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 desirable 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.

2. THE PROGRAMMING LANGUAGE

The programming language used is derived from Algol 68. It contains the usual assignment, conditional, while, 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

\[ \text{resource } r_{1} (\text{variable list}), \ldots, \]
\[ r_{m} (\text{variable list}); \]
\[ \text{parbegin } S_{1} // \ldots // S_{n} \text{ parend} \]

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:

with r when B do S

has the following interpretation: r is a resource, B is a Boolean expression, and S is a statement which uses the variables of r. When a process attempts to execute such a statement, it is delayed until the condition B is true and r is not being used by another process. When the process has control of r and B is true, S is executed. Upon termination r is free for further use by other processes.

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 programming 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

$(P') S (Q')$, $(P) S (Q)$

$(P) S (Q)$

A1 assignment

$(P) x =$-E $(P)$

A2 null

$(P) ; (P)$

A3 composition

$(P_1) S_1 (P_2)$, $(P_3) S_3 (P_3)$, ..., $(P_n) S_n (P_n)$

$(P) S_1 ; ; S_n (P_n)$

A4 alternation

$(P \lor B) S_1 (Q_1)$, $(P \lor \neg B) S_2 (Q_2)$

$(P) \text{ if } B \text{ then } S_1 \text{ else } S_2 (Q)$

A5 iteration

$(P \lor B) S (P)$

$(P) \text{ while } B \text{ do } S (P \lor \neg B)$

A6 critical section

$(P \lor B \land I(r)) S (Q \land I(r))$

$(P) \text{ with } r \text{ when } B \text{ do } S (Q)$

A7 parallel

$(P_i) S_i (Q_i) 1 \leq i \leq n$

$(P_1 \land \ldots \land P_n \land \lambda r_1 \ldots \land \lambda r_m)$

resource $r_1$, ..., $r_m$

parbegin $S_1$ // \ldots // $S_n$ parend

$(Q_1 \land \ldots \land Q_n \land \lambda r_1 \ldots \land \lambda r_m)$

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 $\epsilon$ AV $\Rightarrow$ x appears in S' only in assignments y $\leftrightarrow$ E, where y $\epsilon$ AV. 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

$(P) S' (Q)$

$(P) S (Q)$

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
Definition: Let S be the statement

\[ \text{resource } r_1(), \ldots, r_m(): \]
\[ \text{parbegin } S_1 // \ldots // S_n \text{ parend} \]

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

\[ \text{Proof-var}(r_j, S) \]
\[ = \{x: x \text{ is not assigned a value in } S \]
\[ \text{except in a critical section for } r_j \} \]

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

(We will use \text{Proof-var}(r_j, S) and \text{Proof-var}(S', S) when S is obvious from context).

Definition: Let S be a parbegin statement

with processes \( S_1 \ldots S_n \) and Resources \( r_1 \ldots r_m \). Then a proof of \{P \}S_i \{Q \} uses variables safely iff

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

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

Finally, \( \text{A8} \) 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, \( \text{A8} \) 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.

\[ \{x=0\} \]
\[ \text{add1: begin } y:=0; z:=0; \]
\[ \{y=0 \land z=0 \land I(r)\} \]
\[ \text{resource } r(x, y, z): \text{parbegin} \]
\[ \{y=0\} \]
\[ \text{with } r \text{ when true do} \]
\[ \{y=0 \land I(r)\} \]
\[ \text{begin } x:=x+1; \ y:=1 \text{ end} \]
\[ \{y=1\} \]
\[ \{z=0\} \]
\[ \text{with } r \text{ when true do} \]
\[ \{z=0 \land I(r)\} \]
\[ \text{begin } x:=x+1; \ z:=1 \text{ end} \]
\[ \{z=1\} \]
\[ \text{parend} \]
\[ \{y=1 \land z=1 \land I(r)\} \]
\[ \{x=2\} \]
\[ I(r) = \{x=y+z\} \]

Figure 2. Assertions from a proof of \{x=0\} add1 \{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 (non-deterministically) 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 convenience 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 \( \delta \) for program S maps statements X program states to program states.
\[ \delta(T, (c, v)) \] represents the effect of initiating \( T \) in state \( (c, v) \).

\[ \delta(T, (c, v)) = \text{undefined if } T \notin \mathcal{C}, \text{ or if } T \]

is the statement with \( r \) when \( B \)<br>
\[ \text{do } T_1 \text{ and either } B(v) = \text{false or } c \text{ already contains a} \]

statement which belongs to a critical section for \( r \)

\[ = (c', v') \text{ otherwise.} \]

where \( v'(x) = E(v) \) if \( T \) is the statement \n
\[ x = E \]

\[ v(x) \text{ otherwise} \]

\[ c' = (c \cup \{T\}) \cup \text{successor } (T, (c, v)) \]

where successor \( (T, (c, v)) \) is the set of statements to be initiated after \( T \). For example,

\[ \text{a) if } T \text{ is } B \text{ then } T_1 \text{ else } T_2, \]

\[ \text{successor } (T, (c, v)) = \text{if } B(v) \text{ then } \]

\[ T_1 \text{ else } T_2 \]

\[ \text{b) if } T \text{ is } \text{resource } r_1, \dots, r_n; \]

\[ \text{parbegin } T_1 \text{ // } \dots \text{ // } T_n \text{ parend; } \]

\[ \text{successor } (T, (c, v)) = (T_1, \ldots, T_n) \]

\[ \text{c) if } T \text{ is the last statement in } S, \text{ or} \]

the last statement in a set of parallel processes which still have \n
statements in \( c \cup \{T\} \), \text{successor } \n
\[ (S, (c, v)) = \emptyset. \]

Definition: A partial computation of the interpreter for program \( S \) is a sequence of program states, \( C = (c_0, v_0), \)

\[ \ldots, (c_n, v_n), \text{ such that } c_0 = (S), \text{ and} \]

\[ (c_i, v_i) = \delta(S_i, (c_{i-1}, v_{i-1})) \text{ for some} \]

\[ S_i \in C_{i-1}, 1 \leq i \leq n. \text{ A computation} \]

is a partial computation which terminates \n
\[ i.e. c_n = \emptyset. \text{ Note that } C \text{ is completely} \]

determined by \( c_0 \) and the sequence of \n
states \( S_1, S_2, \ldots, S_n \).

This completes the definition of the interpreter. Some related concepts with 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 if any computation \( C = (c_0, v_0), \ldots, (c_n, v_n) \) for which \( P(v_0) \)

is true has \( Q(v_n) = \text{true} \), i.e., any computation \( \text{which starts } S \text{ with } P \) 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 \text{begin } S_1; \ldots; S_n \text{ end} \) is the concatenation of computations for \text{S}_1; \ldots; \text{S}_n. \text{ Rule } A6 \text{ is valid only within the context of a parbegin statement; fortunately this is the only place where critical sections can be used.} \text{ If } A6 \text{ is to be sound, any computation which starts the statement with } r \text{ when } B \text{ do } S \text{ with } P \text{ true must start statement } S \text{ with } P \land A(r) \text{ true.} \text{ This can only be established within a parbegin statement, whose proof includes the requirement that } I(r) \text{ holds} \text{ 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 \( \{P_i\}S'_{Q_i} \text{ } 1 \leq i \leq k \), is derived using \( A0 \) and \( \{P_{i-1}\}S'_{Q_{i-1}} \).

Thus \( P_{k+1} \cdot \ldots \cdot P_1 \) and \( Q_{1} \cdot Q_{2} \cdot \ldots \cdot Q_{k} \). The formula \( \{P\}S'_{Q} \) 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 = (c0,v0),..., (cn, vn) be a partial computation for S with P(v0) = 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'(vn) = true.
   b) if C finishes T, Q'(vn) = 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)(vn) = true.

Note that 1b implies that any computation for S which starts with P 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), v0). By assumption, P(v0) = true.
   If P' is any other pre-condition of S, P ⊢ P' (Lemma 5.1).
   Thus P'(v0) = true.
   1b and 2 do not apply.

Induction step: Let C' = (c0,v0),..., (cn-1,vn-1). By induction, 1 and 2 are satisfied for C'. We must show they are satisfied for C. Let Sn be the state in c0-1 such that (c0,v0) = δ(Sn,c0-vn-1)).

Proof of 1: Consider cases of Sn and T.

Case 1: Sn and T are in different processes. Now vn agrees with vn-1 on all variables in Proof-var(T), since Sn can not change any of those variables.
   a) If T is current after C, it was current after C', and P'(vn-1) = true, by induction. So P'(vn) = true.
   b) If C finishes T then C' finishes T, and by similar reasoning Q'(vn) = true.

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

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

L1: while B do x:=E;
   Sn

After C, i.e., after the execution of Sn, Sn is finished and L1 is current. Thus we must show that the post-conditions of Sn and the pre-conditions of L1 hold in vn.

First, note that the first line in the proof of (P)S(Q) which refers to Sn must have the form (P')S(Q) (Lemma 5.1).

Now Sn is current after C', so by induction Rn_{E_n} = true. Then R(n_{n-1}) = true since vn_{n-1} is obtained from vn_{n-1} by assigning the value of E to x. If R' is any other post-condition of Sn, R ⊢ R' (Lemma 5.1), so that R'(vn_{n}) = true. Now the last line of the proof which refers to Sn is used to derive (P')L1: while B do Sn (P' A-B) using A5 (Lemma 5.1), so it must have the form (P' A-B)S(Q)'. Since P' is a post-condition of Sn, P'(vn_{n}) = true. But P' is also the first pre-condition of L1, so by Lemma 5.1 P' ⊢ P', where P' is any pre-condition of L1; thus P'(vn_{n}) = true.

Example 2: Sn is with B 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' A B A I(r)} T {Q' A I(r)}

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

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

Example 4: C finishes a parbegin statement T. Then Sn finishes the last process of T (all the others were finished by C'). By previous arguments, the post-condition of each process holds for vn; 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 vn.

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', $I(r_1) (V_{n-1}) = true$
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 does S1 was in execution in C', but not in C,
then C finishes S1. Now S1 must have a post-condition with the form $Q' \land I(r)$
(Lemma 5.1), so by lb $I(r) (V_n) = true$.

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 affect the
flow of control or the values assigned to other program variables. Thus $S'$
have 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) = $\phi$.
so that the only post-condition possible
for A1 and A2 is $P' \land I(r) = I(r)$ (Lemma 5.1), and
$\gamma(I(r) => x=2)$, since I(r) must hold
initially, when x=0. Even without the
restrictions on proof-variables, the
strongest valid candidate for post-condi-
tion of A1 and A2 is $P' = \{x<2\}$, and
the strongest valid invariant is
$I(r) = \{0<x<2\}$. But $\gamma(P' \land I(r) => x=2)$,
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 $\mid$ (concatenation, to be defined
shortly). The completeness result is then
generalized to programs with any enumerable
data domain and recursive operations.

The concatenation operation $\mid$ 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 se-
quence in an integer; here we choose to
represent the empty sequence by 0 and the
sequence $a_1,a_2,...,a_k$ (or $a_1 \mid a_2 \mid \ldots \mid a_k$)
by the number

$$
\begin{array}{c}
1111 \ldots 1011 \ldots 01 \ldots 1 \\
\hline
a_1+1 \\
1's
\end{array}
\begin{array}{c}
\hline
a_2+1 \\
1's
\end{array}
\begin{array}{c}
\hline
\vdots
\end{array}
\begin{array}{c}
\hline
a_k+1 \\
1's
\end{array}

Thus 0$\mid 2$ $\mid 1 = 101111011$.

Theorem 6.1: (Relative completeness of
A0-A8 for programs over the natural num-
ers). Let T be a program whose data
domain is the natural numbers with $<$, $=+
$*, and $\mid$. If $(P)T(Q)$ is true for the
interpreter, then $(P)T(Q)$ can be proved
using A0-A8 and a 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 prin-

ciple 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 vari-
ables 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 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 AG, 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 \texttt{parbegin} statement of the form

\begin{verbatim}
T_0: resource r_1,...,r_M;
parbegin T_1/.../T_N paren end
\end{verbatim}

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

1. \texttt{initstate:} records the values of all variables when execution of \texttt{T_0} begins.

2. \texttt{Ptime[1:M] and Rtime[1:M]: "clock" variables which are used to establish the relative times at which events occurred in the execution of \texttt{T_0}.}

3. \texttt{Pcomp[1:M], Rcomp[1:M], comp[1:N,1:M]: sequences which record the history of critical section execution. Pcomp[i] is a history for process \texttt{T_i}, Rcomp[j] is a history for resource \texttt{r_j}, and comp[i,j] is a history of critical sections in process \texttt{T_i} which involve resource \texttt{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 \texttt{T^*} required in the proof of \((P)T(Q)\) is obtained as follows.

1. If \texttt{T} contains no \texttt{parbegin} statement, \texttt{T^*} = \texttt{T}.

2. Otherwise, replace the single \texttt{parbegin} statement \texttt{T_0} by

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

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

\begin{verbatim}
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)|
  Rcomp[j]:=Rcomp[j]|num(CS)|
  Ptime[i]|\texttt{y_1}||...||\texttt{y_m}|
  Rtime[j]|\texttt{y_1}||...||\texttt{y_m}|
  Pcomp[i]|\texttt{y_1}||...||\texttt{y_m}|
  Rcomp[j]|\texttt{y_1}||...||\texttt{y_m}|
end
\end{verbatim}

Thus the variable state before parallel execution begins is recorded in \texttt{initstate}, while the history of resource use during parallel execution is recorded in \texttt{Pcomp} and \texttt{Rcomp}. The final values of \texttt{Pcomp[i]} and \texttt{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, \texttt{Ptime[i]} and \texttt{Rtime[j]}, are updated in such a way that the times recorded in \texttt{Pcomp[i]} and \texttt{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 \texttt{initstate}, \texttt{Pcomp[i]}, and \texttt{comp[i,j]} belong to \texttt{Proof-var(T_i)}, while \texttt{initstate}, \texttt{Rcomp[j]}, and \texttt{comp[i,j]} belong to \texttt{Proof-var(r_j)}.

The variables added in \texttt{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 \texttt{T^*} has the same effect as \texttt{T} on the variables in \texttt{P} and \texttt{Q}. 

79
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 if the following conditions on \( Rcomp[j] \) and \( comp[i,j] \), \( 1 \leq i \leq N \), are satisfied.

1) \( v(Rcomp[j]) = a_{11} | a_{12} | \ldots | a_{2n-1} | a_{2n} \), where
   a) \( a_k = \text{num}(CS_k) | t_k | x_{k,1} | \ldots | x_{k,m} \), \( 1 \leq k \leq 2n \), where \( CS_k \) is the label of a critical section for \( r_j \).
   b) \( CS_{2k-1} = CS_{2k} \), \( 1 \leq k \leq n \).
   c) \( t_k < t_{k+1} \), \( 1 \leq k < 2n \).
   d) \( x_{2k+1,h} = x_{2k,h} \), \( 1 \leq k < n \) and \( 1 \leq h \leq m \).
   e) \( v(y_h) = x_{2n,h} \), \( 1 \leq h \leq m \).

2) For \( 1 \leq i \leq N \), \( v(comp[i,j]) \) 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 \((a_{2k})\) (la and b).

Time is strictly increasing in this history \((lc)\). 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 \((ld)\). 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 \( \{c_0, c_1, \ldots, \}, 0, 1, \ldots\) \( t, +, \ast, \vdash, 0, 1, \ldots\) \( C \). 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 \( \text{Proof-var}(r_j) \), since \( \text{Rcomp}[j] \) and \( \text{comp}[i,j] \) both belong to \( \text{Proof-var}(r_j) \). 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\((S)(v) \equiv S \) 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\((S)(v) \equiv S \) 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 \( \text{Proof-var}(S) \). We will obtain such a definition by using the concept of a process computation.

Definition: Let \( T_0; \text{resource } r_1, \ldots, r_n; \) parbegin \( L_1:T_1; \ldots; L_n:T_n \) paren be the single \( \text{parbegin} \) 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 \( 1 \leq k \leq n \), \( S_k \) such that \( (c_k, v_k) = (S_{k-1}) \). 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_s 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) ≡ 3 a computation process PC = (c_0, v_0), ..., (c_n, v_n) for process T_i such that pre(T_0)(v_0) = true and v(x) = v_n(x) ∀x ∈ Proof-var(S), and S is current in c_n. If S belongs to a critical section CS_r_j for r_j, let (c_k, v_k) be the last state in PC such that (c_k, v_k) = δ(CS_r_j, c_k-1, v_k-1)).

Then I(r_j) (v_k) = true.

post(S)(v) ≡ 3 a computation process PC = (c_0, v_0), ..., (c_n, v_n) for process T_i such that pre(T_0)(v_0) = true and v(x) = v_n(x) ∀x ∈ Proof-var(S) and PC finishes S. If S belongs to a critical section CS_r_j 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)*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 {post(S)} A1
2. {pre(S)} -D post(S) Y Lemma 6.1
3. {pre(S)} y:=E {post(S)} 1,2,A0

Lemma 6.1: We first show that

∀v (pre(S)(v) ⇒ post(S) Y_E(v)).

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

pre(S)(v) ≡ 3 a computation C = (c_0, v_0), ..., (c_n, v_n) such that P(v_0) = true and v = v_n and S is current after C.

⇒ 3 a computation

C' = (c_0, v_0), ..., (c_n, v_n), (c', v')

where (c', v') = δ(S, c_n, v_n)

such that P(v_0) = true and C' finishes S.

⇒ post(S)(v')

⇒ 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

∀v (pre(S)(v) ⇒ post(S) Y_E(v)),

and since D is a complete proof system for the natural numbers, pre(S) -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 |- pre(S') lemma 6.2
3. post(S') |- 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 $v'(x) = v_n(x)$ for all $x$. Then $v'(x) = v(x) \forall x \in r$. 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 $v(x) = v'(x) \forall x \in \text{Proof-var}(S')$, this yields $pre(S')(v)$. Thus

$$\forall v (pre(S)(v) \land B \land \text{I}(r) \implies pre(S')(v)), \quad \text{so pre}(S) \land B \land \text{I}(r) \vdash \text{D} \text{ pre}(S').$$

Lemma 6.3: Let $v$ be a variable state with $post(S')(v) = \text{true}$, 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 $I(r_j)(v) = \text{true}$, note that $post(S')(v) = I(r_j)(v_k)$, where $v_k$ is the variable state of $PC$ upon starting $S$.

Now executing $S$ preserves $I(r_j)(v)$ (check the definition of $T^*$), yielding $I(r_j)(v_n)$.

Since $v(x) = v_n(x) \forall x \in \text{Proof-var}(S')$, and $\text{Proof-var}(r_j) \subseteq \text{Proof-var}(S')$, this implies $I(r_j)(v)$. Thus

$$\text{post}(S') \vdash \text{D} \text{ post}(S) \land \text{I}(r_j).$$

Case 3: $S$ is

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

1. $(pre(T_1)) T_1 \{post(T_1)\}, \quad 1 \leq i \leq N$

   induction

2. $(pre(T_1) \land \ldots \land pre(T_N) \land I(r_1) \land \ldots \land I(r_M)) \implies$
   
   $(post(T_1) \land \ldots \land post(T_N) \land I(r_1) \land \ldots \land I(r_M))$

   $1, A7

3. $pre(S) \vdash pre(T_1) \land \ldots \land pre(T_N) \land I(r_1) \land \ldots \land I(r_M)$

   lemma 6.4

4. $post(T_1) \land \ldots \land post(T_N) \land I(r_1) \land \ldots \land I(r_M) \vdash post(S)$

   lemma 6.5

5. $(pre(S)) \{post(S)\}$

   2, 3, 4, A0

Lemma 6.4: $pre(T_1)(v) = \text{pre}(T_1)(v)$ immediately from the definitions.

pre$(T_1)(v) = I(r_j)(v)$ 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 $v(\text{init-state})$. Thus $I(r_j)(v) = \text{true}$.

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

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

we must derive a computation for $T^*$ which finishes $T_0$ in state $v$; this will be done by merging the process computations for $T_i, 1 \leq i \leq N$, whose existence is implied by $\text{post}(T_i)$. This merger will preserve the ordering of statements by "time" which can be inferred from the times stored in $v(\text{Pcomp}[i])$. 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, $\land \text{post}(T_i)(v)$ implies that all $i=1$

the processes started $T_0$ with the same initial state: the one recorded in $v(\text{init-state})$. Post$(T_i)(v)$ also implies that $v(P\text{comp}[i])$ is a history of critical section execution in $T_i$, and that for $1 \leq j \leq M$, $v(P\text{comp}[i])$ gives the same history as $v(\text{comp}[i,j])$ for critical sections for resource $r_j$. Now $I(r_j)(v)$ implies that $v(R\text{comp}[j])$ gives the same history as $v(\text{comp}[i,j])$ for critical sections in process $T_i$. Thus $v(P\text{comp}[i])$ and $v(R\text{comp}[j])$ 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 $v(R\text{comp}[j])$, and by $I(r_j)(v)$, $v(R\text{comp}[j])$ 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. $(pre(T^*)) T^* \{post(T*)\} \quad \text{step 3}$
2. \( P \vdash \text{pre}(T^*) \) lemma 6.6
3. \( \text{post}(T^*) \vdash Q \) lemma 6.7
4. \( (P)T^*(Q) \)
   \(1,2,3,4,5\)

Lemma 6.6: \( \forall v (P(v) \Rightarrow \text{pre}(T^*)(v)) \) using the computation \( C = ((T^*),v) \) to satisfy the definition of \( \text{pre}(T^*)(v) \). Thus \( P \vdash D \text{pre}(T^*) \).

Lemma 6.7: \( \forall v, \text{post}(T^*)(v) \equiv 3 \) 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 \( \text{post}(T^*) \vdash D Q \).

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 \( x \) \( (x = F\text{comp}[i], \text{Rcomp}[i], \text{comp}[i, j]) \) use

\[ x := x || \text{Time}[i] || e(x_1) \ldots e(x_n) \]

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 \( \text{post}(S) \) from \( S \) and \( \text{pre}(S) \), independent of the remainder of program \( T^* \); for parallel programs \( \text{post}(S) \) 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


There are certain facts about the variable state $v$ which can be deduced from
post($T_1$)($v$) $\land$ I($r_j$)($v$) = true.

1) post($T_1$)($v$) $\Rightarrow$ $\exists$ PC$^i$ = ($c_0^i$, $v_0^i$),...,($c_n^i$, $v_n^i$) such that pre($T_0$)($v_0^i$) $=$ true and PC$^i$ finishes $T_1$ and $v_n^i$($x$) = $v(x)$ $\forall x \in$ Proof-var($T_1$).

2) v(Pcomp[i]) = $v_n^i$($n$)($Pcomp[i]$), since Pcomp[i] $\in$ Proof-var($T_1$). Since v(Pcomp[i]) is the result of a process computation for $T_1$, it must have the following form.

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

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

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 $X_{k,1},...,X_{k,m}$

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 $C_k$ 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\textsuperscript{i}, there are N sequences of statements
\[ D^i = S^i_1, \ldots, S^i_N, \quad 1 \leq i \leq N \]
such that in
\[ PC^i, \quad (c^i_k, v^i_k) = \delta(S^i_k, (c^i_{k-1}, v^i_{k-1})). \]
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
\[
\text{time}(S^i_k) = \text{time}(S^i_{k+1}) = \cdots = \text{time}(S^i_{k+m})
\]
be the corresponding element of \( \text{v}(\text{comp}[i]) \).

Then \( \text{time}(S^i_k) = t_k \). For all other \( S^i_k \),
\[
\text{time}(S^i_k) = 0 \quad \text{and} \quad \text{time}(S^i_{k+1}) = \text{time}(S^i_{k-1}).
\]

Note that \( \text{time}(S^i_k) \) is non-decreasing with increasing \( k \), from \( 2b+c \) above. Now the statement sequences \( D^i, 1 \leq i \leq N \), 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 \( D \), \( \text{time}(S^i_k) < \text{time}(S^j_k) \). 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 \( v(\text{comp}[j]) \).

Now the computation \( C \) for \( T_0 \) is defined as follows. From \( \text{post}(T_0)(v) \),
\[
v_0 \]
satisfies \( \text{pre}(T_0)(v_0) \). This means that there is a computation \( C' = (c_0, v_0), \ldots, (c_m, v_m) \)
for \( T^* \) with \( \text{P}(v_0) = \text{true} \) and \( v_m = v_0 \), 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
\[
(c_0, v_0), \ldots, (c_m, v_m) \quad \text{come from} \quad C',
\]
\[
(c_{m+1}, v_{m+1}) = \delta(T_0, (c_m, v_m))
\]
\[
(c_{m+k+1}, v_{m+k+1}) = \delta(S_k, (c_{m+k}, v_{m+k}))
\]
\[ 1 \leq k \leq n. \]

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 \( 1 \leq k \leq n \), and that \( C \)'s final state is \( v \), then its post \( (T_0)(v) \) will be established. This is proved using induction.

**Induction hypothesis:** Consider
\[ (c_{m+k+1}, v_{m+k+1}) \quad 0 \leq k \leq n. \]
Let \( S^i_k \) 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, \( 1 \leq k \leq n. \)

b) \[ S^i_k \subseteq c_{m+k+1}, \text{i.e. the statement current in } PC^i \text{ after } S^i_k \text{ is current in } C \text{ after } S^i_k. \]

c) \[ v^i_h(x) = v_{m+k+1}(x) \quad \forall x \in \text{Proof-\var(S^i_k)} \]
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 \( S^i_k \). 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 \( R(\text{comp}[j]) \), i.e. \( x_{2p+1}, \ldots, x_{2p,m} \).

**Base step:** For \( k = 0 \), a) does not apply.

b) holds because \( c^i_0 = \{T_1\} \)
and
\[ c^i_0 = \{T_1, T_2, \ldots, T_N\}. \]

**Induction step:** If \( S^i_k \) is not a critical section statement, a)-d) follow from the induction hypothesis, since executing \( S^i_k \) has the same effect in \( C \) as in \( PC^i \). If \( S^i_k \) 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 \( B(v_{m+k}) = \text{true} \). The first condition is satisfied because \( C \) starts and finishes critical sections for \( r_j \) in the order given by \( v(\text{comp}[j]) \) (5 above), and in \( v(\text{comp}[j]) \)
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 $B(v^i_h) = true$ from the fact that $FC^i$ is a process computation.

Now $v^i_h(x) = v^i_{m+k+1}(x) \forall x \in Proof-var(S_k) \forall r$ (by c of the induction hypothesis). For $x \in r$,

$$v^i_{m+k}(x) = \text{value recorded in Rcomp[j] or initstate (induction-d)}$$

$$= \text{value recorded in Pcomp[i] (4 above)}$$

$$= v^i_h(x) \quad (2c \text{ above})$$

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

To finish the proof of the lemma we must show that $v = v^i_{m+n+1}$. Now

$$v^i_{m+n+1}(x) = v^i_{n(i)}(x) = v(x)$$

$\forall x \in Proof-var(T_i)$, from induction hypothesis c and 1 above. From induction hypothesis d and $I(r_j)(v)$, $v^i_{m+n+1}(x) =$ 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,

$$v^i_{n+m+1} = v. $$

This finishes the proof of lemma 6.5.