A CONSISTENT AND COMPLETE DEDUCTIVE SYSTEM FOR THE VERIFICATION OF PARALLEL PROGRAMS+

Susan Owicki

TR 76-278

May 1976

Department of Computer Science
Cornell University
Ithaca, New York 14853

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

A CONSISTENT AND COMPLETE DEDUCTIVE SYSTEM FOR THE VERIFICATION OF PARALLEL PROGRAMS+

Susan Owicki
Cornell University

ABSTRACT:

The semantics of a simple parallel programming language is presented in two ways: deductively, by a set of Hoare-like axioms and inference rules, and operationally, by means of an interpreter. It is shown that the deductive system is consistent with the interpreter. It would be desireable to show that the deductive system is also complete with respect to the interpreter, but this is impossible since the programming language contains the natural numbers. Instead it is proved that the deductive system is complete relative to a complete proof system for the natural numbers; this result is similar to Cook's relative completeness for sequential programs.

The deductive semantics given here is an extension of an incomplete deductive system proposed by Hoare. The key difference is an additional inference rule which provides for the introduction of auxiliary variables in a program to be verified.

1. INTRODUCTION

The presence of parallelism in a programming language greatly complicates the problem of program verification, due to the essential non-determinism introduced by concurrency. A number of techniques for verifying parallel programs have been suggested, see [1], [2], [3], [4], [10], [12], [13], [14], [15], [17], [20]. The technique presented here is a deductive system for proving the partial correctness of parallel programs; it is an extension of Hoare's work [10]. The utility of the proposed deductive system has been demonstrated elsewhere (see [8], [18], [19]): it provides an easy-to-use technique for proving partial correctness; it gives the programmer guidance in creating well-structured and easily-verified programs; and it can be the starting point in the proof of a number of additional properties of parallel programs (e.g. termination, freedom from deadlock, mutual exclusion). In this paper the deductive system is evaluated from a more mathematical perspective and is shown to be consistent and in some sense complete with respect to an interpretive model of program execution.

The consistency and completeness of Hoare's deductive system for sequential programs has been discussed by Cook [5], who introduced the important concept of completeness relative to a proof system for the data types of the programming language. The approach taken here is quite similar to Cook's: first the programming language is presented and its semantics given both by a deductive system and by an interpreter. It is then possible to show that the deductive semantics is consistent and relatively complete with respect to the interpreter, although these results cannot be proved as directly as they can be for sequential programs. In particular, in order to obtain completeness, an inference rule is needed which allows the addition of variables and assignment statements to the program to be verified.

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

2. THE PROGRAMMING LANGUAGE

The programming language used is derived from Algol 60. It contains the usual assignment, conditional, <u>while</u>, compound and null statements, plus two statements intended for parallel programming. Variables and expressions range over the natural numbers with the usual operations. Procedures and variable declarations are not included, since they introduce complications which are irrelevant to the problems of parallelism.

Parallel execution is initiated by a statement of the form

<u>resource</u> r_1(variable list),...,
r_m(variable list):
<u>parbegin</u> S_1 // ... // S_n <u>parend</u>

Here a resource r_i is a set of logically connected shared variables, and $S_1 \ldots S_n$ 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:

<u>with</u> r <u>when</u> B <u>do</u> 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.

Much of the complexity of parallel programs stems from the way processes can interfere with each other as they use shared variables. The following syntax restrictions ensure that all variables which could cause conflict are accessible to only one process at a time.

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_i, it cannot appear in S_j $(i \neq j)$ unless it belongs to a resource.

3. DEDUCTIVE SEMANTICS -- THE AXIOMS AND INFERENCE RULES

In the deductive semantics, the meaning of a programming language statement S is given by the formula {P}S{Q}. Here P and Q are assertions, i.e. formulas of the first-order predicate calculus (the assertion language used in this paper includes the natural numbers with the usual operations). Informally, the partial correctness formula {P}S{Q} means that if P is true before S is executed, either S will fail to halt or Q will hold after S finishes execution.

Figure 1 gives the axioms and inference rules for the parallel language of Section 1. A0-A5 are Hoare's sequential rules [9]. The proof system D in A0 can be any sound proof system for the natural numbers -- this is discussed further in Section 6. A6-A7 are stronger versions of the rules proposed by Hoare in [10]. Note that an invariant relation $I(r_j)$ is required for each resource r_j; $I(r_j)$ describes the "reasonable" states of the resource. $I(r_j)$ must be true when parallel execution begins (A7) and is preserved by each critical section (A6); thus in A7 it is assumed to hold when parallel execution ends.

A0 consequence

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

A1 assignment

$$\{P^x_E\}\ x{:=}E\ \{P\}$$

A2 null

$$\{P\}\ ;\ \{P\}$$

A3 composition

$$\frac{\{P_1\}S_1\{P_2\},\ \{P_2\}S_2\{P_3\},\ldots,\{P_n\}S_n\{P_{n+1}\}}{\{P_1\}\ \underline{\text{begin}}\ S_1;\ldots;S_n\ \underline{\text{end}}\ \{P_{n+1}\}}$$

A_4 alternation

$$\frac{\{P \wedge B\}\ S_1\ \{Q\},\ \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \underline{\text{if}}\ B\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \{Q\}}$$

A5 iteration

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \underline{\text{while}}\ B\ \underline{\text{do}}\ S\ \{P \wedge \neg B\}}$$

A6 critical section

$$\frac{\{P \wedge B \wedge I(r)\}\ S\ \{Q \wedge I(r)\}}{\{P\}\ \underline{\text{with}}\ r\ \underline{\text{when}}\ B\ \underline{\text{do}}\ S\ \{Q\}}$$

A7 parallel

$$\frac{\{P_i\}\ S_i\ \{Q_i\}\ \ 1 \leq i \leq n}{\begin{array}{l}\{P_1 \wedge \ldots \wedge P_n \wedge I(r_1) \wedge \ldots \wedge I(r_m)\}\\ \quad \underline{\text{resource}}\ r_1(),\ldots,r_m():\\ \quad\quad \underline{\text{parbegin}}\ S_1\ //\ \ldots\ //\ S_n\ \underline{\text{parend}}\\ \{Q_1 \wedge \ldots \wedge Q_n \wedge I(r_1) \wedge \ldots \wedge I(r_m)\}\end{array}}$$

provided the proof of {P} S {Q} uses variables safely (see text)

A8 auxiliary variables

Let AV be a set of variables such that x εAV => x appears in S' only in assignments y:=E, where y εAV. Then if P and Q are assertions which do not contain free any variables from AV, and if S is obtained from S' by deleting all assignments to variables in AV

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

Figure 1. Axioms and Inference Rules for the Parallel Programming Language

Rule A7 requires that the proof of $\{P_i\}S_i\{Q_i\}$ use variables safely. This is a syntactic restriction which ensures that in each line {P'}S'{Q'} in the proof of

$\{P_i\}S_i\{Q_i\}$, P' and Q' contain only those variables which process S_i has a right to access at S'.

Definition: Let S be the statement

resource $r_1()$, ... , $r_m()$:
  parbegin S_1 // ... // S_n parend

and let S' be a statement in process S_i. Then

Proof-var(r_j,S)

= {x: x is not assigned a value in S except in a critical section for r_j}

Proof-var(S',S)

= {x: x is not assigned a value in any process S_j with $i \neq j$, or $x \in$ Proof-var(r_j,S) and S' is inside a critical section for r_j}

(We will use Proof-var(r_j) and Proof-var(S') when S is obvious from context).

Definition: Let S be a parbegin statement with processes S_1 ... S_n and resources r_1 ... r_m. Then a proof of $\{P_i\}S_i\{Q_i\}$ uses variables safely iff

1. all free variables in $I(r_j)$ belong to Proof-var(r_j,S), $1 \leq j \leq m$
2. if S' is a statement in S_i, and {P}S'{Q} is a line in the proof, then all free variables in P and Q belong to Proof-var(S',S), $1 \leq i \leq n$

Note that the variables in Proof-var(S',S) are exactly those which cannot be changed by another process when S' is being executed.

Finally, A8 is a new inference rule which provides for the introduction of auxiliary variables in a program to be verified. After verifying the partial correctness of the expanded program, A8 can be used to derive the partial correctness of the original. Many authors have noted the usefulness of auxiliary variables (see for example [3], [15]) but have not provided a formal mechanism for incorporating them in program proofs.

In this paper it will be assumed that program proofs contain no extraneous derivations, i.e. every line in the proof, except the last, is used in a subsequent line.

An example of parallel program verification based on the deductive semantics is presented very informally in Figure 2. Here the assertions from a formal proof are set off by braces {} and interspersed with the program text.

```
{x=0}
addl: begin y:=0; z:=0;
        {y=0 ∧ z=0 ∧ I(r)}
        resource r(x,y,z): parbegin
            {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}
       parend
       {y=1 ∧ z=1 ∧ I(r)}
end
{x=2}

I(r) = {x=y+z}
```

Figure 2. Assertions from a proof of {x=0} addl {x=2}

4. THE INTERPRETER

The semantics of the parallel language of Section 2 can also be presented by giving an interpreter for programs in the language. In this section we define such an interpreter in terms of the computations it may exhibit in executing a program. Informally, the interpreter executes sequential statements in the usual way. It implements parallelism by selecting (nondeterministically) one of the parallel processes and advancing that process according to the usual rules for sequential program execution. Thus the interpreter uses non-determinism to simulate parallelism, but it is defined in such a way that the results are equivalent to those which would be obtained using true parallelism.

Definition: A program state for the interpreter executing a program S is an ordered pair (c,v). The control state c is a set of statements from S; these are the statements which are next to be executed in each process. For conveniente we will assume that each statement in S has a distinct label, and will use the statement and its label interchangeably in c. The variable state v is a mapping from variables of S to values. If E is an expression on the program variables of S, E(v) denotes the result of evaluating E in state v.

Definition: The state transition function δ for program S maps statements X program states to program states.

$\delta(T,(c,v))$ represents the effect of initiating T in state (c,v).

$\delta(T,(c,v))$ = undefined if $T \notin c$, or if T is the statement with r when B do T_1 and either $B(v)$ = false or c already contains a a statement which belongs to a critical section for r
= (c',v') otherwise.

where $v'(x) = E(v)$ if T is the statement $x:=E$
$= v(x)$ otherwise

$c' = (c \sim \{T\}) \cup$ successor $(T,(c,v))$, where successor $(T,(c,v))$ is the set of statements to be initiated after T. For example,

a) if T is if B then T_1 else T_2, successor $(T,(c,v))$ = if $B(v)$ then $\{T_1\}$ else $\{T_2\}$

b) if T is resource $r_1,\ldots,r_n$: parbegin T_1 // ... // T_N parend; successor $(T,(c,v)) = \{T_1,\ldots,T_N\}$

c) if T is the last statement in S, or the last statement in a set of parallel processes which still have statements in $c \sim \{T\}$, successor $(S,(c,v)) = \phi$.

Definition: A partial computation of the interpreter for program S is a sequence of program states, $C = (c_0,v_0),\ldots,(c_n,v_n)$, such that $c_0 = \{S\}$, and $(c_i,v_i) = \delta(S_i,(c_{i-1},v_{i-1}))$ for some $S_i \in c_{i-1}$, $1 \le i \le n$. A computation is a partial computation which terminates i.e. $c_n = \phi$. Note that C is completely determined by c_0 and the sequence of statements $S_1,S_2,\ldots,S_n$.

This completes the definition of the interpreter. Some related concepts which will be useful in Section 5 and 6 are defined in terms of the interpreter.

Definition: Let T be a statement in program S, and $C = (c_0,v_0),\ldots,(c_n,v_n)$ be a partial computation for S. Then T is current in (c_i,v_i) if $T \in \{c_i\}$, and T is current after C if it is current in (c_n,v_n). Also C finishes T if C has executed the last statement of T and has not initiated another statement from the same process as T.

Definition: The formula $\{P\}S\{Q\}$ is true for the interpreter iff any computation $C = (c_0,v_0),\ldots,(c_n,v_n)$ for which $P(v_0)$ = true has $Q(v_n)$ = true, i.e., any computation which starts S with P true must end with Q true.

5. CONSISTENCY

This section demonstrates that the axiomatic semantics of Section 3 is consistent with the interpreter of Section 4. Similar consistency results for sequential programs have been proved by Hoare and Lauer [11], Cook [5], Gorelick [7], and Donahue [6]. In these papers the consistency results were obtained by showing that the axioms (A1, A2) and inference rules (A0, A3-A5) are sound with respect to some sequential interpreter.

In the parallel case this approach fails for both A6 and A7. For A7, problems arise because the computations for a parbegin statement cannot be obtained simply by combining the independent computations of its components; this makes it difficult to prove the soundness of A7 directly. (In contrast consider A3. Here a direct proof is easy because a computation for begin $S_1;\ldots;S_n$ end is the concatenation of computations for $S_1,\ldots,S_n$.) Rule A6 is valid only within the context of a parbegin statement; fortunately this is the only place where critical sections can be used. If A6 is to be sound, any computation which starts the statement with r when B do S with P true must start statement S with $P \land B \land I(r)$ true. This can only be established within a parbegin statement, whose proof includes the requirement that $I(r)$ holds when parallel execution begins and is preserved by each critical section. Because of these difficulties, the proof of the consistency theorem demonstrates the soundness of a complete program proof rather than the soundness of each proof rule.

Definition: Let S' be a statement in a program S. The pre-conditions (post-conditions) of S' in a proof of $\{P\}S\{Q\}$ are the assertions which appear before (after) S' in formulas in the proof.

Lemma 5.1: Suppose S' appears k times in the proof of $\{P\}S\{Q\}$, in the formulas $\{P_1\}S'\{Q_1\},\ldots,\{P_k\}S'\{Q_k\}$ (in that order). Then $\{P_1\}S'\{Q_1\}$ is derived using one of the rules A1-A8, and $\{P_i\}S'\{Q_i\}$ $1 < i \le k$, is derived using A0 and $\{P_{i-1}\}S'\{Q_{i-1}\}$. Thus $P_k \vdash P_{k-1} \vdash \ldots \vdash P_1$ and $Q_1 \vdash Q_2 \vdash \ldots \vdash Q_k$. The formula $\{P_k\}S'\{Q_k\}$ is either the last line of the proof (if S = S') or is used in deriving a formula $\{P'\}T\{Q'\}$, where T is either the statement immediately containing S' in S or a reduction of S' according to A8.

Proof: Follows from the requirement that the proof of $\{P\}S\{Q\}$ contains no extraneous derivations.

Theorem 5.1: (Consistency of A0-A7) If $\{P\}S\{Q\}$ can be proved using A0-A7, then $\{P\}S\{Q\}$ is true for the interpreter.

Proof: We start with a proof of $\{P\}S\{Q\}$ and show that the pre and post conditions from that proof must hold at the appropriate times during the execution of S. More formally, let $C = (c_0,v_0),\dots,(c_n,v_n)$ be a partial computation for S with $P(v_0) = \text{true}$. Then C must satisfy the following conditions.

1. Let T be a statement in S, P' a pre-condition, and Q' a post-condition of T in the proof of $\{P\}S\{Q\}$. Then

   a) if T is current after C, $P'(v_n) = \text{true}$.

   b) if C finishes T, $Q'(v_n) = \text{true}$.

2. Let r be a resource in S with invariant I(r). Then if C is executing the parbegin statement where r is declared, but is not in the midst of a critical section for r, then $I(r)(v_n) = \text{true}$.

Note that 1b implies that any computation for S which starts with P true must finish with Q true, since Q is a post-condition of S in the proof of $\{P\}S\{Q\}$. Thus a proof of 1 and 2 establishes that $\{P\}S\{Q\}$ is true for the interpreter. The proof of 1 and 2 uses induction on the length of C.

Base step: 1a. $C = (\{S\},v_0)$. By assumption, $P(v_0) = \text{true}$.
If P' is any other pre-condition of S, $P \vdash P'$ (Lemma 5.1).
Thus $P'(v_0) = \text{true}$.
1b and 2 do not apply.

Induction step: Let $C' = (c_0,v_0),\dots,(c_{n-1},v_{n-1})$. By induction, 1 and 2 are satisfied for C'; we must show they are satisfied for C. Let S_n be the statement in c_{n-1} such that $(c_n,v_n) = \delta(S_n,(C_{n-1},v_{n-1}))$.

Proof of 1: Consider cases of S_n and T.

Case 1: S_n and T are in different processes. Now v_n agrees with v_{n-1} on all variables in Proof-var(T), since S_n can not change any of those variables.

a) If T is current after C, it was current after C', and $P'(v_{n-1}) = \text{true}$, by induction. So $P'(v_n) = \text{true}$.

b) If C finishes T then C' finishes T, and by similar reasoning $Q'(v_n) = \text{true}$.

Case 2: S_n and T are from the same process. A complete verification of condition 1 requires a detailed analyses of all cases of S_n and T. Such an analysis is given in [19]; here we present some representative examples.

Example 1: S_n is an assignment statement which appears in S in the context

$$L_1\text{: }\underline{\text{while}}\ B\ \underline{\text{do}}\ \underbrace{x:=E;}_{S_n}$$

After C, i.e. after the execution of S_n, S_n is finished and L_1 is current. Thus we must show that the post-conditions of S_n and the pre-conditions of L_1 hold in v_n.

First, note that the first line in the proof of $\{P\}S\{Q\}$ which refers to S_n must have the form $\{R^x_E\}S_n\{R\}$ (Lemma 5.1).

Now S_n is current after C', so by induction $R^x_E(v_{n-1}) = \text{true}$. Then $R(v_n) = \text{true}$ since v_n is obtained from v_{n-1} by assigning the value of E to x. If R' is any other post-condition of S_n, $R \vdash R'$ (Lemma 5.1), so that $R'(v_n) = \text{true}$. Now the last line of the proof which refers to S_n is used to derive $\{P'\}L_1\text{: }\underline{\text{while}}\ B\ \underline{\text{do}}\ S_n\{P' \wedge \neg B\}$ using A5 (Lemma 5.1), so it must have the form $\{P' \wedge B\}S_n\{P'\}$. Since P' is a post-condition of S_n, $P'(v_n) = \text{true}$. But P' is also the first pre-condition of L_1, so by Lemma 5.1 $P' \vdash P''$, where P" is any pre-condition of L_1; thus $P''(v_n) = \text{true}$.

Example 2: S_n is with r when B do T;

T is current after C, and no statements are finished. The last line of the proof which refers to T is

$$\{P' \wedge B \wedge I(r)\}\ T\ \{Q' \wedge I(r)\}$$

and the first which refers to S_n is $\{P'\}S_n\{Q'\}$(Lemma 5.1). By induction $P'(v_{n-1}) = \text{true}$, since S_n is current in v_{n-1}. Since S_n can execute after C', $B(v_{n-1})$ is true, and no critical section on r is in execution for C'. But then $P' \wedge B \wedge I(r)$ holds for v_{n-1}, using part 2 of the induction hypothesis, and also for v_n, since $v_{n-1} = v_n$. Applying Lemma 5.1, all pre-conditions of T hold in v_n.

Example 3: S_n is a parbegin statement All of the processes of S_n will be current after C, and it is easily checked that their pre-conditions hold for v_n.

Example 4: C finishes a parbegin statement T. Then S_n finishes the last process of T (all the others were finished by C'). By previous arguments, the post-condition of each process holds for v_n; also each resource invariant holds because no critical sections can be in execution when the processes finish. Thus the post conditions of the parbegin statement T hold in v_n.

Proof of 2: Suppose no critical section for resource r is in execution in C. If no critical section for r was in execution in C', $I(r_j)(v_{n-1})$ = true by induction. Since v_n must agree with v_{n-1} on all variables in Proof-var(r), $I(r)(v_n)$ = true also.

If some critical section with r when B do S_1 was in execution in C', but not in C, then C finishes S_1. Now S_1 must have a post-condition with the form $Q' \wedge I(r)$ (Lemma 5.1), so by 1b $I(r)(v_n)$ = true.

Theorem 5.2 (Consistency of A8): If {P}S{Q} can be derived from {P}S'{Q} using A8, and {P}S'{Q} is true for the interpreter, then {P}S{Q} is true for the interpreter.

Proof: Deleting the assignments to auxiliary variables does not effect the flow of control or the values assigned to other program variables. Thus S' has the same effect as S on the variables which appear in P and Q.

Theorem 5.3 (Consistency of A0-A8): If {P}S{Q} can be proved, it is true for the interpreter.

Proof: If the proof of {P}S{Q} uses A8, it can be rewritten so that all the steps using A8 appear at the end.

Let {P}S'{Q} be the last step which does not use A8. Then {P}S'{Q} is true for the interpreter by the consistency of A0-A7, and {P}S{Q} is true by the consistency of A8.

6. COMPLETENESS

The deductive system for parallel programs as proposed by Hoare (A0-A5, with weaker versions of A6 and A7) was not complete. For example, even A0-A7 are not powerful enough to prove the true formula {x=0} add2 {x=2}, where program add2 is shown.

```
add2: resource r(x): parbegin
      A1: with r when true do x:=x+1
      //
      A2: with r when true do x:=x+1
      parend
```

Figure 3. The program add2

Note that, for this program, Proof-var(A1) = Proof-var(A2) = ϕ. so that the only post-condition possible for A1 and A2 is P' = {true}. Then the strongest post-condition for add2 is $P' \wedge P' \wedge I(r) = I(r)$ (Lemma 5.1), and $\neg(I(r) \Rightarrow x=2)$, since I(r) must hold initially, when x=0. Even without the restrictions on proof-variables, the strongest valid candidate for post-condition of A1 and A2 is P" = $\{1 \le x \le 2\}$, and the strongest valid invariant is $I(r) = \{0 \le x \le 2\}$. But $\neg(P'' \wedge I(r) \Rightarrow x=2)$, so {x=2} still cannot be a post-condition of add2.

Section 6 is devoted to proving that A0-A8 are relatively complete in Cook's sense [5]. This implies that any true formula {P}S{Q} can be proved given sufficient knowledge about the data types of S, and strongly suggests that A0-A8 capture all the information about program execution which is relevant for partial correctness. As a first step, we consider the case when the data types of S are the natural numbers with operations <, =, +, *, and || (concatenation, to be defined shortly). The completeness result is then generalized to programs with any enumerable data domain and recursive operations.

The concatenation operation || which was mentioned above is used to represent finite sequences of natural numbers by a single number; it is included in the programming language operations because it will be needed with auxiliary variables.

There are many ways of encoding a sequence in an integer; here we choose to represent the empty sequence by 0 and the sequence $a_1, a_2, \ldots, a_k$ (or $a_1||a_2||\ldots||a_k$) by the number

$$\underbrace{1111\ldots1}_{a_1+1\ \text{1's}}\ 0\ \underbrace{11\ldots1}_{a_2+1\ \text{1's}}\ 0\ \ldots\ 0\ \underbrace{1\ldots.1}_{a_k+1\ \text{1's}}$$

Thus 0||2||1 = 10111011.

Theorem 6.1: (Relative completeness of A0-A8 for programs over the natural numbers): Let T be a program whose data domain is the natural numbers with <, =, +, *, and ||. If {P}T{Q} is true for the interpreter, then {P}T{Q} can be proved using A0-A8 and a complete proof system D for the natural numbers (clearly D is non-effective).

Proof: The theorem is proved for the case in which T contains at most one parbegin statement. If it contains more the principle is the same, although the details are more complicated. The proof is quite lengthy, and requires most of Section 6. An outline of the approach is given below.

Step 1: Give an algorithm for adding auxiliary variables to T, yielding a new program T*. Some of the auxiliary variables are used to encode a program history using the natural numbers. Note that {P}T*{Q} is true for the interpreter.

Step 2: Define predicates pre(S) and post(S) for each statement S in T*, and I(r) for each resource r. These predicates depend on the appropriate proof-variables. Roughly, pre(S) is true for any values of variables in Proof-var(S) which could occur when S is ready to execute. Post(S) and I(r) are defined similarly. Note that these are recursively enumerable predicates and so can be expressed by assertions, since the assertion language contains the natural numbers with the usual operations.

Step 3: For each statement S in S*, prove {pre(S)}S{post(S)}.

Step 4: From {pre(T*)}T*{post(T*)} derive {P}T*{Q}, using A0, $P \vdash_D$ pre(T*), and post(T*) $\vdash_D$ Q.

Step 5: Conclude {P}T{Q} from Step 4 and A8.

6.1 Step 1: Auxiliary Variables

Auxiliary variables are required only if T contains a parbegin statement of the form

T_0: resource $r_1,\dots,r_M$:
parbegin $T_1//\dots//T_N$ parend

In that case the auxiliary variables listed below must be added to T.

1. initstate: records the values of all variables when execution of T_0 begins.
2. Ptime[1:N] and Rtime[1:M]: "clock" variables which are used to establish the relative times at which events occurred in the execution of T_0.
3. Pcomp[1:N],Rcomp[1:M],comp[1:N,1:M]: sequences which record the history of critical section execution. Pcomp[i] is a history for process T_i, Rcomp[j] is a history for resource r_j, and comp[i,j] is a history of critical sections in process T_i which involve resource r_j.

Of course, it is assumed that none of these variables occur in the original program. If this is not the case, some variables must be renamed.

The program T* required in the proof of {P}T{Q} is obtained as follows.

1. If T contains no parbegin statement, T* = T.
2. Otherwise, replace the single parbegin statement T_0 by

```
begin initstate:= z_1||z_2||...||z_n;
(where {z_i} = {variables of T})
      Ptime:=Rtime:=Pcomp:=Rcomp:=
         comp:=0;
      resource r_1(...Rtime[1]),...,
         r_M(...Rtime[M]):
         parbegin T_1*//...//T_N* parend
```

T_i* is obtained by adding auxiliary variables to each critical section of T_i. Let CS be a critical section for r_j in T_i, and let the variables of r_j be $y_1,\dots,y_m$. Let num(CS) be a natural number, with each critical section in T being assigned a distinct number. Then replace

```
   CS: with r_j when B do S, by
CS: with r_j when B do
      begin
         if Ptime[i]<Rtime[j]
            then Ptime[i]:= Rtime[j]+1;
            else Ptime[i]:= Ptime[i]+1;
         Pcomp[i]:=Pcomp[i]||num(CS)||
            Ptime[i]||y_1||...||y_m;
         Rcomp[j]:=Rcomp[j]||num(CS)||
            Ptime[i]||y_1||...||y_m;
         comp[i,j]:=comp[i,j]||num(CS)||
            Ptime[i]||y_1||...||y_m;
         S*;
         Ptime[i]:=Ptime[i]+1;Rtime[j]:=Ptime[i];
         Pcomp[i]:=Pcomp[i]||num(CS)||
            Ptime[i]||y_1||...||y_m;
         Rcomp[j]:=Rcomp[j]||num(CS)||
            Ptime[i]||y_1||...||y_m;
         comp[i,j]:=comp[i,j]||num(CS)||
            Ptime[i]||y_1||...||y_m;
      end
```

Thus the variable state before parallel execution begins is recorded in initstate, while the history of resource use during parallel execution is recorded in Pcomp and Rcomp. The final values of Pcomp[i] and Rcomp[j] are sequences of entries corresponding to the beginning and end of execution of critical sections. Each entry contains the identity of the critical section involved (its number), the "time" at which it was started or finished, and the values of resource variables at that time. The time variables, Ptime[i] and Rtime[j], are updated in such a way that the times recorded in Pcomp[i] and Rcomp[j] are strictly increasing. The use of these auxiliary variables in a proof of {P}T*{Q} will be explained in Step 3. For now, note that a large part of their usefulness stems from the fact that initstate, Pcomp[i], and comp[i,j] belong to Proof-var(T_i), while initstate, Rcomp[j], and comp[i,j] belong to Proof-var(r_j).

The variables added in T* satisfy the definition of auxiliary variables, so {P}T{Q} can be proved by first proving {P}T*{Q} and then using A8 to remove the added statements. Note that {P}T*{Q} must be true for the interpreter, since T* has the same effect as T on the variables in P and Q.

Step 2: Assertions pre(S), post(S), and I(r).

We now define the assertions to be used in the proof of {P}T*{Q}. We first give the resource invariants, and then the pre and post conditions for each statement S in T*. The invariant for resource r must hold at any time in a computation when no critical section for r is in execution. The definition below specifies such an invariant, which is then interpreted informally.

Definition: Let r_j be a resource in T, with variables $y_1, y_2, \ldots, y_m$. The predicate $I(r_j)(v)$, defined on variable states of T*, holds iff the following conditions on Rcomp[j] and comp[i,j], $1 \le i \le N$, are satisfied.

1) $v(\text{Rcomp}[j]) = a_1 || a_2 || \ldots || a_{2n-1} || a_{2n}$, where

   a) $a_k = \text{num}(CS_k) || t_k || x_{k,1} || \ldots || x_{k,m}$, $1 \le k \le 2n$, where CS_k is the label of a critical section for r_j.

   b) $CS_{2k-1} = CS_{2k}$, $1 \le k \le n$

   c) $t_k < t_{k+1}$ $1 \le k < 2n$

   d) $x_{2k+1,h} = x_{2k,h}$, $1 \le k < n$, $1 \le h \le m$ and $x_{1,h}$ = value recorded for y_h in initstate.

   e) $v(y_h) = x_{2n,h}$ $1 \le h \le m$

2) For $1 \le i \le N$, v(comp[i,j]) is the subsequence of v(Rcomp[j]) obtained by deleting all elements a_k of v(Rcomp[j]) where CS_k does not belong to process T_i.

Informally, $I(r_j)(v)$ states that v(Rcomp[j]) is a history of the execution of critical sections for resource r_j. It consists of pairs of (n+2) tuples which record the order of critical section execution and the time and values of variables in r_j when the critical section was started (a_{2k-1}) and finished (a_{2k}) (1a and b). Time is strictly increasing in this history (1c). The initial value of the variables in r_j is recorded in initstate, and the variables do not change value between the end of one critical section and the beginning of the next (1d). For all i, comp[i,j] contains a history for critical sections for r_j in process T_i (2) which agrees with Rcomp[j].

The predicate $I(r_j)(v)$ is recursively enumerable, so it can be expressed as a first-order formula in the assertion language whose non-logical symbols are {<,=, +,*,||,0,1,...}. We will use $I(r_j)$ to refer to both the predicate and the assertion which expresses it. Note that the assertion can be chosen to contain free only variables from Proof-var(r_j), since Rcomp[j] and comp[i,j] both belong to Proof-var(r_j). Thus $I(r_j)$ can be used as a resource invariant in the proof of {P}T*{Q}.

Next we define a set of pre and post conditions for the statements of T*, considering first the case of a statement which does not belong to a parallel process in T*.

Definition: Let S be a statement in T* which does not belong to a parallel process. The predicates pre(S)(v), post(S)(v), defined on variable states of T*, are

pre(S)(v) $\equiv \exists$ a computation $C = (c_0,v_0), \ldots, (c_n,v_n)$ for T* such that $P(v_0)$ = true and $v_n = v$ and S is current after C.

post(S)(v) $\equiv \exists$ a computation $C = (c_0,v_0), \ldots, (c_n,v_n)$ for T* such that $P(v_0)$ holds and $v_n = v$ and C finishes S.

In other words, pre(s)(v) is true iff it is possible to start T* with P true and reach S in variable state v. Post(S)(v) is true iff it is possible to start T* with P true and finish S in variable state v. As with I(r), the predicates pre(S) and post(S) are recursively enumerable and can be expressed by assertions; we will use pre(S) and post(S) to refer to both the predicates and the assertions.

The definition of pre(S) and post(S) for a statement S belonging to a parallel process of T* is complicated by the fact that pre and post-conditions of S in a proof of {P}T*{Q} must depend only on variables in Proof-var(S). We will obtain such a definition by using the concept of a process computation.

Definition: Let

T_0: resource $r_1, \ldots, r_M$: parbegin $L_1{:}T_1 // \ldots // L_N{:}T_N$ parend

be the single parbegin statement in T*. A process computation for process T_i is a sequence $PC = (c_0,v_0), \ldots, (c_n,v_n)$ such that $c_0 = \{T_i\}$, and for $1 \le k \le n$, $\exists$ S_k such that $(c_k,v_k) = \delta(S_k,(c_{k-1},v_{k-1}))$ except that if S_k is a critical section for some resource r_j, the variables of r_j may take on arbitrary values in v_k so long as

$B(v_k)$ = true.

This definition describes a computation from the viewpoint of a single process: parallel execution is like non-parallel execution except that the values of resource variables may change unpredictably when they are not controlled by the process.

Definition: Let S be a statement in process T_i of the parbegin statement in T*. The predicates pre(S)(v) and post(S)(v), defined on variable states of T*, are

pre(S)(v) $\equiv \exists$ a process computation
$PC = (c_0,v_0),\ldots,(c_n,v_n)$ for process T_i such that $pre(T_0)(v_0)$ = true and $v(x) = v_n(x)$ $\forall x \in$ Proof-var(S), and S is current in c_n. If S belongs to a critical section CS for r_j, let (c_k,v_k) be the last state in PC such that $(c_k,v_k) = \delta(CS,(c_{k-1},v_{k-1}))$. Then $I(r_j)(v_k)$ = true.

post(S)(v) $\equiv \exists$ a process computation
$PC = (c_0,v_0),\ldots,(c_n,v_n)$ for process T_i such that $pre(T_0)(v_0)$ = true and $v(x) = v_n(x)$ $\forall x \in$ Proof-var(S) and PC finishes S. If S belongs to a critical section CS for resource r_j, let (c_k,v_k) be defined as before. Then $I(r_j)(v_k)$ = true.

In other words, pre(S)(v) is true iff it is possible to start T* with P true, reach the parbegin statement T_0 in some state v_0, then execute the process T_i independently until reaching a state where S is current and the variables in Proof-var(S) have the values given in v. Moreover, if S belongs to a critical section for r_j, $I(r_j)$ was true when that critical section was started. Post(S)(v) has a similar interpretation. Once again, the predicates pre(S) and post(S) are recursively enumerable and can be expressed by assertions; pre(S) and post(S) will be used to denote the assertions as well as the predicates. The assertions can be chosen to contain free only variables from Proof-var(S) and thus can be used as pre and post-conditions of S in a proof of {P}T*{Q}.

Step 3: Proving {pre(S)}S{post(S)}.

For each statement S in T* we prove {pre(S)}S{post(S)}, using induction on the structure of S. Some representative cases for S are given below. The only difficult case is the single parbegin statement.

Case 1: S is the assignment y:=E;

1. $\{post(S)^y_E\}$ y:=E {post(S)} A1
2. $\{pre(S)\} \vdash_D post(S)^y_E$ Lemma 6.1
3. {pre(S)} y:=E {post(S)} 1,2,A0

Lemma 6.1: We first show that

$\forall v(pre(S)(v) \Rightarrow post(S)^y_E(v))$.

Suppose S is not part of a parallel process in T*. Then

pre(S)(v) $\equiv \exists$ a computation $C = (c_0,v_0),\ldots,(c_n,v_n)$ such that $P(v_0)$ = true and $v = v_n$ and S is current after C.

$\Rightarrow \exists$ a computation
$C' = (c_0,v_0),\ldots,(c_n,v_n),(c',v')$
where $(c',v') = \delta(S,(c_n,v_n))$
such that $P(v_0)$ = true and
C' finishes S.

$\Rightarrow$ post(S)(v')

$\Rightarrow post(S)^y_E(v)$

If S belongs to a parallel process, the proof is the same except that it involves process computations rather than computations.

Now since

$\forall v(pre(S)(v) \Rightarrow post(S)^y_E(v))$,

and since D is a complete proof system for the natural numbers, $pre(S) \vdash_D post(S)^y_E$.

Case 2: S is with r_j when B do S'

1. {pre(S')}S'{post(S')} induction
2. $pre(S) \wedge I(r_j) \wedge B \vdash pre(S')$ lemma 6.2
3. $post(S') \vdash post(S) \wedge I(r_j)$ lemma 6.3
4. $\{pre(S) \wedge I(r_j) \wedge B\}S'\{post(S) \wedge I(r_j)\}$ 1,2,3,A0
5. {pre(S)}S{post(S)} 4,A6

Lemma 6.2: Let v be a variable state with $pre(S)(v) \wedge I(r_j)(v) \wedge B(v)$ = true.

Let $PC = (c_0,v_0),\ldots,(c_n,v_n)$ be the process computation whose existence is implied by pre(S)(v). Let $v'(x) = v_n(x)$ $\forall x \notin r$ and $v'(x) = v(x)$ $\forall x \in r$. Then $PC' = (c_0,v_0),\ldots,(c_{n-1},v_{n-1}),(c_n,v'),(c',v')$, where $(c',v') = \delta(S,(c_n,v'))$, is the process computation whose existence is required for pre(S')(v'). Since $v(x) = v'(x)$ $\forall x \in$ Proof-var(S'), this yields pre(S')(v). Thus

$\forall v(\text{pre}(S)(v) \wedge B(v) \wedge I(r)(v) \Rightarrow \text{pre}(S')(v))$,

so $\text{pre}(S) \wedge B \wedge I(r) \vdash_D \text{pre}(S')$.

Lemma 6.3: Let v be a variable state with post(S')(v) = true, and let

$PC = (c_0,v_0),\ldots,(c_n,v_n)$

be the process computation whose existence is implied by post(S')(v).

Now since PC finishes S', it also finishes S, yielding post(S)(v). To see that $I(r_j)(v)$ = true, note that $\text{post}(S')(v) \Rightarrow I(r_j)(v_k)$, where v_k is the variable state of PC upon starting S. Now executing S' preserves $I(r_j)$ (check the definition of T*), yielding $I(r_j)(v_n)$. Since $v(x) = v_n(x)$ $\forall x \in$ Proof-var(S'), and Proof-var(r_j) $\subseteq$ Proof-var(S'), this implies $I(r_j)(v)$. Thus

$\text{post}(S') \vdash_D \text{post}(S) \wedge I(r_j)$.

Case 3: S is

T_0: resource $r_1,r_2,\ldots,r_M$:
parbegin $L_1{:}T_1//\ldots//L_N{:}T_N$ parend.

1. $\{\text{pre}(T_i)\}T_i\{\text{post}(T_i)\}$, $1 \le i \le N$ — induction
2. $\{\text{pre}(T_1)\wedge\ldots\wedge\text{pre}(T_N)\wedge I(r_1)\wedge\ldots\wedge I(r_M)\}$ S $\{\text{post}(T_1)\wedge\ldots\wedge\text{post}(T_N)\wedge I(r_1)\wedge\ldots\wedge(r_M)\}$ — 1,A7
3. $\text{pre}(S) \vdash \text{pre}(T_1)\wedge\ldots\wedge\text{pre}(T_N) \wedge I(r_1)\wedge\ldots\wedge I(r_M)$ — lemma 6.4
4. $\text{post}(T_1)\wedge\ldots\wedge\text{post}(T_N)\wedge I(r_1) \wedge\ldots I(r_M) \vdash \text{post}(S)$ — lemma 6.5
5. $\{\text{pre}(S)\}S\{\text{post}(S)\}$ — 2,3,4,A0

Lemma 6.4: $\text{pre}(T_0)(v) \Rightarrow \text{pre}(T_i)(v)$ immediately from the definitions. $\text{pre}(T_0)(v) \Rightarrow I(r_j)(v)$ because the initialization performed in T* before starting S guarantees that the auxiliary variables used in $I(r_j)$ have the value zero in v, and the variables belonging to r_j have the value in v which is recorded in v(initstate). Thus $I(r_j)(v)$ = true.

Lemma 6.5: This is the most difficult case in the completeness proof. In order to show that

$\text{post}(T_1)(v) \wedge\ldots\wedge \text{post}(T_N)(v) \wedge I(r_1)(v) \wedge\ldots\wedge I(r_M)(v) \vdash \text{post}(T_0)(v)$,

we must derive a computation for T* which finishes T_0 in state v; this will be done by merging the process computations for T_i, $1 \le i \le N$, whose existence is implied by post(T_i). This merger will preserve the ordering of statements by "time" which can be inferred from the times stored in v(Pcomp[i]). The proof that such a merger is possible is quite complicated, and it is given in the appendix. Here we argue very informally that it is possible because the auxiliary variables guarantee that the independent process computations PC^i are in some sense compatible.

First, $\bigwedge_{i=1}^{N} \text{post}(T_i)(v)$ implies that all the processes started T_0 with the same initial state: the one recorded in v(initstate). Post(T_i)(v) also implies that v(Pcomp[i]) is a history of critical section execution in T_i, and that for $1 \le j \le M$, v(Pcomp[i]) gives the same history as $\bar{v}$(comp[i,j]) for critical sections for resource r_j. Now $I(r_j)(v)$ implies that v(Rcomp[j]) gives the same history as v(comp[i,j]) for critical sections in process T_i. Thus v(Pcomp[i]) and v(Rcomp[j]) give the same history for the critical sections they have in common, namely those for resource r_j in process T_i. Thus all processes manipulate resource r_j in the way recorded in v(Rcomp[j]), and by $I(r_j)(v)$, v(Rcomp[j]) describes a legitimate resource history. Since the parallel processes interact consistently with each resource, and their only interaction is through the resources, the independent process computations are compatible and can be merged.

Step 4: Proving {P}T*{Q}.

1. $\{\text{pre}(T^*)\}T^*\{\text{post}(T^*)\}$ — step 3

2. P $\vdash$ pre (T*) lemma 6.6

3. post (T*) $\vdash$ Q lemma 6.7

4. {P}T*{Q} 1,2,3,A0

Lemma 6.6: $\forall$v(P(v) => pre(T*)(v)) using the computation C = ({T*},v) to satisfy the definition of pre(T*)(v). Thus P $\vdash_D$ pre(T*).

Lemma 6.7: $\forall$v, post(T*)(v) $\equiv$ $\exists$ a computation C = $(c_0,v_0),\ldots,(c_n,v_n)$ such that $P(v_0)$ holds and v = v_n and C finishes T*. But then Q(v) holds, since {P}T*{Q} is true for the interpreter. So post(T*) $\vdash_D$ Q.

Step 5: Proving {P}T{Q}.

1. {P}T*{Q} step 4

2. {P}T{Q} 1,A8

End of Theorem 6.1.

6.2 Corollary (Relative completeness):

Let T be a program with an enumerable data domain and recursive data operations. If {P}T{Q} is true for the interpreter, it can be proved using A0-A8 and a proof system D which is complete for both the data types of T and the natural numbers.

Proof: Let e be an enumeration of the data domain of T. Add auxiliary variables as before, except to update the history variable x (x = Pcomp[i], Rcomp[j], or comp[i,j]) use

$x:= x||Ptime[i]||e(x_1)||\ldots||e(x_m)$;

This use of the enumeration function e is required because concatenation is defined on natural numbers. The remainder of the proof proceeds exactly as before.

This completeness result is similar to Cook's result for sequential languages, but there are some significant differences. First, Cook did not specify the languages to be used for assertions and for expressions in the programming language, requiring only that the assertion language be expressive, i.e. powerful enough to express the required assertions. This paper has tied both languages to the natural numbers, because they provide a convenient way of encoding program histories for auxiliary variables. A second difference is in the method of proving the completeness theorem. With sequential programs, it is possible to derive the predicate post(S) from S and pre(S), independent of the remainder of program T*; for parallel programs post(S) may depend on all of T*. A final difference is in the use of auxiliary variables, which are not required in sequential programs. The need for auxiliary variables can be avoided by attaching assertions to global program control points (see [1], [2]) rather than to control points in each process as was done here. In this approach, however, the number of assertions can grow exponentially with program size. Keller [13] and Lamport [14] avoid the need for auxiliary variables by allowing assertions to include special variables which are essentially program counters for the parallel processes. These techniques, however, lack an attractive feature of the deductive approach -- that the reasoning required for program verification closely resembles and may even guide the reasoning required in program creation.

7. REFERENCES

[1] Ashcroft, E.A. and Z. Manna. Formalization of properties of parallel programs. Machine Intelligence 6, pp. 17-41, University of Edinburgh Press, 1971.

[2] Ashcroft, E.A. Proving assertions about parallel programs. Technical Report CS-73-01. Department of Applied Analysis and Computer Science, University of Waterloo, 1973.

[3] Brinch Hansen,P. Concurrent Programming Concepts. Computing Surveys 5:4, pp. 223-245, 1973.

[4] Cadiou, J.M. and J.J. Lévy. Mechanizable proofs about parallel processes. Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 34-48, 1973.

[5] Cook, S.A. Axiomatic and interpretive semantics for an Algol Fragment. Technical Report 79, Department of Computer Science, University of Toronto, 1975.

[6] Donahue, J.E. Complementary definitions of programming language semantics. Ph.D. Thesis, University of Toronto, 1975.

[7] Gorelick, G. A complete axiomatic system for proving assertions about recursive and non recursive programs. Technical Report 75, Department of Computer Science, University of Toronto, 1975.

[8] Gries, D. An exercise in proving properties of parallel programs. Submitted to Comm. of the ACM.

[9] Hoare, C.A.R. An axiomatic basis for computer programming. Comm. of the ACM 12:10, pp. 576-580, 1969.

[10] Hoare, C.A.R. Towards a theory of parallel programming, in Hoare and Perrott, ed. Operating System Techniques, Academic Press, 1972.

[11] Hoare, C.A.R. and P.E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. Acta Informatica 3, pp. 135-153, 1974.

[12] Howard, J.H. Proving Monitors. To appear in Comm. of the ACM, May, 1976.

[13] Keller, R.M. On formal verification of parallel programs. Technical Report 186, Computer Science Laboratory, Princeton University, 1975.

[14] Lamport, L. Formal correctness proofs for multiprocess algorithms. Submitted to Second International Symposium on Programming, Paris, 13-15 April, 1976.

[15] Lauer, H.C. Correctness in operating systems. Ph.D. Thesis, Carnegie-Mellon University, 1974.

[16] Lauer, P.E. Consistent formal theories of the semantics of programming languages. IBM Laboratory Vienna TR 25.121, 1975.

[17] Lipton, R.J. Reduction: a new method for proving properties of systems of processes. Yale Computer Science Research Report 30, 1974.

[18] Owicki, S.S. Axiomatic proof techniques for parallel programs. Ph.D. Thesis, Cornell University, 1975.

[19] Owicki, S. and Gries, D. Verifying properties of parallel programs: an axiomatic approach. Proceedings of the Fifth Symposium on Operating Systems Principles, Austin, Texas, 1975. To appear in Comm. of the ACM, May, 1976.

[20] Rosen, B.K. Correctness of parallel programs: the Church-Rosser approach. To appear in Theoretical Computer Science.

Appendix: Proof of Lemma 6.5

We must show that

$$\text{post}(T_1)(v) \wedge \ldots \wedge \text{post}(T_N)(v) \wedge I(r_1)(v) \wedge \ldots \wedge I(r_M)(v) \Rightarrow \text{post}(T_0)(v)$$

where T_0 is the parbegin statement in T*.

There are certain facts about the variable state v which can be deduced from $\text{post}(T_i)(v) \wedge I(r_j)(v) = \textbf{true}$.

1) $\text{post}(T_i)(v) \Rightarrow \exists\, PC^i = (c_0^i, v_0^i), \ldots, (c_{n(i)}^i, v_{n(i)}^i)$ such that $\text{pre}(T_0)(v_0^i) =$ true and PC^i finishes T_i and $v_{n(i)}^i(x) = v(x)\ \forall x \in \text{Proof-var}(T_i)$.

2) $v(\text{Pcomp}[i]) = v_{n(i)}^i(\text{Pcomp}[i])$, since $\text{Pcomp}[i] \in \text{Proof-var}(T_i)$. Since v(Pcomp[i]) is the result of a process computation for T_i, it must have the following form.

   a. $v(\text{Pcomp}[i]) = b_1{}^i || \ldots || b_n{}^i$, where $b_k^i = \text{num}(CS_k) || t_k || y_{k,1} || \ldots || y_{k,m}$. Here CS_k is a critical section in process T_i for some resource r_j with m variables.

   b. $t_k < t_{k+1} \quad 1 \leq k < n$

   c. In PC^i, critical sections are started and finished in the order given by v(Pcomp[i]), and at start and finish the resource variables have the values recorded in the corresponding elements of v(Pcomp[i]), i.e. in $x_{k,1}, \ldots, x_{k,m}$.

3) v(Pcomp[i]) gives the same history as v(comp[i,j]) for critical sections for resource r_j in process T_i. More precisely, comp[i,j] is the subsequence of v(Pcomp[i]) consisting of b_k^i's whose CS_k is a critical section for r_j. This must be true since both v(Pcomp[i]) and v(comp[i,j]) are the result of a process computation for T_i.

4) v(Rcomp[j]) gives the same history as v(Pcomp[i]) for critical sections on resource r_j in process T_i. This follows from 3 above and from part 2 of $I(r_j)(v)$.

From the N process computations PC^i, there are N sequences of statements $D^i = S^i_1,\ldots,S^i_{n(i)}$, $1\le i\le N$ such that in PC^i, $(c^i_k,v^i_k) = \delta(S^i_k,(c^i_{k-1},v^i_{k-1}))$. We want to merge the D^i to obtain a single sequence of statements D which determine a computation for T_0. This will be done by merging the statements according to the "time" at which they were executed.

Definition: Let S^i_k be a statement in the sequence D^i. If S^i_k starts or finishes a critical section in PC^i, let $b^i_k = (\mathrm{num}(CS_k)||t_k||x_{k,1}||\ldots||x_{k,m})$ be the corresponding element of v(Pcomp[i]); then $\underline{\mathrm{time}}(S^i_k) = t_k$. For all other S^i_k, $\underline{\mathrm{time}}(S^i_1) = 0$ and $\underline{\mathrm{time}}(S^i_k) = \mathrm{time}(S^i_{k-1})$.

Note that $\mathrm{time}(S^i_k)$ is non-decreasing with increasing k, from 2b+c above. Now the statement sequences D^i, $1\le i\le N$, can be merged to yield a single sequence $D = S_1,S_2,\ldots,S_n$ in a way that preserves the time ordering, i.e. if S^i_k precedes S^j_k in D, $\mathrm{time}(S^i_k) \le \mathrm{time}(S^j_k)$. Then 2c and 4 above imply

5) the statements in D start and finish critical sections for resource r_j in the order given by v(Rcomp[j]).

Now the computation C for T_0 is defined as follows. From $\mathrm{post}(T_1)(v)$, v^1_0 satisfies $\mathrm{pre}(T_0)(v^1_0)$. This means that there is a computation $C' = (c_0,v_0),\ldots,(c_m,v_m)$ for T* with $P(v_0)$ = true and $v_m = v^1_0$ and T_0 current after C'. Let $C = (c_0,v_0),\ldots,(c_m,v_m),(c_{m+1},v_{m+1}),\ldots,(c_{m+n+1},v_{m+n+1})$, where

$(c_0,v_0),\ldots,(c_m,v_m)$ come from C'

$(c_{m+1},v_{m+1}) = \delta(T_0,(c_m,v_m))$

$(c_{m+k+1},v_{m+k+1}) = \delta(S_k,(c_{m+k},v_{m+k}))$

$1 \le k \le n.$

If we can show that C defined in this way is really a computation (i.e. $\delta(S_k,(c_{m+k},v_{m+k}))$ is defined for $1\le k\le n$), and that C's final state is v, then $\mathrm{post}(T_0)(v)$ will be established. This is proved using induction.

Induction hypothesis: Consider (c_{m+k+1},v_{m+k+1}) $0\le k\le n$. Let S^i_h be the last statement from process i executed in reaching (c_{m+k+1},v_{m+k+1}) in C. Then

a) $(c_{m+k+1},v_{m+k+1}) = \delta(S_k,(c_{m+k},v_{m+k}))$ is defined, $1\le k\le n$.

b) $c^i_h \subseteq c_{m+k+1}$, i.e. the statement current in PC^i after S^i_h is current in C after S_k.

c) $v^i_h(x) = v_{m+k+1}(x)$ $\forall x \in \mathrm{Proof\text{-}var}(S^i_h)$

d) If no critical section on r_j is in execution in (c_{m+k+1},v_{m+k+1}), let p be the number of critical sections on r_j which have been executed in C before S_k. If p = 0, the values of the variables of r in v_{m+k+1} are the values recorded in initstate. If p>0, they are the values recorded in Rcomp[j], i.e. $x_{2p,1},\ldots,x_{2p,m}$.

Base step: For k = 0, a) does not apply. b) holds because $c^i_0 = \{T_i\}$ and $c_{m+1} = \{T_1,T_2,\ldots,T_N\}$. c)+d) hold because $v^i_0(x)$ = value recorded in v(initstate).

Induction step: If S_k is not a critical section statement, a)-d) follow from the induction hypothesis, since executing S_k has the same effect in C as in PC^i. If S_k is the critical section with r_j when B do S, a) holds if no critical section on r_j is in execution in c_{m+k} and if $B(v_{m+k})$ = true. The first condition is satisfied because C starts and finishes critical sections for r_j in the order given by v(Rcomp[j]) (5 above), and in v(Rcomp[j]) a critical section is not started before the last was finished (by $I(r_j)(v)$).

For the second condition, suppose S_k comes from process T_i. Then $B(v_h^i)$ = true from the fact that PC^i is a process computation. Now $v_h^i(x) = v_{m+k+1}(x)\ \forall x \in \text{Proof-var}(S_k) \sim r$ (by c of the induction hypothesis). For $x \in r$,

$v_{m+k}(x)$ = value recorded in Rcomp[j] or initstate (induction-d)

= value recorded in Pcomp[i] (4 above)

$= v_h^i(x)$ (2c above)

So $v_{m+k}(x) = v_h^i(x)\ \forall x \in \text{Proof-var}(S_k)$. Then $B(v_{m+k})$ = true, and a) is satisfied. b)-d) follow from the fact that executing S_k has the same effect in C as in PC^i.

To finish the proof of the lemma we must show that $v = v_{m+n+1}$. Now $v_{m+n+1}(x) = v^i_{n(i)}(x) = v(x)$ $\forall x \in \text{Proof-var}(T_i)$, from induction hypothesis c and 1 above. From induction hypothesis d and $I(r_j)(v)$, $v_{m+n+1}(x)$ = last value in Rcomp[j] $= v(x)\ \forall x \in r_j$. Since every variable in T* belongs to either a process or a resource, $v_{n+m+1} = v$.

This finishes the proof of lemma 6.5.