions for freedom from deadlock.

(6.5) **Theorem.** Let S be a statement with proof $\{P\}\ S\ \{Q\}$. Let the **await**s of S which do not occur within **cobegin**s of S be

A_j: **await** B_j **then** ...

Let the **cobegin**s of S which do not occur within other **cobegin**s of S be

T_k: **cobegin** S_1^k // S_2^k // ... // $S_{n_k}^k$ **coend**

Define

$$D(S) = \left[\bigvee_j (pre(A_j) \wedge \neg B_j)\right] \vee \left[\bigvee_k D_1(T_k)\right]$$

$$D_1(T_k) = \left[\bigwedge_i (post(S_i^k) \vee D(S_i^k))\right] \wedge \left[\bigvee_i D(S_i^k)\right]$$

Then $D(S) = \textbf{false}$ implies that in no execution of S can S be blocked. Hence, if S is a program, S is free from deadlock.

Proof. We show by induction on the level of nesting of **cobegin**s in S that S blocked in state m implies $D(S)[m] = \textbf{true}$. Hence $D(S) = \textbf{false}$ would indicate that S cannot be blocked. Suppose S has no **cobegin**s. Then it is blocked at a single **await** with label A_j. Therefore $(pre(A_j) \wedge \neg B_j)[m] = \textbf{true}$ and $D(S)[m] = \textbf{true}$.

Suppose S contains **cobegin**s, and is blocked in state m. Then either it is blocked at an **await** A_j, in which case $D(S)[m] = \textbf{true}$ as above, or one of its parallel processes T_k is blocked. Consider one of T_k's processes S_i^k. By induction, we know that if S_i^k is blocked in state m, that $D(S_i^k)[m] = \mathbf{true}$. Now, since T_k is blocked, then each of its processes S_i^k has terminated or is blocked, and moreover, at least one of its processes S_i^k is blocked. Inspection of formula $D_1(T_k)$ shows therefore that $D_1(T_k)[m] = \mathbf{true}$. Hence $D(S)[m] = \mathbf{true}$. q.e.d.

Note that (6.5) provides a static check in order to prove a property of all executions of S; to show freedom from deadlock we need only manipulate the assertions in the proof of correctness. The amount of detail is directly proportional to the level of nesting of parallel statements.

If a statement S contains no parallel statements, then $\bigvee_k D_1(T_k)$ is the empty union and is false, and hence $D(S)$ reduces to

$$\bigvee_j ((pre(A_j) \wedge \neg B_j).$$

If, further, S has no **await**s, then this union is also empty and $D(S)$ is **false**. Thus, a sequential program without **await**s is free from deadlock. It is also easy to apply the theorem to show that if a program has no **await**s, or if all **await**s have the form **await true then** ..., then the program is free from deadlock. Finally if a parallel statement T is not supposed to terminate, i.e. $post(T) = \mathbf{false}$, then $D_1(T)$ reduces to

$$D_1(T) = \bigwedge_i D(S_i) \qquad \text{where the } S_i \text{ are the processes of } T.$$

Section 4 contains several examples of programs with proof outlines. Program (4.1) is free from deadlock since the conditions of the **awaits** are all **true**. *Findpos* in (4.2) is free from deadlock since it has no **awaits**.

To prove freedom from deadlock for the producer-consumer program (4.7), we use its proof outline given in (4.8)–(4.10). We have

$$\begin{aligned}
&D(producer) && \Rightarrow in < M \wedge in\text{-}out = N\\
&post(producer) && \Rightarrow in = M\\
&D(consumer) && \Rightarrow out < M \wedge in\text{-}out = 0\\
&post(producer) && \Rightarrow out = M
\end{aligned}$$

Writing $D_1(fg1') = x \wedge y$, where $fg1'$ is the **cobegin** statement, we then rewrite x as the "or" of 4 terms.

$$\begin{aligned}
x \Rightarrow\ & [in < M \wedge in\text{-}out = N \wedge out < M \wedge in\text{-}out = 0]\\
& \vee [in < M \wedge in\text{-}out = N \wedge out = M]\\
& \vee [in = M \wedge out < M \wedge in\text{-}out = 0]\\
& \vee [in = M \wedge out = M]\\
& \Rightarrow N = 0 \vee N < 0 \vee \mathbf{false} \vee in = out = M\\
& \Rightarrow N \leqq 0 \vee in = out = M\\
y = {} & D(producer) \vee D(consumer) \Rightarrow in < M \vee out < M\\
D(fg1) = {} & D_1(fg1') = x \wedge y \Rightarrow N \leqq 0.
\end{aligned}$$

Hence, sufficient conditions for freedom from deadlock in $fg1$ is that $N > 0$—that is, the buffer has room at least one element.

In some programs using semaphores, it is often useful to know how many processes can be blocked at a particular moment, waiting to enter a critical section. We can prove a general theorem about such programs, generalizing the idea of blocking a bit at the same time.

(6.6) **Theorem.** Consider a program of the form (6.7). Then at any point of execution at most $n-m$ of the processes $S1, \dots, Sn$ can be blocked at $P(s)$. Furthermore, if a process is blocked at $P(s)$, then m processes are executing the critical section or $V(s)$.

(6.7) $s := m; \dots$
cobegin $S1$ // ... // Sn // ... // Sp **coend**

where each Si, $1 \leq i \leq n$, has the form given below, none of the processes Si, $i > n$, reference s, and the only references to s are those shown:

```
Si: ...
    while true do
      begin noncritical section;
            P(s);
            critical section;
            V(s);
            noncritical section
      end
```

Proof. In (6.8) we show this same program written using **await**s, with auxiliary variables, and with a proof outline. The assertions that $INCi=1$ throughout the critical section and $INCi=0$ elsewhere are justified since the only operations on $INCi$ are those explicitly shown. Simillarly, assertion I holds throughout because there are no other operations on s. The interference-free requirement is easily verified, because each assertion is a statement about $INCi$, which is not changed in Sj, $j \neq i$, and about I, which is invariant over the statements in process Sj.

Now suppose $n-m+k$, $k \geq 0$, processes are blocked at $P(s)$. Then we have $INCi=0$ for these processes, and hence $s = m - \sum_{j=1}^{n} INCj > 0$. But the fact that the processes are blocked at $P(s)$ implies that $s=0$, and we have a contradiction.

Secondly, suppose a process is blocked but only $m-k$, $k>0$ processes are executing their critical section or the **await true** statement. Because a process is blocked we have $s \leq 0$. But since $m-k$ processes are executing their critical section, for each of these processes we have $INCi=1$, and together with invariant I this yields $s>0$. Thus we have a contradiction.

(6.8) $s := m; INC1, INC2, \dots, INCn := 0, 0, \dots, 0; \dots$
$\{I \wedge (INCi=0, 1 \leq i \leq n)\}$
cobegin S1 // ... // Sn // ... // Sp **coend**
{**false**}
where Si, $1 \leq i \leq n$, is
$\{I \wedge INCi=0\}$
Si: ...
while true do

```
begin {I ∧ INCi=0}
      noncritical section;
      {I ∧ INCi=0}
      await s>0 then begin s:= s−1; INCi:= 1 end;
      {I ∧ INCi=1}
      critical section;
      {I ∧ INCi=1}
      await true then begin s:= s+1; INCi:= 0 end;
      {I ∧ INCi=0}
      noncritical section
end
{false}
```

where $I \equiv s = m - \sum_{i=1}^{n} INCi \wedge (\forall i, 1 \leqq i \leqq n, 0 \leqq INCi \leqq 1)$

Theorem 6.6 thus confirms our understanding of the semaphore.

7. Termination

Let us suppose that all operations are defined so that they always yield a value in the expected range. Then the only way a sequential program can fail to terminate is to loop infinitely in some **while** loop. In order to include proof of termination in a useful practical manner, one can replace the iteration inference rule (2.4) with another. Let t be an integer function, $t \geqq 0$. Let us also let $wdec(Q, S, t)$ mean that execution of S with precondition Q decreases the value of t by at least one. We can write this as

$$wdec(Q, S, t) \equiv \{Q \wedge t = c\}\ S\ \{t < c\}.$$

Then the new inference rule for iteration is

(7.1) *iteration with termination* $\dfrac{\{P \wedge B\}\ S\ \{P\},\ t \geqq 0,\ wdec(P \wedge B, S, t)}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{P \wedge \neg B\}}$

An alternate formulation allows t to become negative, but then requires that $(P \wedge t \leqq 0) \Rightarrow \neg B$:

(7.2) *iteration with termination* $\dfrac{\{P \wedge B\}\ S\ \{P\},\ wdec(P \wedge B, S, t),\ (P \wedge t \leqq 0) \Rightarrow \neg B}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{P \wedge \neg B\}}$

In any case, we have "axiomatized" loop termination in a practical, useful manner.

While there are some parallel programs which do not terminate, it would still be convenient to be able to prove termination of parallel programs. Suppose that we prove that each process of a parallel program S terminates, using (7.1) instead of (2.4). What else must we do to prove that S itself terminates? First of all, we must show that parallel execution of processes does not invalidate proof of sequential termination of the processes. If we do that, then the only way for the program not to terminate is the occurrence of deadlock. This leads us to redefine first of all the interference-free property:

(7.3) **Definition.** Given a proof $\{P\}\ S\ \{Q\}$ and a statement T with precondition $pre(T)$, we say that T *does not interfere with* $\{P\}\ S\ \{Q\}$ if the following three conditions hold:

(a) $\{Q \wedge pre(T)\}\ T\ \{Q\}$;

(b) Let S' be any statement within S but which is not within an **await**. Then $\{pre(S') \wedge pre(T)\}\ T\ \{pre(S')\}$;

(c) Let W be a loop within S, but not within an **await** of S. Let t be the integer function used in the proof of correctness of the loop (using (7.1) or (7.2)). Then $\{t=c \wedge pre(T)\}\ T\ \{t \leqq c\}$.

(7.4) **Definition.** $\{P1\}\ S1\ \{Q1\}, \ldots, \{Pn\}\ Sn\ \{Qn\}$ are *interference-free* if the following holds. Let T be an **await** or assignment statement (which does not appear in an **await**) of process Si. Then for all $j, j \neq i$, T does not interfere with $\{Pj\}\ Sj\ \{Qj\}$.

We can then redefine the rule (3.3) for the **cobegin** statement:

(7.5) *cobegin with termination*
$$\frac{\{P1\}\ S1\ \{Q1\}, \ldots, \{Pn\}\ Sn\ \{Qn\} \text{ interference-free, } \quad \{P1\}\ S1\ \{Q1\}, \ldots, \{Pn\}\ Sn\ \{Qn\} \text{ deadlock-free}}{\{P1 \wedge \ldots \wedge Pn\}\ \textbf{cobegin}\ S1\ //\ \ldots\ //\ Sn\ \textbf{coend}\ \{Q1 \wedge \ldots \wedge Qn\}}$$

The property "deadlock-free" for a set of parallel processes is defined as the sufficient conditions given in theorem (6.5) for freedom from deadlock.

As an example, consider program *Findpos* (4.3). We have thus far shown partial correctness. To show termination of *Evensearch* using rule (7.2) instead of (2.4), we introduce the function

$$te \equiv min(oddtop, eventop) - i$$

Note that for the loop in *Evensearch*, $te \leqq 0 \Rightarrow \neg B$. Secondly,

$$wdec(ES \wedge i < eventop \wedge i < M+1,\ body\ (Evensearch),\ te).$$

Similarly, we use the integer function $t0 \equiv min\ (eventop, oddtop) - j$ to show that *Oddtop* terminates. To show non-interference of *Evensearch* by *Oddsearch*, we must show that *Oddsearch* does not increase *te* (the argument for *Evensearch* not interfering with *Oddsearch* is similar). The only statement in Oddsearch which changes a variable of *te* is $oddtop := j$. We now show that execution of this does not increase *te*:

$$\{te = c \wedge pre(oddtop := j)\}$$
$$\{min(oddtop, eventop) - i = c \wedge j < oddtop\}$$
$$\{min(j, eventop) - i \leqq c\}$$
$$oddtop := j$$
$$\{min(oddtop, eventop) - i \leqq c\}$$

Finally, there is no deadlock since there are no **await**s in the program.

8. Conclusions

We have developed a deductive system for proving properties of parallel programs, building on work by Hoare [9, 10]. Besides partial correctness, the system lends itself to proving other properties: freedom from deadlock, and termination. A paper is in preparation concerning mutual exclusion. Once one has a partial correctness proof, one can often prove these other properties just by manipulating in some fashion the assertions already created for the partial correctness proof. Hence the proofs of these properties of execution only require work with static objects—the assertions—instead of with the dynamic execution of the program.

A number of other properties could be considered: priority assignments, progress for each process, blocking of some subset of the processes, etc. Many of these are difficult to define in a uniform way, while others require a model with definite rules for scheduling competing processes. Hopefully, future work will broaden the range of properties which can be dealt with using axiomatic methods.

The synchronization primitive discussed is admittedly primitive, and a paper is in preparation (19) which covers the same material using a higher level synchronization statement, Hoare's **with-when** statement. However, this primitive synchronization statement has proved useful. First, it has given us insight into how to understand parallel processes, as discussed in Section 3. Secondly, we have used it on a number of parallel programs from the literature—*Findpos*, the consumer-producer problem, etc., and we feel it will be useful in practical work with parallel programs. It gives us a method for dealing more formally with other synchronization primitives.

The "insight" gained from this work, towards understanding parallelism, may not have come across well if the reader already understood the examples beforehand. A quite complicated problem with as fine a grain of interleaving as can be imagined, Dikjstra's on-the fly garbage collector [6], has been proved correct in what we feel is a satisfactory manner [7], and we invite the reader to study it. The first author was also able to verify the semaphore solutions for readers and writers proposed by Courtois, Heymans and Parnas. This was fairly hard to do, because of the complexity of their solution which gives priority to the writer.

References

1. Ashcroft, E. A., Manna, Z.: Formalization of properties of parallel programs. Machine Intelligence **6**. Edinburgh: University of Edinburgh Press 1971, p. 17–41
2. Ashcroft, E. A.: Proving assertions about parallel programs. Dept of Computer Science, University of Waterloo, CS 73-01, 1973
3. Cadiou, J. M., Levy, J. J.: Mechanical proofs about parallel processes. Proc. 14 Annual IEEE Symposium on Switching and Automata Theory, 1973, p. 34–48
4. Clint M.: Program proving: coroutines. Acta Informatica **2**, 50–63 (1973)
5. Cook, S. A.: Axiomatic and interpretive semantics for an ALGOL fragment. Dept. of Computer Science, Toronto, TR 79, 1975.
6. Dijkstra, E. W. et al.: On-the-fly garbage collection: an exercise in cooperation. In Working Material for the NATO Summer School on Language Hierarchies and Interfaces, Munich, 1975
7. Gries, D.: An exercise in proving properties of parallel programs. (Submitted to Comm. ACM)
8. Habermann, A. N.: Synchronization of communicating processes. Comm. ACM **15**, 171–176 (1972)
9. Hoare, C. A. R.: An axiomatic basis for computer programming. Comm. ACM **12**, 576–580 (1969)

10. Hoare, C. A. R.: Towards a theory of parallel programming. In: Hoare, C.A.R., Perrot, R. H. (eds.): Operating systems techniques. New York: Academic Press 1972
11. Hoare, C. A. R., Lauer, P. E.: Consistent and complementary formal theories of the semantics of programming languages. Acta Informatica **3**, 135–153 (1974)
12. Lauer, P.E.: Consistent formal theories of the semantics of programming languages. IBM Laboratory Vienna, TR 25.121, 1971
13. Lipton, R. J.: On synchronization primitive systems. Carnegie Mellon University, PhD Thesis, 1974
14. Lipton, R. J.: Reduction: a new method for proving properties of systems of processes. Yale Computer Science Research Report 30, 1974
15. Newton, G.: Proving properties of interacting processes. Acta Informatica
16. Owicki, S.: Axiomatic proof techniques for parallel programs. Computer Science Dept., Cornell University, PhD thesis, 1975
17. Rosen, B. K.: Correctness of parallel programs: the Church-Rosser approach. T. J. Watson Research Center, Yorktown Heights (N. Y.), IBM Research Report RC5107, 1974
18. Dijkstra, E. W.: The structure of the THE multiprogramming system. Comm. ACM **11**, 341–347 (1968)
19. Owicki, S., Gries, D.: Axiomatic proof techniques for parallel programs II. In preparation

Susan Owicki
Computer Science Dept.
Cornell University
Ithaca, NY 14853
USA

David Gries
Cornell University
Dept. of Computer Science
Upson Hall
Ithaca, N.Y. 14850
USA